A Unified Approach to Dynamic Point Location, Ray Shooting and Shortest Paths in Planar Maps

Yi-Jen Chiang, Franco P. Preparata and Roberto Tamassia

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-07
January 1992

A Unified Approach to Dynamic Point Location, Ray Shooting, and Shortest Paths in Planar Maps*

Yi-Jen Chiang   Franco P. Preparata   Roberto Tamassia

Department of Computer Science
Brown University
Providence, RI 02912–1910
{yjc,franco,rt}@cs.brown.edu

Abstract

We describe a new technique for dynamically maintaining the trapezoidal decomposition of a planar map $\mathcal{M}$ with n vertices, and apply it to the development of a unified dynamic data structure that supports point-location, ray-shooting, and shortest-path queries in $\mathcal{M}$. The space requirement is $O(n \log n)$. Point-location queries take time $O(\log n)$. Ray-shooting and shortest-path queries take time $O(\log^3 n)$ (plus $O(k)$ time if the k edges of the shortest path are reported in addition to its length). Updates consist of insertions and deletions of vertices and edges, and take $O(\log^3 n)$ time (amortized for vertex updates).

(December 1991)

*Research supported in part by the National Science Foundation under grants CCR–90–07851 and CCR–91–96176, by the U.S. Army Research Office under grant DAAL03-91–G–0035, by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014–91–J–4052, ARPA order 8225.

1 Introduction

A number of operations within the context of planar maps (or subdivisions, as determined by a planar graph embedded in the plane) have long been regarded as important primitives in computational geometry. First and foremost among these operations is planar point-location, i.e., the identification of the map region containing a given query point; but also shortest-path and ray-shooting queries have been considered very prominently.

Starting with the pioneering work in planar point-location of the seventies [10, 18], over the years several techniques have been developed, culminating with asymptotically time- and space-optimal methods [12, 17, 28] that are also of sufficiently practical flavor. Such methods, however, refer to the *static* case where no alteration of the map is allowed during its use. Due to the obvious importance of the *dynamic* setting, in recent years considerable attention has been devoted to the development of dynamic point-location algorithms [6, 8, 14, 15, 21, 25, 26, 30].

The best results to-date for dynamic point-location in an n-vertex map are due to Cheng-Janardan [6], with $O(\log^2 n)$ query-time, $O(\log n)$ update time, and $O(n)$ space. Baumgarten, Jung, and Mehlhorn are reported to have recently developed a fractional-cascading method with $O(\log n \log\log n)$ query-time and polylog amortized update time [2]. In many real-time applications, point-location queries are executed more frequently than updates, so that it is often desirable to achieve optimal $O(\log n)$ query time in a dynamic setting. The only previous techniques that support $O(\log n)$-time queries in a dynamic environment are restricted to monotone maps [8, 15]. For a survey of dynamic point location techniques and other dynamic algorithms in computational geometry, see the paper of Chiang and Tamassia [9].

Algorithmic research on shortest-path and ray-shooting queries has also experienced steady progress, resulting in time-optimal techniques for the static setting [1, 5, 7, 16, 19]. In particular, the linear-space data structures of Chazelle-Guibas [5] and of Guibas-Hershberger [16] support in $O(\log n)$ time ray-shooting and shortest path queries, respectively, in a simple polygon with n vertices. No polylog-time method was previously known to perform such important queries in a dynamic setting. Sublinear-time techniques are known only for ray shooting queries. The known dynamic methods have $O(\sqrt{n}\,\mathrm{polylog}(n))$ query/update time [1, 7] and support ray-shooting in a set of possibly intersecting segments without taking advantage of the structure of planar maps.

A property that appears to greatly facilitate the development of dynamic point-location techniques is *monotonicity* ([8, 15, 25]). Whereas the restriction to monotone maps is quite adequate for many important applications (e.g., convex maps, such as Voronoi diagrams), yet the exclusion of general maps is a severe shortcoming. In the static case, a general map can be reduced to monotone (or, as we say in this paper, *normalized*) by the straightforward insertion of (auxiliary) diagonals. The same approach, when attempted for the dynamic setting, could lead to onerous updates, such as when the insertion of an edge causes the removal of a very large number of normalizing diagonals. A rather complicated and only partially documented technique due to Fries [13], is reported to assure that only a logarithmic number of normalizing diagonals be involved in any update.

The approach reported in this paper combines the feature just stated with the underpinnings of the trapezoid method, whose search efficiency both in theory [4, 23] and practice [11] is well-established. This leads to the adoption of horizontal normalizing diagonals, called *lids*. The method rests on three major components:

1. A *normalization structure* that transforms a general map into a monotone one by the addition of horizontal diagonals, while guaranteeing that no more than a logarithmic number of such diagonals are affected by insertions/deletions of edges/vertices.
2. A *hull structure* that stores the convex hulls of the chains and subchains of the monotone

subregions, so that ray-shooting and shortest-path queries can be efficiently performed.

3. A *location structure* that represents with a balanced tree a recursive decomposition of the normalized map into trapezoidal regions, and supports point-location queries in optimal time.

It is important to underscore that a single tree sturcture—the normalization structure—provides the unifying framework for the three applications considered. In fact, this structure, while ensuring efficient updates by controlling the size of the modifications, can be natually augmented with node-appended secondary structures to support shortest-path and ray-shooting queries. It can also be supplemented with a distinct, but tightly complied, structure designed for efficient point-location. The main normalization structure and its two auxiliary components act in a tightly integrated fashion: indeed, we will show that point-location is crucially used in shortest-path and ray-shooting queries and in the update of the normalization structure.

The fundamental constituents of our data structures are monotone chains, and trapezoids determined by edges and horizontal lines through vertices. This provides the unifying framework for the three applications mentioned earlier. Indeed, a simple augmentation of the normalization structure provides the right environment for all three queries, as we shall illustrate. It should be underscored that, although their linkings are obviously elaborate, the elementary data structures employed are particularly simple, so that not only asymptotic efficiency is established, but also practical potential is apparent.

Our main results are outlined in the following theorem:

Theorem 1 *There exists a fully dynamic data structure that supports point-location, ray-shooting, and shortest-path queries in a planar map $\mathcal{M}$ with n vertices. The space requirement is $O(n \log n)$. Point-location queries take time $O(\log n)$. Ray-shooting and shortest-path queries take time $O(\log^3 n)$ (plus $O(k)$ time if the k edges of a shortest path are reported in addition to its length). Updates take $O(\log^3 n)$ time (amortized for vertex updates).*

As a corollary, we can also perform stabbing queries, i.e., determine the k edges of map $\mathcal{M}$ intersected by a query segment, in $O(k \log^3 n)$ time.

The contributions of this work can be summarized in the following points:

- We present the first polylog-time dynamic data structure for shortest-path queries in planar maps. All previous data structures for shortest paths are static and take linear time for either queries or updates when used in a dynamic environment.
- We provide the first polylog-time dynamic data structure for ray-shooting queries in planar maps. The previous best result is $O(\sqrt{n}\,\mathrm{polylog}(n))$ query time.
- We present the first dynamic data structure for point location queries in general planar maps with optimal $O(\log n)$ query time and polylog update time. The previous best result is $O(\log n \log\log n)$ query time.
- We provide the first dynamic point-location data structure that checks the validity of an edge insertion, i.e., whether the new edge does not intersect the current edges of the map. Previous dynamic point location data structures did not have such a capability due to the lack of an efficient dynamic ray-shooting technique.

In Section 2 we briefly review the terminology of planar maps and the basic data structures used by our method. The mechanics of the dynamic maintenance of a normalized map are described in Section 3, while Sections 4, 5, and 6 are respectively devoted to shortest-path, point-location, and ray-shooting queries.

2 Review of Background

For the geometric terminology used in this paper, see [24]. A *planar map* $\mathcal{M}$ is a subdivision of the plane into polygonal regions whose underlying planar graph is connected. Thus, each region r of $\mathcal{M}$, bounded or unbounded, is a simple polygon. In the following, n denotes the number of vertices of the planar map $\mathcal{M}$ curently being considered.

In the plane we have an orthogonal frame of reference (x, y). A polygonal chain γ is *monotone* if an arbitrary horizontal line intersects it either in a single point or in a single interval. A simple polygon r is *monotone* if its boundary consists of two monotone chains. A *cusp* of a polygon is a vertex v whose internal angle is greater than π and whose adjacent vertices are both strictly above (lower cusp) or strictly below (upper cusp) v. A polygon is monotone if and only if it has no cusps. A map is monotone if all its regions are monotone.

The *trapezoidal decomposition* of a map $\mathcal{M}$ is obtained by drawing from each vertex v of $\mathcal{M}$ two horizontal rays that terminate when they first meet edges of $\mathcal{M}$: the resulting segments are called *splitters*. It is easily verified that a region of $\mathcal{M}$ with s vertices is partitioned by the splitters into $s - 1$ trapezoids (see Fig. 1). The trapezoidal decomposition of $\mathcal{M}$ is geometrically dualized by mapping each of the obtained trapezoids τ to an arbitrary point $\delta(\tau)$ in the interior of τ: each of the splitters is mapped to an edge between images of trapezoids in the usual way. We let $\delta(\mathcal{M})$ denote the resulting dual graph, which is a forest of trees since the trapezoids of a single region $r \in \mathcal{M}$ dualize to a tree $\delta(r)$ (because r has no holes). Note that each node of $\delta(r)$ has degree at most four (barring degeneracies). Given two splitters s_1 and s_2 contained within the same region r, $\text{SLEEVE}(s_1, s_2)$ denotes the union of the trapezoids traversed by the shortest path within r between any point of s_1 and any point of s_2. (Note the duality between "sleeves" in region r and paths in tree $\delta(r)$.) In a notationally consistent manner, $\delta(s)$ denotes the edge of $\delta(r)$ that is the dual of splitter s.

Our data structures are based on a variety of balanced search trees. We observe that all the standard operations on balanced search trees (insertion, deletion, split, and join) can be performed by means of a logarithmic number of more basic primitives, which we call "elementary joins and splits", defined as follows:

- An *elementary join* of two binary trees T_1 and T_2 forms a new tree T by making T_1 and T_2 the left and right subtrees of a new root node.
- An *elementary split* yields the left and right subtrees T_1 and T_2 of T by removing its root.

In particular, a simple rotation can be viewed as a sequence of four elementary splits and joins.

Two special types of balanced binary trees will be used in this paper: biased binary trees [3] and $BB[\alpha]$-trees [20].

A *biased binary tree* [3] is a binary search tree whose leaves store weighted items. Let w be the sum of all weights. We have that the depth of a leaf with weight w_i is at most $\log(w/w_i) + 2$, and each of the following update operations can be done in $O(\log w)$ time: change of the weight of an item, insertion/deletion of an item, and split/splice of two biased trees [3].

A $BB[\alpha]$-tree [20] (where α is a fixed real, with $\frac{1}{4} < \alpha \le 1 - \frac{\sqrt{2}}{2}$) is a binary search tree and has the following important properties (among others):

- A $BB[\alpha]$-tree with n nodes has height $O(\log n)$.
- Assume that we augment a $BB[\alpha]$-tree with secondary structures stored at its nodes. Let the subtree with root μ have ℓ leaves, and let the time for updating the secondary structures after a rotation at node μ be $O(\ell \log \ell)$. Then the amortized time of an update operation in a sequence of n insertions and deletions starting from an initially empty $BB[\alpha]$-tree is $O(\log^2 n)$.

3 The Dynamics of Trapezoidal Decompositions

Given a general map $\mathcal{M}$, our objective is first to systematically transform *(normalize)* it into a monotone map, and then to illustrate how to efficiently maintain it under a dynamic regimen of edge and vertex insertions/deletions.

3.1 Normalization

We first address the problem of normalization. Each region r of $\mathcal{M}$ is handled individually. We refer to a bounded region r; handling of the unbounded region involves trivial adaptations. In the following, we denote by m the current number of vertices in region r.

We now imagine to represent $\delta(r)$ as a dynamic tree $\Delta(r)$ [29] (see Fig. 1). We choose an arbitrary node of $\delta(r)$ as the *root*, which immediately forces a direction on each edge, referred to as an "arc" and directed toward the root. An arc is usually denoted either by a single letter or by an ordered pair (origin, destination). Next, the arcs are classified: letting $w(\mu)$, *weight* of μ, denote the number of nodes in the subtree rooted at node μ, an arc is classified *heavy* if $w(child) \geq \frac{1}{2}w(parent)$, and *light* otherwise. The arcs are also classified as *solid* or *dashed* such that at most one solid arc enters a node. The maximal paths of solid arcs (possibly consisting of a single node) are called *solid paths*, and correspond to sleeves of r. Note that each solid path is followed by a dashed arc, unless it contains the root. In the normal status, the following *weight invariant* holds: heavy arcs are solid and light arcs are dashed. However, during the execution of dynamic operations, we are allowed to change heavy arcs to dashed and light arcs to solid, and thus loose the original correspondence. We can restore the weight invariant with the *conceal* operation (to be discussed later).

We insert into r a set of splitters, called *lids*, which are the duals of the following arcs:

1. All light arcs (Rule 1).
2. Any two consecutive heavy arcs whose y-components have opposite directions (Rule 2).

Note that each lid is generated by a vertex of r. The introduction of lids *normalizes* r. Namely, we have

Lemma 1 *The introduction of lids partitions r into a collection of monotone polygons.*

Proof: Let τ_1, τ_2, and τ_3 be the three trapezoids sharing a cusp c of polygon r, and, as before, let $\delta(\tau)$ denote the dual of trapezoid τ in $\delta(r)$.

Without loss of generality, we assume that $(\delta(\tau_1), \delta(\tau_2))$ is an arc of $\Delta(r)$. If also $(\delta(\tau_3), \delta(\tau_2))$ is an arc of $\Delta(r)$ (see Fig. 2 (a)), then $\delta(\tau_2)$ is the parent of both $\delta(\tau_1)$ and $\delta(\tau_3)$. Note that at most one of $\delta(\tau_1)$ and $\delta(\tau_3)$ can have weight greater than or eaqual to $\frac{1}{2}w(\delta(\tau_2))$, i.e., there is at most one heavy arc. Thus at least one of the two arcs $(\delta(\tau_1), \delta(\tau_2))$ and $(\delta(\tau_3), \delta(\tau_2))$ must be light, corresponding to a lid issuing from cusp c (Rule 1).

Suppose then that $(\delta(\tau_2), \delta(\tau_3))$ is an arc of $\Delta(r)$ (see Fig. 2 (b)). Then $(\delta(\tau_1), \delta(\tau_2))$ and $(\delta(\tau_2), \delta(\tau_3))$ have vertical components of opposite directions. If both arcs are heavy, we issue two horizontal lids from cusp c (Rule 2); otherwise there is at least one light arc and thus we issue at least one horizontal lid from c (Rule 1).

Hence, the process of normalization ensures that there is at least a horizontal lid issuing from each cusp of r, thereby achieving a decomposition of r into monotone regions. □

Lemma 2 *Each directed path of the dynamic tree $\Delta(r)$ contains at most $\log_2 m$ light arcs.*

Proof: Moving away from the root, each light arc traversed reduces the size of the current subtree by at least one half, since $w(child) < \frac{1}{2}w(parent)$. □

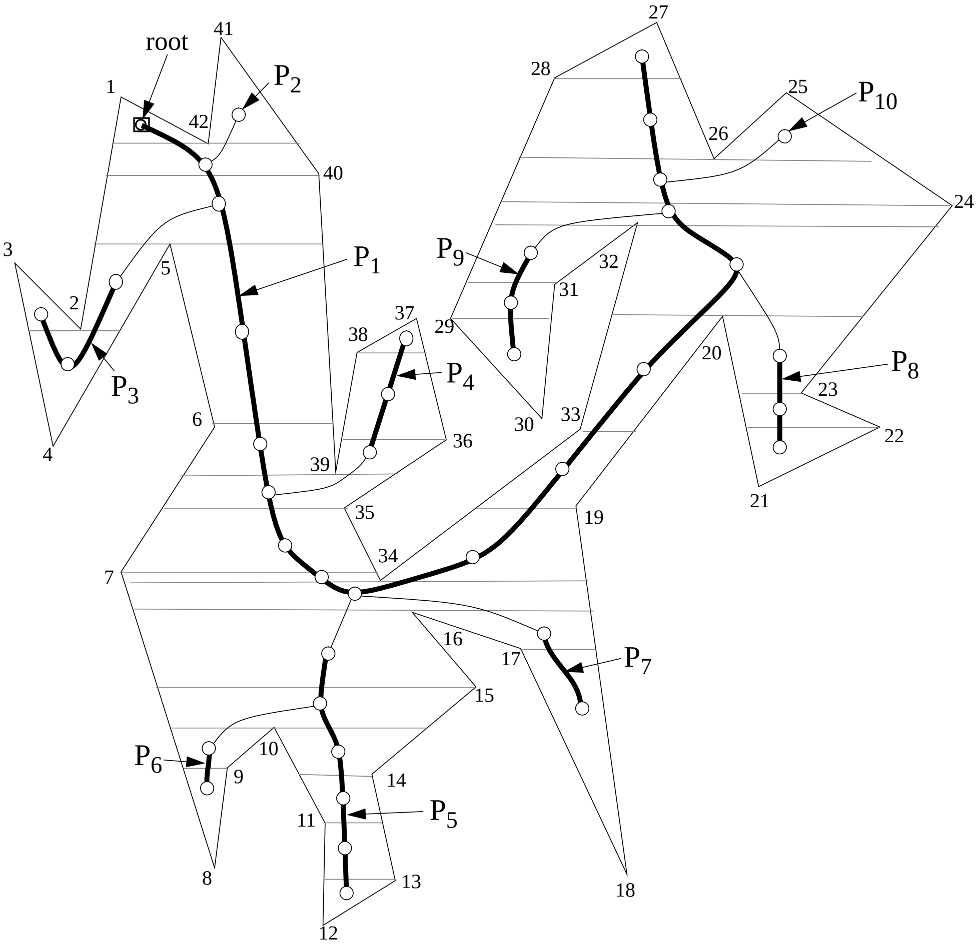

Figure 1: Example of a region r and its dynamic tree $\Delta(r)$ (P_1,...,P_{10} are solid paths).

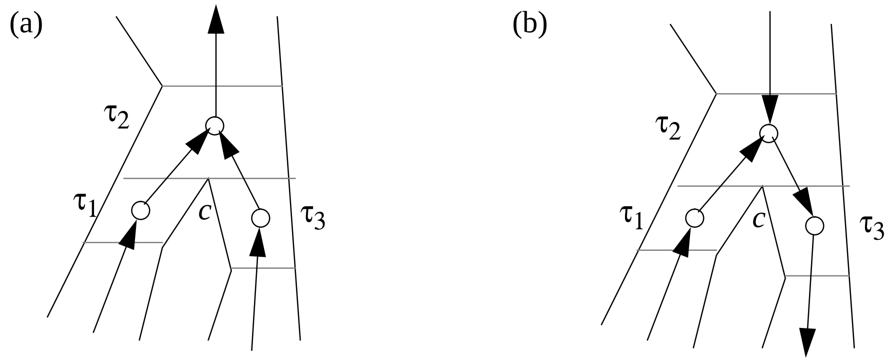

Figure 2: Proof of Lemma 1.

Corollary 1 *Altering the weight of a node of $\Delta(r)$ can cause at most $2\log_2 m$ arcs to change their classification.*

Proof: Suppose we alter the weight $w(\mu)$ of node μ. There is a unique path π from μ to the root of $\Delta(r)$ and the nodes on this path are the only ones whose weights are modified. Thus the arcs that may change classification are those incident on nodes of π: since there can only be one heavy outgoing arc per node, there are two sets of arcs (of cardinality $\leq \log_2 m$, by Lemma 2) which may exchange their classification. □

3.2 The Double-Thread Data Structure

It is intuitively clear that insertion or deletion of an edge may substantially modify the set of trapezoids, whereas it alters only slightly the structure of region boundaries. For this reason we adopt a data structure that represents a solid path of the dynamic tree $\Delta(r)$ by two "threads"; these two threads respectively correspond to two chains whose union is the boundary of the polygon associated with the solid path. The proposed structure is referred to as *double-thread data structure.*

Each arc α of $\Delta(r)$ intersects its dual splitter issuing from some vertex v of r. Therefore we associate α with v. Notice that each vertex v in $\mathcal{M}$ is associated twice: if v is a cusp of some region r, then the two splitters issuing from v both lie in r and thus cross two arcs of $\Delta(r)$; otherwise, v belongs to two regions r_1 and r_2 and the two splitters issuing from v cross respectively an arc of $\Delta(r_1)$ and an arc of $\Delta(r_2)$. Instead of maintaining the nodes of $\Delta(r)$, we may equally well maintain the arcs of $\Delta(r)$ using the vertices of r as their representatives, by associating each node of $\Delta(r)$ to the arc issuing from it. Specifically, each solid path P is represented by two binary trees $lthread(P)$ and $rthread(P)$, referred to as *path trees*, whose implementation is described below. Recall that each solid path is directed toward the root. Each vertex v associated with an arc on solid path P is classified as follows: walking along P toward the root, vertex v is classified *left* if it lies to the left of P, and *right* otherwise. Notice that if P is followed by a light arc α (every solid path except the one containing the root of $\Delta(r)$ has such an arc), then we also include α as an arc on solid path P in our representation.

The arcs of a solid path P can be partitioned into maximal monotone (with respect to the y-axis) subpaths $Q_1, Q_2, \cdots, Q_k$. Our path trees $lthread(P)$ and $rthread(P)$ are each actually implemented as a two-level balanced binary tree (i.e., the roots of lower-level trees are leaves of the higher-level tree). Referring to $lthread(P)$, in the lower level, we have a balanced binary tree $ltree(Q_i)$ for each Q_i, where the leaves of $ltree(Q_i)$ store the left vertices associated with the arcs on Q_i, in their path order. Path tree $rthread(P)$ is analogously organized, with $rtree(Q_i)$ storing the right vertices associated with the arcs on Q_i; the roots of $ltree(Q_i)$ and $rtree(Q_i)$ are bidirectionally linked. The implementation of $ltree(Q_i)$ and $rtree(Q_i)$ is deferred to Section 4; suffice it to say that we postulate logarithmic time operation. In the higher level, $lthread(P)$ (and analogously $rthread(P)$) has the roots of $ltree(Q_1)$, $ltree(Q_2)$, $\cdots$, $ltree(Q_k)$ as leaves in their path order. Between any two such leaves, as $ltree(Q_i)$ and $ltree(Q_{i+1})$, we insert an additional leaf H_i, called a *coupler*, representing a trapezoid, whose function will be soon apparent. We also establish a bidirectional link between the roots of $lthread(P)$ and $rthread(P)$. The double-thread data structure for our running example is shown in Fig. 3.

Any node on solid path P might be pointed to (through light arcs) by some other solid paths. Suppose that some solid path P' points to P through some arc α' associated with vertex v'. If v' is also associated with an arc of P (e.g., see paths P_2, P_3 and P_4 in Fig. 1 with $P = P_1$), we note that v' is a left or right vertex of P (thus stored as a lower-level leaf of $lthread(P)$ or of $rthread(P)$) according to whether P' appears to the left or right of P. We establish a pointer from each root of $lthread(P')$ and $rthread(P')$ to that lower-level leaf v' (see Fig. 3).

In the other case, where v' is not associated with any arc of P (e.g., see paths P_5 and P_7 in Fig. 1 with $P = P_1$), v' appears in neither $lthread(P)$ nor $rthread(P)$. Observe that in the original dynamic tree $\Delta(r)$, solid path P' points to some node μ of P where μ joins two monotone subpaths Q_i and Q_{i+1}; since μ does not belong to either Q_i or Q_{i+1}, we make it correspond to the coupler H_i. We let both roots of $lthread(P')$ and $rthread(P')$ point to the appropriate coupler leaf of $lthread(P)$ or $rthread(P)$ according to whether P' is to the left or right of P (see Fig. 3).

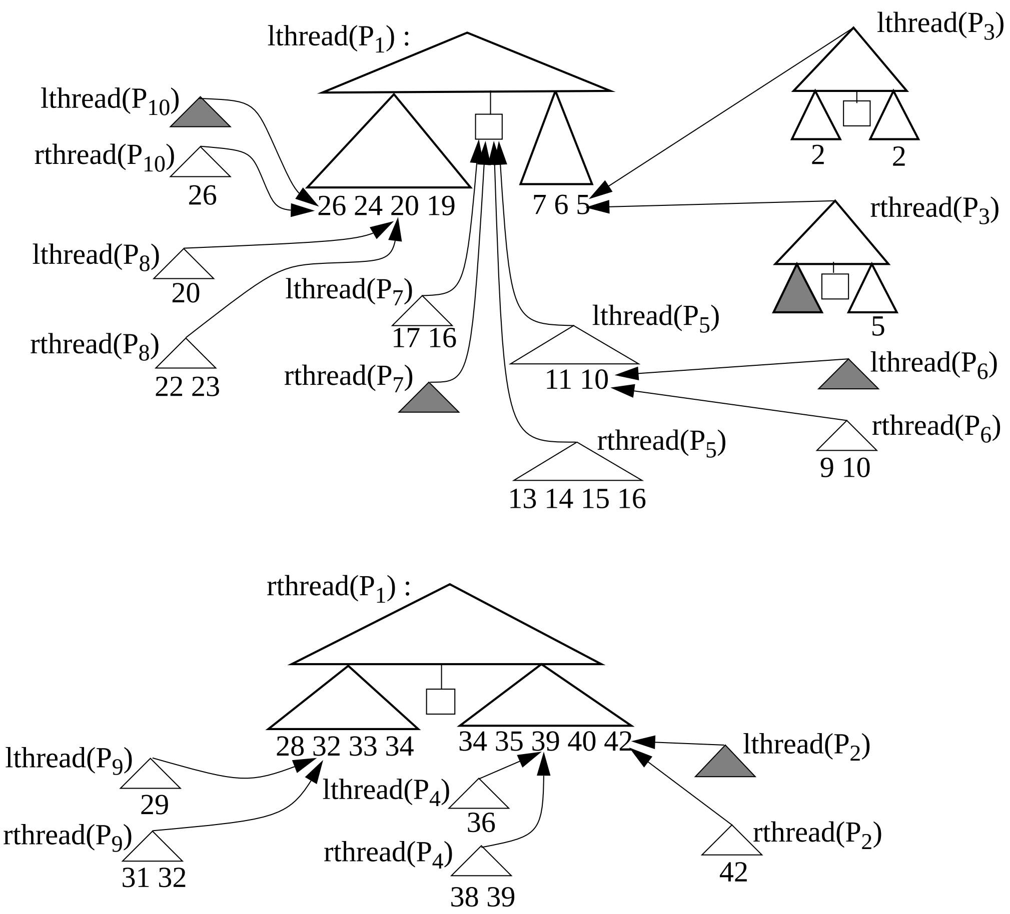

Figure 3: Double-thread data structures for representing the solid paths in Fig. 1 (the bidirectional pointers linking pairs of corresponding path trees and path subtrees are omitted).

We still need to define the weight for lower-level and higher-level leaves. In the lower level, the weight of a leaf v is the sum of the weights of all path trees pointing to v plus one (for vertex v itself); in the higher level, if a leaf λ is the root of some subtree T (either $ltree(Q_i)$ or $rtree(Q_i)$ for some Q_i), then the weight of λ is the sum of the weights of the leaves in T, otherwise leaf λ is some coupler H_i and its weight is the sum of the weights of the path trees pointing to λ (where the weight of a path tree is the sum of the weights of all its higher-level leaves).

If vertex v is, say, stored in $ltree(Q_i)$, we define the vertical interval $I_v \stackrel{\triangle}{=} [y', y'']$ where y' and y'' are respectively the ordinates of the two consecutive vertices of $rtree(Q_i)$ such that $y' < y(v) < y''$. Given v, I_v can be found as follows: since v is a leaf of $ltree(Q_i)$, we walk from v to the root of $ltree(Q_i)$, follow the pointer to the root of $rtree(Q_i)$, and then perform a binary search in $rtree(Q_i)$

to locate the interval I_v by y-coordinates. The same approach applies when v is stored in $rtree(Q_i)$. Clearly, the operation of finding I_v given v can be carried out in logarithmic time.

In summary, the weight $w(\mu)$ of node μ in the dynamic tree $\Delta(r)$ is obtained by summing the weights of leaves and couplers of both $lthread(P)$ and $rthread(P)$ up to and including vertex v and the corresponding interval I_v, where v is the vertex associated with the arc issuing from μ.

The preceding discussion establishes the following lemma.

Lemma 3 *The space complexity of the normalization structure for an n-vertex map is $O(n)$.*

3.3 Update Operations

We define the following update operations on a general map $\mathcal{M}$:

INSERTEDGE$(e, v_1, v_2, r; r_1, r_2)$: Insert edge $e = (v_1, v_2)$ into region r such that r is partitioned into two regions r_1 and r_2.

REMOVEEDGE$(e, v_1, v_2, r_1, r_2; r)$: Remove edge $e = (v_1, v_2)$ 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$(v_1, e; v_2)$: Insert edge $e = (v_1, v_2)$ and degree-one vertex v_2 inside some region r, where v_1 is a vertex of r.

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

With the above repertory, the following theorem is immediate.

Theorem 2 *An arbitrary general map $\mathcal{M}$ with n vertices can be assembled from the empty map, and disassembled to obtain the empty map, by a sequence of $O(n)$ operations drawn from the set $\{$ point-location query,* INSERTVERTEX, REMOVEVERTEX, INSERTEDGE, REMOVEEDGE, ATTACHVERTEX, DETACHVERTEX $\}$.

Finally, we show that ATTACHVERTEX and DETACHVERTEX can be simulated by a sequence of $O(1)$ operations taken among the first four of the repertory and point-location query. Referring for simplicity to ATTACHVERTEX$(v_1, e; v_2)$, we have the following emulation routine: perform a point-location query of v_2 to obtain the region r containing it, compute the two horizontal projection points v' and v'' of v_2 on the boundary of r, insert vertices v' and v'', insert edge (v', v''), insert vertex v_2 on (v', v''), insert edge e, remove edges (v', v_2) and (v_2, v'') and finally remove vertices v' and v''.

In the rest of this section, we describe how to implement the first four operations of the above repertory on the dynamic tree $\Delta(r)$ of an arbitrary region r (a simple polygon), represented by the double-thread data structure described above.

3.3.1 Primitive Dynamic Tree Operations

We begin by considering some elementary dynamic tree operations, referred to as *primitives*, in terms of which the operations of the above repertory can be immediately expressed. In the course of some updates, we may change a solid arc to dashed and *vice versa* and thus violate the weight invariant; so we need the capability to restore such weight invariant. Such actions are effected by the operation *expose* and *conceal* introduced in [29]. Operation $expose(\mu)$, for some node μ of $\Delta(r)$, transforms the unique path P from node μ to the root of $\Delta(r)$ into a solid path, by changing the dashed arcs in P to solid and the solid arcs incident to P to dashed. Since this transformation may violate the weight invariant of dynamic trees, we need the inverse operation $conceal(P)$ to remove the violation, by identifying all the light arcs in P and making them dashed, and also identifying all heavy arcs (if any) among the arcs incident to P and making them solid.

The primitive operation used in *expose* and *conceal* is $splice(P_1, P_2; P', P'')$, acting on two given paths P_1 and P_2 to produce two new paths P' and P'' (see Fig. 4). Originally solid path P_2 points to node μ of solid path P_1 via a dashed arc α. Denoting by α' the (solid) arc of P_1 terminating at μ (if any), *splice* exchanges the roles of α and α', i.e., it creates two new solid paths P' and P'' with P'' pointing to P' via dashed arc α' (again, P'' and α' might be empty).

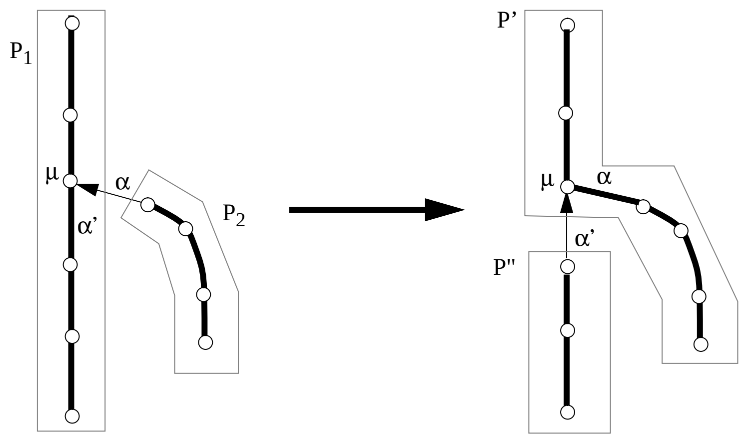

Figure 4: Example of $splice(P_1, P_2; P', P'')$.

Operation $splice(P_1, P_2; P', P'')$ essentially involves splitting and concatenating both threads of the paths concerned. Specifically, $lthread(P_1)$ is split into $lthread(P''')$ and $lthread(P'')$, and then $lthread(P_2)$ is concatenated with $lthread(P''')$ to form $lthread(P')$; analogously for *rthread*. To specify where the split occurs, assume without loss of generality that P_2 is to the left of P_1. In the case where $lthread(P_2)$ and $rthread(P_2)$ point to some coupler H_i in $lthread(P_1)$ (e.g., see path P_5 in Fig. 3), the split occurs at H_i, which is then deleted. In the other case, where $lthread(P_2)$ and $rthread(P_2)$ point to some leaf v of $lthread(P_1)$ (e.g., see path P_3 in Fig. 3), $lthread(P_1)$ is split at v, which is retained in one of the two resulting trees, and $rthread(P_1)$ is split at interval I_v. If combining P''' with P_2 locally violates monotonicity, then a new coupler separating two resulting monotone subpaths is also created. Clearly, operation *splice* can be executed in $O(\log m)$ time on the double-thread data structure.

Since each directed path in $\Delta(r)$ contains at most $\log_2 m$ light arcs by Lemma 2, *expose* uses at most $\log_2 m$ *splice* operations, and therefore is executed in $O(\log^2 m)$ time.

Given a solid path P, operation $conceal(P)$ identifies the arcs of P which have to be made dashed in order to comply with the weight invariant of dynamic trees. It can be carried out by finding the topmost (i.e., closest to the root of $\Delta(r)$) light arc α, splitting P at α, removing the subpath from the root up to and including α, and then repeating the process for the remaining solid path, until no light arc is found. So the main issue is how to find the topmost light arc.

Before describing its adaptation to the double-thread data structure, we briefly review for the reader's convenience the standard single-path implementation of operation *conceal* as proposed by

Sleator and Tarjan [29]. Let solid path P be stored in path tree $T(P)$ with the root of $\Delta(r)$ on the right. Each leaf μ stores $weight(\mu)$, defined as the sum of the weights of all the other path trees pointing to μ (if any) plus one (to account for μ itself), where the weight of a path tree is the sum of the weights of all its leaves. For each internal node v, $weight(v)$ is defined as the sum of the weights of all leaves in the subtree T_v rooted at v. Denoting by λ the rightmost leaf in T_v, and by x the leaf adjacent to λ on the left, variable $lefttilt(v)$ is defined as the sum σ of the weights of the leaves in $T(P)$ from the leftmost one up to and including x, minus $weight(\lambda)$. We recall that arc (x, λ) is light if and only if $w(x) < \frac{1}{2}w(\lambda)$, i.e., $\sigma < \frac{1}{2}(\sigma + weight(\lambda))$, namely $lefttilt(v) < 0$.

Moreover, define $leftmin(v) \overset{\triangle}{=} min\{lefttilt(u) : u \text{ is an internal node of } T_v\}$. It follows that if $leftmin(v) \geq 0$, then there is no light arc between any two adjacent leaves of T_v. Also, variable $netleft(v)$ is defined as $leftmin(v)$ if v is the root of $T(P)$ and $leftmin(v) - leftmin(parent(v))$ otherwise. In summary, each internal node v of $T(P)$ stores three values: $weight(v)$, $netleft(v)$, and $netright(v)$; correspondingly, variables $netright(v)$, $rightmin(v)$, and $righttilt(v)$ are defined symmetrically in a straightforward manner by summing the weights from right to left for the purpose of reversing the path direction.

To find the topmost light arc in P, we traverse a path from the root of $T(P)$ with the following advancing mechanism: Assume inductively that, for the current node v, parameter $leftmin(v)$ (< 0) is known. Let v' and v'' be the left and right children of v, respectively, and x be the leftmost leaf of $T_{v''}$. From the definition

$$netleft(v'') = leftmin(v'') - leftmin(v)$$

we obtain $leftmin(v'')$. If $leftmin(v'') < 0$, then we proceed to v''. Otherwise, we compare $weight(v')$ and $weight(x)$. If $weight(v') < weight(x)$, then the arc leading to x is the sought light arc; else, we compute $leftmin(v') = leftmin(v) + netleft(v')$ (which is necessarily < 0) and proceed to v' (this establishs the inductive step). By this process akin to binary search, the topmost light arc can be found in $O(\log m)$ time, where m is the size of $T(P)$. Recall that by Lemma 2, there are at most $\log_2 m$ light arcs in $T(P)$.

We are now ready to consider the implementation of *conceal* for the double-thread data structure. We treat $lthread(P)$ and $rthread(P)$ independently as two path trees of the type described above. By the method just illustrated, we identify at most $\log_2 m$ light arcs from each of $lthread(P)$ and $rthread(P)$. Note that a light arc (x, λ) in P assures the existence of a light arc (x', λ) in either $lthread(P)$ or $rthread(P)$. Indeed, in $T(P)$ the sum σ of the weights up to and including x satisfies $\sigma < weight(\lambda)$; but in the appropriate tree (i.e., either $lthread(P)$ or $rthread(P)$), the sum σ' of the weights up to and including x' is only a fraction of σ (σ is contributed by *both* $lthread(P)$ and $rthread(P)$), so that $\sigma' \leq \sigma < weight(\lambda)$, and (x', λ) is light. Hence $2\log_2 m$ light arcs from $lthread(P)$ and $rthread(P)$ give all possible candidates for light arcs in P. For each such candidate (x', λ), we perform a binary search in the paired path tree to locate the point just before λ, at the same time accumulate the total weight σ'' up to and including this point in that tree, then compute σ by adding σ'' to σ', and check if $\sigma < weight(\lambda)$. Therefore, we find $2\log_2 m$ candidates, perform $2\log_2 m$ binary searches for checking, identify at most $\log_2 m$ light arcs in P and then split accordingly—each of these operations within $O(\log m)$ time. This leads to the following lemma.

Lemma 4 *The update of the double-thread data structure as required by the operation conceal can be performed in $O(\log^2 m)$ time.*

The operation of moving the root of $\Delta(r)$ to any arbitrary node is briefly referred to as *evert*. We implement *evert* by adding a "direction" bit to each node of the path trees, so that when we reverse the direction of a solid path P, each of the direction bits of the roots of $lthread(P)$ and *rthread*(P) is complemented, indicating that the meanings of left and right subtrees and also of *lthread*(P) and *rthread*(P) are interchanged. Given the direction bits and operations *expose* and *conceal* described above, we can perform *evert* in the double-thread data structure in $O(\log^2 m)$ time.

In the following, if μ is a node of $\Delta(r)$ and a an incoming arc of μ, the notations *expose*(a) and *expose*(μ) are equivalent, and similarly for *evert*.

Lemma 5 *Given splitters s_1 and s_2 of region r with m vertices,* SLEEVE(s_1, s_2) *and the corresponding solid path between $\delta(s_1)$ and $\delta(s_2)$ can be constructed in $O(\log^2 m)$ time, by means of $O(\log^2 m)$ elementary splits/joins of path trees.*

Proof: We obtain a solid path between $\delta(s_1)$ and $\delta(s_2)$ by *evert*$(\delta(s_1))$ and *expose*$(\delta(s_2))$. Each of operations *evert* and *expose* uses $O(\log^2 m)$ elementary splits/joins and takes $O(\log^2 m)$ time. □

The double-thread structure adds two new primitive operations to the original repertory of dynamic trees. Operation *part*$(P, e; P_1, P_2)$ on a solid monotone path P separates *lthread*(P) and *rthread*(P), and creates two new solid paths P_1 and P_2 by adjoining *lthread*(P) and *rthread*(P) to a new edge e. Namely, *lthread*(P_1) = *lthread*(P), *rthread*$(P_1) = e$, *lthread*$(P_2) = e$, and *rthread*(P_2) = *rthread*(P). The operation *pair*$(P_1, P_2; P, e)$ is the inverse operation of *part*$(P, e; P_1, P_2)$ and is implemented similarly.

Lemma 6 *Operations part and pair have time complexity $O(1)$.*

As we shall see in the next section, operations *part* and *pair* are crucial in the efficient execution of INSERTEDGE and REMOVEEDGE.

3.3.2 Insertion and Deletion of Edges and Vertices

Operation INSERTEDGE$(e, v_1, v_2, r; r_1, r_2)$ is carried out as follows:

1. For $i = 1, 2$, if v_i is an extreme vertex of r, let $s_i = v_i$, else let s_i be a splitter of r induced by v_i (if v_i is a cusp of r there are two such splitters, and we choose either one).
2. Construct SLEEVE(s_1, s_2), and the corresponding solid path P, by applying Lemma 5 (with minor modifications if v_1 and/or v_2 is an extreme vertex of r).
3. Insert edge $e = (v_1, v_2)$, which essentially corresponds to *part*$(P, e; P_1, P_2)$. We remove v_1 and v_2 from *lthread*(P) and *rthread*(P) (if any), make *lthread*(P) to be *lthread*(P_1) and *rthread*(P) to be *rthread*(P_2), and create *rthread*(P_1) and *lthread*(P_2) each of which consists of v_1 and v_2 (either vertex may be missing if it is an extreme vertex). In the special case where v_1 is a cusp, prior to the update we check if some path P' to the left of v_1 points to leaf v_1 of *lthread*(P) or some path P'' to the right of v_1 points to leaf v_1 of *rthread*(P): in the former case we splice P' with P_1 and in the latter case we splice P'' with P_2. The other special case where v_2 is a cusp is handled similarly.
4. Make the roots of $\Delta(r_1)$ and $\Delta(r_2)$ be two representatives of v_2, then perform *conceal*(P_1) and *conceal*(P_2).

We analyze the time complexity of the above operation. Step 3 (splitting P into P_1 and P_2) takes $O(\log m)$ time; handling of the special cases takes $O(1)$ *splice* operations. All the other steps globally involve a fixed number of *evert*, *expose* and *conceal* operations, so that total time complexity for updating the double-thread data structure is $O(\log^2 m)$.

Operation REMOVEEDGE$(e, v_1, v_2, r_1, r_2; r)$ is the inverse operation of INSERTEDGE. We first evert v_2 and then expose v_1 in both $\Delta(r_1)$ and $\Delta(r_2)$, pair the two solid paths into one, and conceal it. This can also be done in $O(\log^2 m)$ time.

Operation INSERTVERTEX$(v, e; e_1, e_2)$ is performed as follows. We first perform a point location query of v to find e (refer to Section 5), insert v with $weight(v) = 1$ into $lthread(P)$ and $rthread(P')$ for some solid paths P and P' of different regions r and r', respectively, where both $lthread(P)$ and $rthread(P')$ contain two endpoints v_1 and v_2 of e. Assuming, without loss of generality, that in $\Delta(r_1)$ v_1 is further from the root than v_2 whereas the opposite occurs in $\Delta(r_2)$, we expose v_1 in $\Delta(r_1)$ and expose v_2 in $\Delta(r_2)$. Finally we perform operation *conceal* on these two paths. It is easy to see that these operations are executed in $O(\log^2 m)$ time. Operation REMOVEVERTEX is the inverse operation of INSERTVERTEX, and can be completed within the same time bound.

4 Shortest Path Queries

In this section we illustrate how the normalization data structure presented in Section 3 and designed to efficiently maintain monotonicity in $\mathcal{M}$ can be modified to answer the following important queries in a dynamic regimen:

PATHLENGTH(q_1, q_2, r): Return the length of a shortest path inside region r between query points q_1 and q_2.

PATH(q_1, q_2, r): Return the shortest path inside region r between query points q_1 and q_2.

We point that due to the specific implementation of point location (see Section 5), PATHLENGTH and PATH return an error message if at least one of q_1 and q_2 is not in region r. Specification of region r may be omitted when the region to which q_1 and q_2 belong is unique and can be identified by point location; however, specification is appropriate to cover the case when r is not unique. We show that the normalization data structure can be modified to support such queries in worst-case time $O(\log^3 n)$ and $O(\log^3 n + k)$, respectively, where k is the number of segments in the shortest path reported by PATH.

We first review the notion of "hourglass", which is central to our current problem. We adopt the terminology originally proposed by Guibas and Hershberger [16].

Consider two nonintersecting diagonals $s_1 = (a_1, b_1)$ and $s_2 = (a_2, b_2)$ of r, where the endpoints have been named so that the clockwise cyclic sequence of points in the boundary of r includes the subsequence (a_1, a_2, b_2, b_1). The *hourglass* of s_1 and s_2, denoted HOURGLASS(s_1, s_2), is the subregion of r formed by the union of all the paths PATH(q_1, q_2) with $q_1 \in s_1$ and $q_2 \in s_2$ (see Fig. 5.a). It is known that the boundary of HOURGLASS(s_1, s_2) is the concatenation of s_1, PATH(a_1, a_2), s_2, and PATH(b_2, b_1). Let α be the subchain of r clockwise from a_1 and a_2, and define β similarly for b_2 and b_1. The hourglass has one of the following special structures (as analyzed by [16]):

Open hourglass: If the convex hulls of α and β do not intersect, then PATH(a_1, a_2) is the convex hull of the subchain of α counterclockwise from a_1 to a_2, and similarly for PATH(b_1, b_2).

Closed hourglass: If the convex hulls of α and β intersect, then there exist vertices p_1 and p_2 of $\alpha \cup \beta$ such that PATH$(a_1, a_2) \cap$ PATH$(b_1, b_2) =$ PATH(p_1, p_2). Without loss of generality, assume that p_1 is in α. Then PATH(a_1, p_1) is the convex hull of the subchain of α from a_1 to p_1, while PATH(b_1, p_1) is the union of segment (p_1, q_1) and of the convex hull of the subchain of β from b_1 to q_1, where q_1 is the vertex of β closer to b_1 on the two tangents from p_1 to β. Similar arguments apply to p_2. The union of PATH(a_i, p_i) and PATH(b_i, p_i) ($i = 1$ or 2) is called a *funnel*. Vertices p_1 and p_2 are called the *apices* of the hourglass, and the path between them the *string* of the hourglass (see Fig. 6.a).

Assume that an hourglass is represented by storing the length of the string (if it exists), and the (two to four) convex chains forming the rest of its boundary. Given HOURGLASS(s_1, s_2) it is possible to compute PATHLENGTH(q_1, q_2, r) in $O(\log n)$ time for any two points $q_1 \in s_1$ and $q_2 \in s_2$ by means of $O(1)$ common tangent computations. Also, given HOURGLASS(s_1, s_2) and HOURGLASS(s_2, s_3) in r, with s_1 and s_3 on opposite sides of s_2, it is possible to compute HOURGLASS(s_1, s_3) in time $O(\log n)$ by means of $O(1)$ common tangent computations and $O(1)$ split and join operations on the chains forming the two hourglasses.

We now consider the modifications of the normalization data structure that enable the support of the given path queries. As we shall see, only two items are needed, i.e.:

(i) The choice of an appropriate implementation of the path trees *ltree* and *rtree*;

(ii) The appending of secondary data structures (collectively referred to as "hull structure") to the nodes of these path trees. The *hull structure* stores at the nodes of the path trees the hourglasses of the corresponding sleeves; it establishes an implicit correspondence between the two chains of a monotone sleeve, allowing both efficient access to the hourglass of the sleeve, and fast pairing or parting of the two chains when an edge is inserted or deleted into or from the sleeve.

We first describe the adopted representation of polygonal chains. A concatenable queue, called *chain tree*, will be used to represent a polygonal chain γ. The chain tree T for γ is a balanced tree. We assume that T has inorder thread pointers, which can be maintained without penalizing the asymptotic time complexity in any of the known balanced tree schemes. Each node μ of T corresponds to a subchain γ_μ of γ and stores the endpoints of γ_μ, the common point of the subchains of the children of γ_μ, and the length of γ_μ. It should be clear that this information can be updated in $O(1)$ time per elementary join or split, 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 [24].

We now give the details of our representation of hourglasses. An open hourglass is represented by storing its two convex chains into chain trees. A closed hourglass is represented by storing into separate substructures the four convex chains forming the funnels, and the string between the apices. The convex chains of the funnels and the string are each stored into a chain tree.

Without loss of generality we assume that no two vertices of $\mathcal{M}$ have the same y-coordinate and that the degree of each vertex is at most three. This is not restrictive since we can expand a vertex v with degree $d > 3$ into a chain of degree-3 vertices connected by edges of infinitesimal length. Since the sum of the degrees of all vertices of $\mathcal{M}$ is $O(n)$, the total number of vertices after the expansion is still $O(n)$. Every update operation in the original map $\mathcal{M}$ can be simulated with $O(1)$ operations in the modified map with bounded-degree vertices.

We consider the ordered sequence Y of the y-coordinates of the vertices of $\mathcal{M}$, and establish a one-to-one correspondence between Y and the leaves of a BB$[\alpha]$-tree $\mathcal{Y}$, called *Y-tree*. Tree $\mathcal{Y}$ determines a hierarchical partition of the plane into horizontal strips according to the well-known segment-tree scheme. Each node of $\mathcal{Y}$ corresponds to a *canonical interval* of y-coordinates. A vertical interval (y', y'') with $y', y'' \in Y$ is uniquely partitioned into $O(\log n)$ canonical intervals, called the *fragments* of (y', y''), and their associated nodes in $\mathcal{Y}$ are called the *allocation nodes* of (y', y''). We extend this terminology to any geometric entity that is uniquely associated with a vertical interval, such as an edge, a monotone chain, or a monotone sleeve.

We now introduce the useful notion of "pruned tree". A *pruned tree* of a rooted tree T is a tree S that can be obtained from T by first taking the subtree rooted at a node of T, and then removing from it the subtrees rooted at a selected subset of its nodes. Pruned trees of a balanced tree T support the full repertory of concatenable queue operations. Each operation takes $O(\log n)$ time and is performed by means of $O(\log n)$ elementary joins and splits between pruned trees whose roots are associated with sibling nodes in T. A sequence I of k consecutive intervals with endpoints in Y will be stored in a pruned tree of $\mathcal{Y}$, whose leaves are the allocation nodes of the intervals of I, and whose internal nodes are the ancestors of such leaves. It is easy to verify that the pruned tree associated with I has $O(k \log n)$ nodes and $O(\log n)$ height.

Now, we show how to modify the normalization data structure so that hourglasses can be dynamically maintained (see Fig. 5). We denote with Q a maximal monotone subpath of a solid path P, as used in Section 3.2, and specify the implementation of $ltree(Q)$ and $rtree(Q)$. We use pruned trees augmented with chain-trees as secondary structures. Our scheme uses ideas from [22] and [16].

- Trees *ltree*(Q) and *rtree*(Q) are implemented by means of pruned trees with respect to $\mathcal{Y}$.
- Let μ be a node of *ltree*(Q) (nodes of *rtree*(Q) are handled identically) and ν the parent of μ. Node μ has a pointer to the corresponding node y of $\mathcal{Y}$. Also, if μ is not a leaf, then we establish a back pointer from y to μ. Consider the subpath Q' of Q associated with the subtree of *ltree*(Q) rooted at μ. We denote with $\textsc{sleeve}(\mu)$ the sleeve of Q', and with $\textsc{chain}(\mu)$ the "left chain" of $\textsc{sleeve}(\mu)$, i.e., the chain formed by the edge-fragments stored at the leaves of the subtree of *ltree*(Q) rooted at μ.
  We distinguish several subcases:
  - If μ is a leaf of $ltree(Q)$, then μ stores the corresponding edge fragment.
  - If $\textsc{hourglass}(\mu)$ is open and $\textsc{hourglass}(\nu)$ is closed, then μ stores in a secondary data structure the right convex hull of $\textsc{chain}(\mu)$.
  - If both $\textsc{hourglass}(\mu)$ and $\textsc{hourglass}(\nu)$ are open, then μ stores in a secondary data structure only the endpoints of $\textsc{chain}(\mu)$ and the portion of the right hull of $\textsc{chain}(\mu)$ that is not stored at an ancestor of μ.
  - If $\textsc{hourglass}(\mu)$ is closed and μ is the root of *ltree*(Q), then μ stores in secondary data structure the (up to five) components of the $\textsc{hourglass}(s_1, s_2)$, where s_1 and s_2 are the splitters that delimit $\textsc{sleeve}(\mu)$.
  - If $\textsc{hourglass}(\mu)$ is closed and μ is not the root, then μ stores the apices and the length of the string of $\textsc{hourglass}(s_1, s_2)$, plus the subchains of the funnels of $\textsc{hourglass}(s_1, s_2)$ that are not stored at the ancestors of μ, where s_1 and s_2 are the splitters that delimit $\textsc{sleeve}(\mu)$.
- The trees *lthread*(P) and *rthread*(P) are isomorphic. Also, an internal node μ in the higher-level of *lthread*(P) stores the length and the endpoints of the string of $\textsc{hourglass}(s_1, s_2)$, where s_1 and s_2 are the splitters that delimit $\textsc{sleeve}(\mu)$. The corresponding node of *rthread*(P) stores exactly the same information.

Lemma 7 *The space requirement of the hull structure is* $O(n \log n)$.

Proof: We only need to determine the space used by the secondary structures (the chain-trees) that augment the path trees. Consider the set S of all segments s such that s is either an edge-fragment or the tangent segment in the hourglass of a path-tree node. We claim that the size of S is $O(n \log n)$. Indeed, consider the hourglasses H, H', and H'' of the sleeves of a node μ and its children μ' and μ''. Because of the way H is obtained from H' and H'', H has only $O(1)$ edges that are not already in H' and H'', and the claim follows by induction. Also, each segment of S is stored at most six times in the data structure, since it can have representatives in an allocation node (for an edge fragment), in the the highest open hourglass, and in the highest monotone hourglass, and there are always two such nodes for each segment (recall that edges have two "sides", and the nodes of the *lthread* and *rthread* trees have duplicate information). We conclude that the secondary

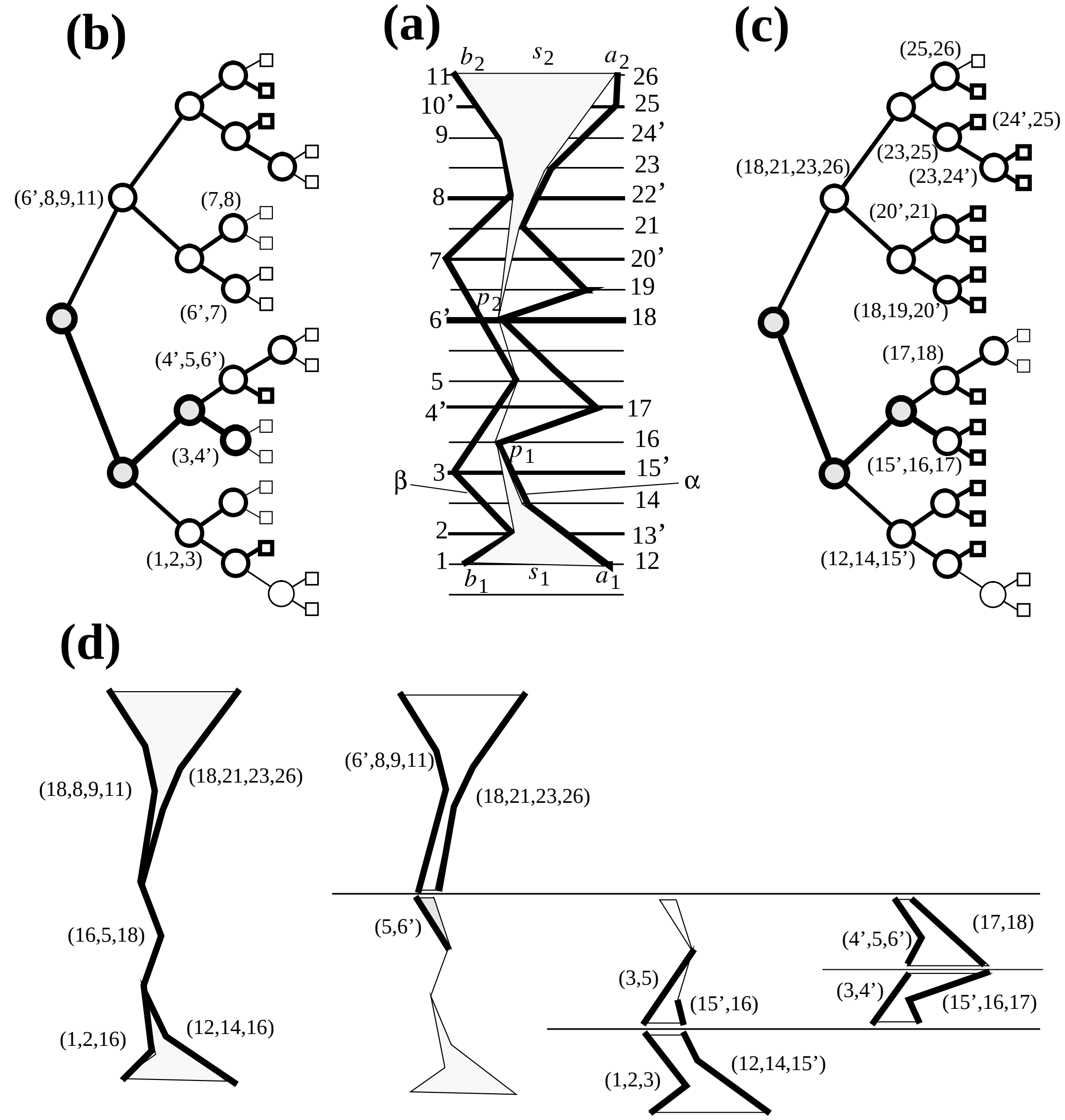

Figure 5: Example of representation of hourglasses in the nodes of $ltree(Q)$ and $rtree(Q)$ of a monotone path Q. **(a)** The sleeve of Q (directed from left to right): the parallel lines drawn on it represent set Y; the points on the sleeve with labels of the type i' delimit fragments of the same edge; the hourglass between the extreme splitters of the sleeve is shown grey-filled. **(b)** Pruned-tree $ltree(Q)$: the nodes of $ltree(Q)$ are those drawn with thick lines, while the nodes drawn with thin lines denote the subtrees of $\mathcal{Y}$ pruned away to construct $ltree(Q)$; the grey-filled nodes are associated with closed sleeves, and the white-filled nodes are associated with open sleeves. Next to each white-filled node μ we show the subchain of HOURGLASS(μ) stored at μ. **(c)** Pruned-tree $rtree(Q)$ (similar comments as in (b) apply). **(d)** Hourglasses of the grey-filled nodes and of their children. The subchains stored at each node are labeled and shown with thick lines.

structures are a collection of balanced trees with a total of $O(n \log n)$ nodes, and hence use total space $O(n \log n)$. □

Query operations PATHLENGTH(q_1, q_2, r) and PATH(q_1, q_2, r) are performed as follows:

1. Find the trapezoids τ_1 and τ_2 of the trapezoidal decomposition of r containing q_1 and q_2, respectively, using the point-location machinery described in Section 5. Let s_1 and s_2 be the splitters on the boundary of τ_1 and τ_2, such that q_1 and q_2 are on opposite sides of SLEEVE(s_1, s_2).
2. Create the solid path P for SLEEVE(s_1, s_2) (P is the path between edges $\delta(s_1)$ and $\delta(s_2)$ of $\delta(r)$), by means of *evert*$(\delta(s_1))$ and *expose*$(\delta(s_2))$. The secondary structure stored at the root of *lthread*(P) (or *rthread*(P)) yields a representation of HOURGLASS(s_1, s_2).

Given the representation of HOURGLASS(s_1, s_2) and after the computation in time $O(\log n)$ of the tangents from q_1 and q_2 to the appropriate funnels, we can answer PATHLENGTH(q_1, q_2, r) and PATH(q_1, q_2, r) in time $O(1)$ and $O(k)$, respectively (where k is the number of edges of the shortest path reported). Finally, we conceal the path exposed in step 2 to satisfy the weight invariant.

Regarding updates, we have (see the example in Fig. 6):

Lemma 8 *An elementary split or join of two solid path trees in the normalization structure augmented with the hull structure takes time $O(\log n)$.*

Proof: After an elementary split or join of two solid path trees, we need to update only the secondary data structures of their roots. Since such data structures represent the hourglasses of the corresponding sleeves, they can be updated in $O(\log n)$ time (see Fig. 6). Note that for a nonmonotone solid path the updates are limited to its leftmost or rightmost monotone subpath. For this reason it is sufficient to store only the the length of the string in the nodes of the *lthread* and *rthread* trees. □

As a consequence, splitting a solid path or joining two solid paths takes time $O(\log^2 n)$. Note that parting or pairing *ltree*(Q) and *rtree*(Q) of a monotone path Q (because of an edge insertion or deletion in the corresponding sleeve) takes $O(1)$ time.

Lemma 9 *Queries* PATHLENGTH(q_1, q_2, r) *and* PATH(q_1, q_2, r) *are performed in time $O(\log^3 n)$ and $O(\log^3 n + k)$, respectively, where k is the number of edges of the shortest path reported.*

Now, we discuss how operation INSERTVERTEX$(v, e; e_1, e_2)$ affects the new data structure. First, we insert $y(v)$ into $\mathcal{Y}$. Let μ be one of the two nodes in the hull structure that stores the fragment of edge e where v is inserted, and let y be the corresponding node of $\mathcal{Y}$. Before the insertion of v, there is a pointer from μ to y but no back pointer from y to μ, since μ is a leaf of pruned tree. After the insertion of v, we add two leaf children to μ and establish a pointer from y to μ.

The insertion of y into $\mathcal{Y}$ may cause rebalancing operations in $\mathcal{Y}$, which are carried out by means of rotations. A rotation exchanges the order in which horizontal cuts are performed. Namely, a rotation between a node y' and its child y'' implies that horizontal cuts at y'' now take priority with respect to horizontal cuts at y'. It is easy to see that the rotation only affects the subtrees of the path trees rooted at the nodes pointed by y'. We rebuild such subtrees from scratch, which can be done in time proportional to their size.

Lemma 10 *Let y be a node of $\mathcal{Y}$ whose subtree has ℓ leaves. The total size of the subtrees in the hull structure whose roots are pointed to by node y is $O(\ell \log \ell)$.*

Proof: The subtree of $\mathcal{Y}$ rooted at y has at most $2\ell - 1$ nodes. Thus there are $O(\ell)$ vertices inside the canonical vertical interval I of node y. The leaves of the subtree rooted at a node pointed by y store the edge fragments that are inside I but do not span I. Hence, the edges contributing to such

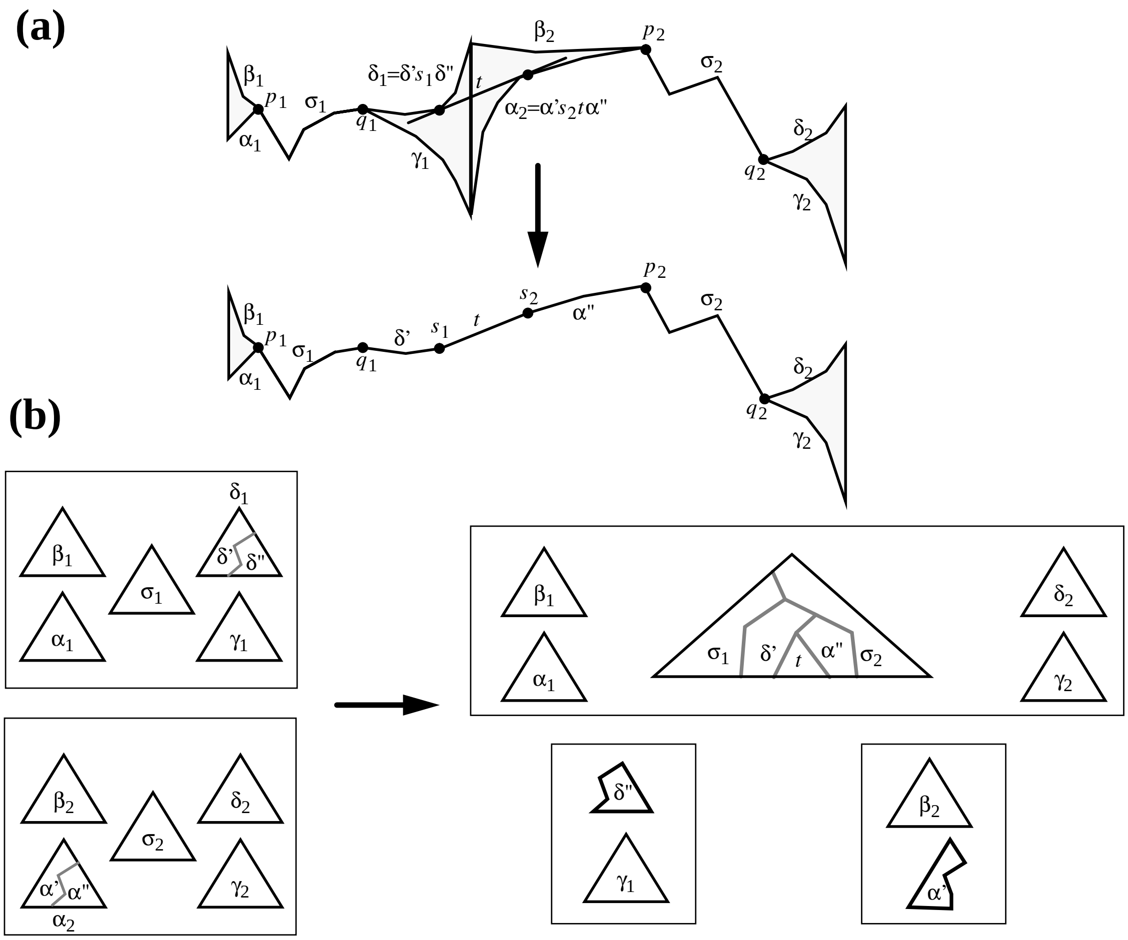

Figure 6: Example of update of the secondary structures in an elementary join of two solid paths. **(a)** Geometric construction of the hourglass. **(b)** Construction of the representation of the root hourglass by means of split and join operations on the chain-trees in the representation of the hourglasses of the children nodes.

fragments must be incident on some vertex inside I. Since each vertex has bounded degree, there are $O(\ell)$ such edges. Also, since the subtree of $\mathcal{Y}$ has height $O(\log \ell)$, each such edge has $O(\log \ell)$ fragments inside I. We conclude that the total number of leaves in the subtrees rooted at node pointed by y is $O(\ell \log \ell)$, and hence their total size is also $O(\ell \log \ell)$. □

By the properties of BB[α]-trees, we derive the following lemma.

Lemma 11 *The amortized rebalancing time of the Y-tree $\mathcal{Y}$ in a sequence of update operations is $O(\log^2 n)$.*

We conclude:

Theorem 3 *Shortest path queries* PATHLENGTH(q_1, q_2, r) *and* PATH(q_1, q_2, r) *in an n-vertex planar map can be performed in worst-case time $O(\log^3 n)$ and $O(\log^3 n + k)$, respectively (where k is the number of edges of the shortest path reported), using a fully dynamic data structure that uses space $O(n \log n)$ and supports updates of the map in time $O(\log^3 n)$ (worst-case for edge updates, and amortized for vertex updates).*

Remark. In a concrete situation where vertices are a priori restricted to a fixed set of ordinates, tree $\mathcal{Y}$ is static; if we then implement the path-trees *ltree* and *rtree* by means of *contracted binary trees* [27] of depth $< \log |Y|$ (whose maintenance requires no rotation), then the update times become $O(\log^2 n \log |Y|)$, in the worst case.

The following are two additional types of queries that can be supported by the described data structure without any modification:

TRAILLENGTH$(q_1, q_2 | e_1, ..., e_\ell)$: allowing edges e_1,...,e_ℓ to be deleted, are points q_1 and q_2 reachable to each other? if so, then return the length of the shortest path.

TRAIL$(q_1, q_2 | e_1, ..., e_\ell)$: allowing edges e_1,...,e_ℓ to be deleted, are points q_1 and q_2 reachable to each other? if so, then return the shortest path.

An immediate application is that viewing the edges of the map as walls, we are allowed to put doors on edges e_1,...,e_ℓ; can a point-like robot at position q_1 reach position q_2? If so, then report the shortest path or its length.

Clearly, by using REMOVEEDGE, point-location query (see Section 5), PATHLENGTH or PATH, and INSERTEDGE operations, queries TRAILLENGTH and TRAIL can be answered in worst-case time $O(\ell \log^3 n)$ and $O(\ell \log^3 n + k)$, respectively, where k is the number of edges of the shortest path reported.

5 Point Location

In this section we consider the problem of answering point-location queries:

LOCATE(q): Find the region containing query point q. If q is on an vertex or edge, then return that vertex or edge.

Our dynamic point-location data structure is inspired by the static trapezoid method [23] and its dynamic version for monotone maps [8]. It uses the normalization and hull structures as the underpinning of update operations. Queries are instead performed in a location structure, a binary tree called *trapezoid tree*.

The trapezoid tree defines a binary partition of the plane obtained by means of vertical and horizontal cuts. It differs in many substantial aspects from the trapezoid trees used in [8, 23], the most striking difference being that it is not balanced.

The trapezoid tree $\mathcal{T}$ for map $\mathcal{M}$ is based on the *Y-tree* $\mathcal{Y}$ (see Section 4) and on the normalization of $\mathcal{M}$ as reflected by the normalization structure (see Section 3). We view unnormalized map $\mathcal{M}$ as a trapezoid with its sides at infinity. If a trapezoid τ contains more than a single edge fragment in its interior, we recursively decompose it into trapezoids whose vertical spans are canonical vertical intervals, according to the following rules (see Fig. 7):

Vertical cut: If τ is a coupler or is vertically spanned by a monotone subpath Q and the hourglass H of SLEEVE(Q) is open, then we decompose τ by "cutting" it along one of the supporting tangents t of H.

Horizontal cut: If no vertical cut is possible, then we decompose τ by cutting it along the horizontal line at the y-coordinate associated with the (unique) allocation node of τ in $\mathcal{Y}$.

Note that a vertical cut always takes priority over a horizontal cut. However, if more vertical cuts are possible, their order is arbitrary. We represent the above decomposition of $\mathcal{M}$ by means of a binary tree $\mathcal{T}$ (see Fig. 7). Each node of $\mathcal{T}$ is associated with a trapezoid of the decomposition and the partitioning object (a tangent or a horizontal line) of that trapezoid, and stores the representation of such object. Nodes of $\mathcal{T}$ are classified into three categories: a $\bigcirc$-*node* is associated with a vertical cut, a ∇-*node* is associated with a horizontal cut, and a $\square$-*node* is associated with a terminal trapezoid of the decomposition and its edge fragment.

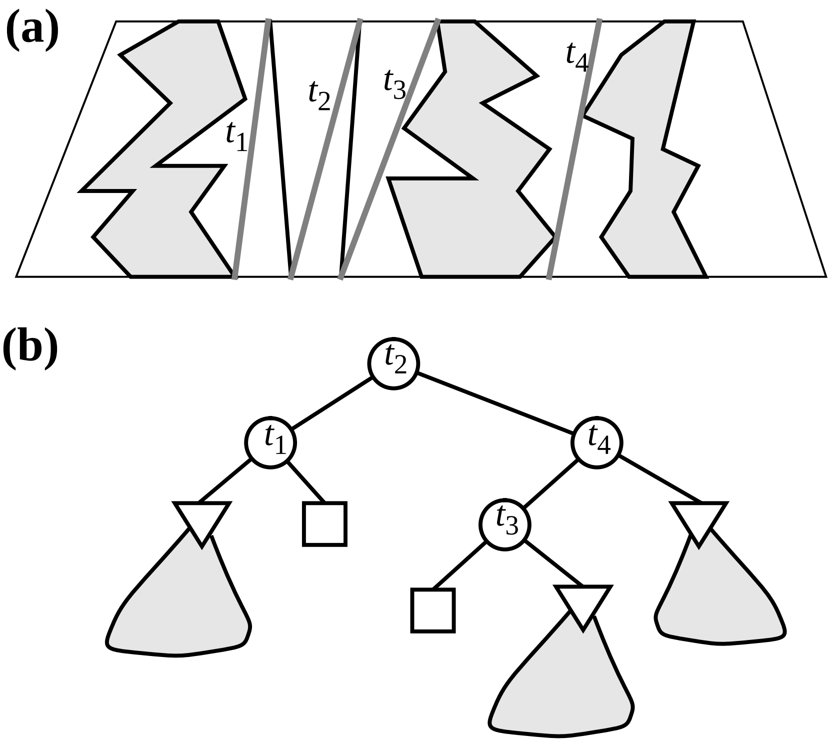

Figure 7: Schematic example of the construction of the trapezoid tree. **(a)** Decomposition of a trapezoid by vertical cuts. **(b)** Trapezoid tree associated with the decomposition in part (a).

The above decomposition process is closely related to the one induced by the segment-tree. In particular, the leaves of $\mathcal{T}$ are in one-to-one correspondence with the fragments of the edges of $\mathcal{M}$, so that tree $\mathcal{T}$ has $O(n \log n)$ leaves. Since each node stores a constant amount of information, we have that the space requirement of the trapezoid tree $\mathcal{T}$ is $O(n \log n)$.

It is clear that a point location query LOCATE(q) can be performed by traversing a root-to-leaf path in $\mathcal{T}$, where at each internal node μ we branch left or right depending on the discrimination of the query point q with respect to the partitioning object stored at μ. Indeed, the leaf reached identifies an edge that is first hit by a horizontal ray through q. Since we did not impose any balance requirement on $\mathcal{T}$ the query time could be linear in the worst-case.

To speed-up queries, we implement $\mathcal{T}$ as a dynamic tree [29], i.e., $\mathcal{T}$ is decomposed into solid paths (such solid paths of $\mathcal{T}$ should not be confused with the solid paths in the trees of the normalization structure), connected by dashed arcs. Each solid path is associated with a path-tree. Note that the sequence of nodes of a solid path of $\mathcal{T}$ identifies a sequence of nested trapezoids. Point-location starts at the root of the path-tree of the topmost solid path of $\mathcal{T}$. At a given internal node η of a path-tree we consider the rightmost node ζ in the left subtree of η (readily available given thread pointers). We discriminate q with respect to the trapezoid τ of ζ and go to the left or right child of η according to whether q is inside or outside τ (recall that an inorder traversal of the path-tree corresponds to traversing the solid path from bottom to top). When we reach a leaf of the path-tree, i.e., a node μ of $\mathcal{T}$, we always exit on a dashed edge, and we always know the exit except for the case of the last node of the solid path, in which case we go to its left or right child by discriminating q with respect to the partitioning object of μ.

Let $(\nu_1, \mu_1), \cdots, (\nu_\ell, \mu_\ell)$ be the sequence of dashed edges traversed by the query algorithm, with ν_i the parent of μ_i. (Note that μ_ℓ is the leaf reached by the query algorithm.) Also, let μ_0 be the root of $\mathcal{T}$. Since path trees of standard dynamic trees are implemented as biased search trees, we have that the number of nodes visited in the solid path of ν_i is at most $\log(\mathit{weight}(\mu_{i-1})/\mathit{weight}(\nu_i)) + 2$. Hence, the time complexity of a point location query is $O(\sum_{i=1}^{\ell} \log(\mathit{weight}(\mu_{i-1})/\mathit{weight}(\nu_i)))$. Since $\mathit{weight}(\mu_i) < \mathit{weight}(\nu_i)$, the above sum telescopes, and we have that a point location query takes time $O(\log n)$.

To perform update operations, we establish bidirectional links between the trapezoid tree and the normalization structure. Let μ be a node of $\mathcal{T}$. We have

- If μ is a $\bigcirc$-node, let Q' be the subpath of a monotone path Q associated with the vertical cut at μ (i.e., the sleeve of Q' spans the trapezoid of μ and has an open hourglass). We establish pointers between μ and the nodes of $\mathit{ltree}(Q)$ and $\mathit{rtree}(Q)$ associated with Q'.
- If μ is a ∇-node, let Q' and R' be subpaths of monotone paths Q and R such that Q' and R' span the leftmost and rightmost regions in the trapezoid of μ. We establish pointers between μ and the nodes of $\mathit{ltree}(R)$ and $\mathit{rtree}(Q)$ associated with R' and Q', respectively. Also, we establish a back pointer from the allocation node y of μ in $\mathcal{Y}$ to μ.
- If μ is a $\square$-node, we establish pointers between μ and the two nodes in the normalization structure associated with the same edge fragment.

Note that every node of a path tree associated with an open hourglass is pointed to by exactly two nodes of $\mathcal{T}$.

Now, we discuss how update operations affect the trapezoid tree. Since the decomposition described by $\mathcal{T}$ is determined by the monotone paths, we update the trapezoid tree whenever monotone paths are changed in the normalization structure. We only need to consider the effects on the trapezoid tree of elementary splits, joins, partings, and pairings of monotone paths. Each such elementary operation in the normalization structure corresponds to performing $O(1)$ link and cut operations in the trapezoid tree. Details are shown in Figs. 8–9. Link and cut operations are performed in $O(\log n)$ time by standard dynamic tree algorithms. Regarding vertex insertions, a rotation at a node y in the *Y-tree* $\mathcal{Y}$ caused by a vertex update is handled by rebuilding the subtrees of $\mathcal{T}$ whose roots are ∇-nodes pointed by y. With an argument analogous to the one of Lemma 10, we can prove the following lemma.

Lemma 12 *Let y be a node of $\mathcal{Y}$ whose subtree has ℓ leaves. Then the total size of the subtrees of $\mathcal{T}$ whose roots are pointed by y is $O(\ell \log \ell)$.*

Hence the amortized cost of rebalancing $\mathcal{Y}$ in a sequence of updates is $O(\log^2 n)$. We conclude

Theorem 4 *Pont location queries* LOCATE(q) *in an n-vertex planar map can be performed in worst-case time $O(\log n)$ using a fully dynamic data structure that uses space $O(n \log n)$ and supports updates of the map in time $O(\log^3 n)$ (worst-case for edge updates, and amortized for vertex updates).*

Note that query LOCATE(q) is used in the update of the hull structure.

6 Ray Shooting

In this section we consider the problem of performing ray-shooting queries of the type:

SHOOT(q, d): Find the first vertex or edge hit by a query ray (q, d) in direction d originating at point q.

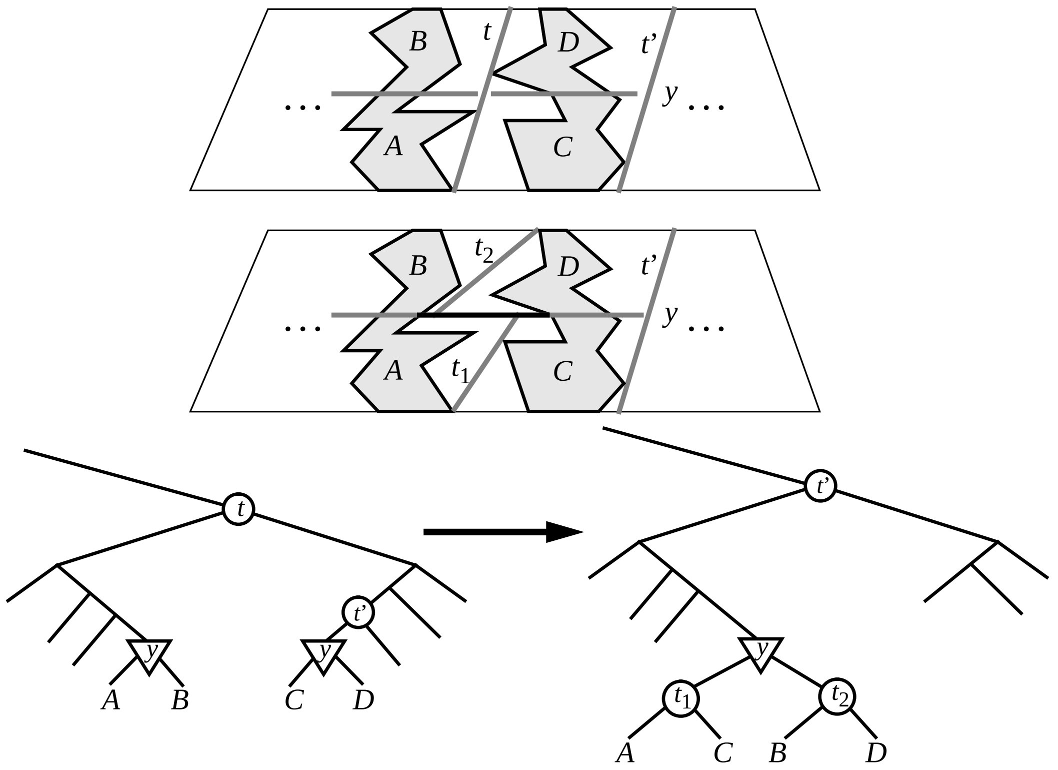

Figure 8: Update of the trapezoid tree in consequence of an elementary split of a monotone path in the normalization structure.

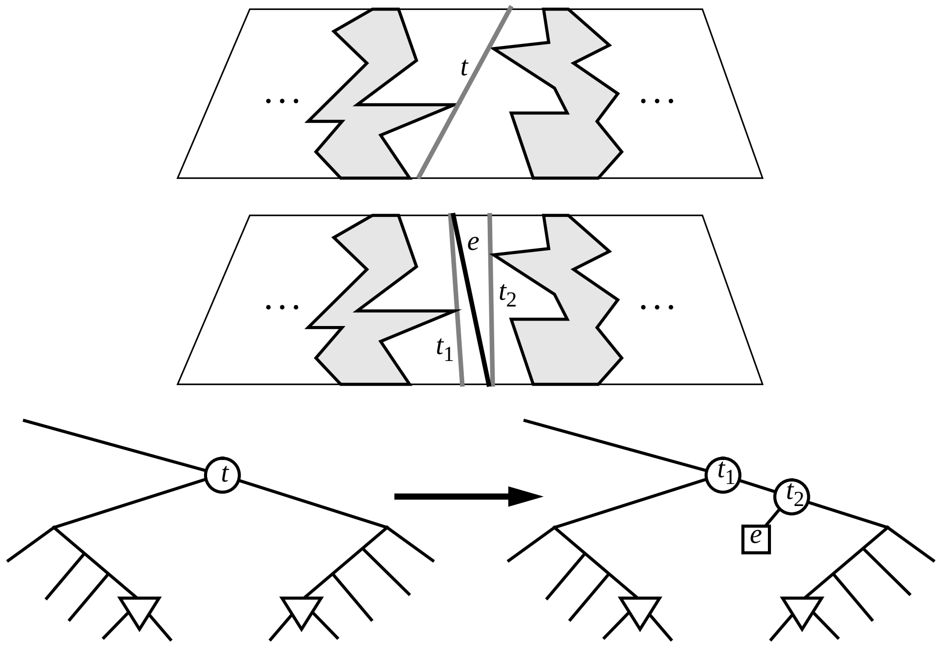

Figure 9: Update of the trapezoid tree caused by parting a monotone path in the normalization structure because of an edge insertion.

We show that the dynamic point location data structure presented in the previous section also supports—with no modification—ray-shooting queries in worst-case time $O(\log^3 n)$. Without loss of generality, assume that (q, d) is oriented upwards (otherwise symmetric arguments apply). The ray-shooting algorithm is as follows:

First, we perform $\text{LOCATE}(q)$ and determine the region r where q lies. If q lies on an a vertex or edge, an infinitesimal perturbation of q in direction d enables us to find the first region r entered by the ray. Query $\text{LOCATE}(q)$ also identifies the monotone sleeve $\text{SLEEVE}(Q)$ of r containing q and the splitter s_1 of $\text{SLEEVE}(Q)$ immediately below q. We show how to find the fist intersection q' of (q, d) with the boundary of $\text{SLEEVE}(Q)$. If q' is on a vertex or edge of r, then we report q' and stop; else (q' is on a lid of $\text{SLEEVE}(Q)$) we apply the algorithm recursively to the new ray (q', d).

We find the fist intersection q' of (q, d) with the boundary of $\text{SLEEVE}(Q)$ by the process below:

1. Find the topmost splitter s_2 of $\text{SLEEVE}(Q)$ such that $\text{HOURGLASS}(s_1, s_2)$ is open, by means of $O(\log n)$ elementary splits and joins of subpaths of Q that yield a a new monotone path R for $\text{SLEEVE}(s_1, s_2) = \text{SLEEVE}(R)$.
2. Find the first intersection p of (q, d) with $\text{SLEEVE}(R)$.
3. If p is not on s_2, then p is the intersection q' we want, so we return p and stop; otherwise (p is on s_2), there are two subcases: if s_2 is the top lid of $\text{SLEEVE}(Q)$, then return p and stop. Else, set $s_1 := s_2$, $q := p$, and repeat the process. Note that this situation can only occur at most twice, since s_2 is the topmost splitter such that $\text{HOURGLASS}(s_1, s_2)$ is open.

In step 2, the first intersection p of (q, d) with $\text{SLEEVE}(R)$ can be found by a binary search in the trees *ltree*(R) and *rtree*(R) as follows: at a current node μ with children μ' and μ'', we determine the intersection of (q, d) with the convex hull of $\text{CHAIN}(\mu)$. If the intersection is on a real edge or on a lid then we are done; else it is on a (fictitious) convex hull edge, and we compute the complete convex hulls of $\text{CHAIN}(\mu')$ and $\text{CHAIN}(\mu'')$, determine which one intersects with (q, d), and then repeat the process on that child.

The computation of point q' can be done in $O(\log^2 n)$ time with $O(\log n)$ elementary joins and splits of solid subpaths of Q. The number of recursive calls is $O(\log n)$ since the query ray intersects $O(\log n)$ lids. At the end, we conceal the path of $\Delta(r)$ traversed by the query ray to restore the weight invariant. We conclude with the following theorem.

Theorem 5 *Ray-shooting queries* $\text{SHOOT}(q, d)$ *in an* n*-vertex planar map can be performed in worst-case time* $O(\log^3 n)$ *using a fully dynamic data structure that supports updates of the map in time* $O(\log^3 n)$ *(worst-case for edge updates, and amortized for vertex updates).*

Theorem 5 also provides the capability of checking the validity of an edge insertion, i.e., whether the new edge does not intersect the current edges of the map. Moreover, as a corollary, we can perform stabbing queries, namely, determine the k edges of the map intersected by a query segment, in time $O(k \log^3 n)$.

References

[1] P. K. Agarwal and M. Sharir, "Applications of a New Space Partitioning Technique," *Lecture Notes in Computer Science*, Vol. 519, 379–391, 1991.

[2] H. Baumgarten, H. Jung, and K. Mehlhorn, "Dynamic Point Location in General Subdivisions," *Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms*, 1992.

[3] S.W. Bent, D.D. Sleator, and R.E. Tarjan, "Biased Search Trees," *SIAM J. Computing*, Vol. 14, 545–568, 1985.

[4] G. Bilardi and F. P. Preparata, "Probabilistic Analysis of a New Geometric Searching Technique," unpublished manuscript, 1981.

[5] B. Chazelle and L. J. Guibas, "Visibility and Intersection Problems in Plane Geometry," *Discrete and Computational Geometry*, 551–581, 1989.

[6] S.W. Cheng and R. Janardan, "New Results on Dynamic Planar Point Location," Technical Report TR 90-13, Dept. of Computer Science, Univ. of Minnesota, 1990. (Prelim. version: *31st FOCS*, 96–105, 1990.)

[7] S. W. Cheng and R. Janardan, "Space Efficient Ray Shooting and Intersection Searching: Algorithms, Dynamizations, and applications," *2nd SIAM-ACM Symposium on Discrete Algorithms*, 7–16, 1991.

[8] Y.-J. Chiang and R. Tamassia, "Dynamization of the Trapezoid Method for Planar Point Location in Monotone Subdivisions," *Proc. 7th ACM Symposium on Computational Geometry*, 61–70, 1991.

[9] Y.-J. Chiang and R. Tamassia, "Dynamic Algorithms in Computational Geometry," Technical Report CS 91-24, Dept. of Computer Science, Brown Univ., 1991.

[10] D. P. Dobkin and R. Lipton, "Multidimensional Searching Problems," *SIAM J. Computing*, Vol. 5, No. 2, 181–186, 1976.

[11] M. I. Edahiro, I. Kokubo and T. Asano, "A New Point-Location Algorithm and its Practical Efficiency—Comparision with Existing Algorithms," *ACM Transaction on Graphics*, Vol. 3, No. 2, 86–109, 1984.

[12] H. Edelsbrunner, L.J. Guibas, and J. Stolfi, "Optimal Point Location in a Monotone Subdivision," *SIAM J. Computing*, Vol. 15, 317–340, 1986.

[13] O. Fries, "Zerlegung einer planaren Unterteilung der Ebene und ihre Anwendungen," M.S. thesis, Inst. Angew. Math. and Inform., Univ. Saarlandes, Saarbrcken, Germany, 1985.

[14] O. Fries, K. Mehlhorn, and S. Naeher, "Dynamization of Geometric Data Structures," *Proc. (1st) ACM Symp. on Computational Geometry*, 168–176, 1985.

[15] M.T. Goodrich and R. Tamassia, "Dynamic Trees and Dynamic Point Location," *Proc. 23rd ACM Symp. on Theory of Computing*, 523–533, 1991.

[16] L. J. Guibas and J. Hershberger, "Optimal Shortest Path Queries in a Simple Polygon," *Proc. 3rd ACM Symposium on Computational Geometry*, 50–63, 1987.

[17] D. Kirkpatrick, "Optimal Search in Planar Subdivision," *SIAM Journal on Computing*, Vol. 12, 28–35, 1983.

[18] D.T. Lee and F.P. Preparata, "Location of a Point in a Planar Subdivision and its Applications," *SIAM J. Computing*, Vol. 6, 594–606, 1977.

[19] D. T. Lee and F. P. Preparata, "Euclidean Shortest Paths in the Presence of Rectilinear Barriers," *Networks*, Vol. 14, 393–410, 1984.

[20] K. Mehlhorn, *Data Structure and Algorithms 1: Sorting and Searching*, 189-199, 1984.

[21] M. Overmars, "Range Searching in a Set of Line Segments," *Proc. (1st) ACM Symp. on Computational Geometry*, 177–185, 1985.

[22] M. H. Overmars and J. van Leeuwen, "Maintenance of Configurations in the Plane," *J. Compt. and Syst. Sci.*, Vol. 23, 166–204, 1981.

[23] F.P. Preparata, "A New Approach to Planar Point Location," *SIAM J. Computing*, Vol. 10, 473-483, 1981.

[24] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, NY, 1985.

[25] F.P. Preparata and R. Tamassia, "Fully Dynamic Point Location in a Monotone Subdivision," *SIAM J. Computing*, Vol. 18, 811–830, 1989.

[26] F.P. Preparata and R. Tamassia, "Dynamic Planar Point Location with Optimal Query Time," *Theoretical Computer Science*, Vol. 74, 95–114, 1990.

[27] F.P. Preparata, J. Vitter, and M. Yvinec, "Computation of the Axial View of a Set of Isothetic Parallelepipeds," *ACM Transactions on Graphics*, Vol. 9, No. 3, 278–300, 1990.

[28] Sarnak, N. and R.E. Tarjan, "Planar Point Location Using Persistent Search Trees," *Communications ACM*, Vol. 29, 669–679, 1986.

[29] D.D. Sleator and R.E. Tarjan, "A Data Structure for Dynamic Trees," *J. Computer Systems Sciences*, Vol. 24, 362–381, 1983.

[30] R. Tamassia, "An Incremental Reconstruction Method for Dynamic Planar Point Location," *Information Processing Letters*, Vol. 37, 79–83, 1991.