A UNIFIED APPROACH TO DYNAMIC POINT LOCATION, RAY SHOOTING, AND SHORTEST PATHS IN PLANAR MAPS*

YI-JEN CHIANG†, FRANCO P. PREPARATA†, AND ROBERTO TAMASSIA†

Abstract. We describe a new technique for dynamically maintaining the trapezoidal decomposition of a connected 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). This is the first polylog-time dynamic data structure for shortest-path and ray-shooting queries. It is also the first dynamic point-location data structure for connected planar maps that achieves optimal query time.

Key words. point location, ray shooting, shortest path, computational geometry, dynamic algorithm.

AMS subject classifications. 68U05, 68Q25, 68P05, 68P10

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 in asymptotically time- and space-optimal methods [12, 17, 29] 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 [2, 6, 8, 14, 15, 21, 25, 26, 31].

All the known dynamic point location results are for connected maps, since maintaining region names in a disconnected map would require solving half-planar range searching in a dynamic environment, for which no polylog-time algorithm is known. The best results to-date for dynamic point-location in an n-vertex connected map are due to Cheng-Janardan [6] and Baumgarten-Jung-Mehlhorn [2]. The technique of [6] achieves $O(\log^2 n)$ query time, $O(\log n)$ update time, and $O(n)$ space. The data structure of [2] has query and insertion time $O(\log n \log\log n)$, deletion time $O(\log^2 n)$, using $O(n)$ space, where the time bounds are amortized for the updates. 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 technique that supports $O(\log n)$-time queries in

* An extended abstract of this paper was presented at the *4th ACM-SIAM Symposium on Discrete Algorithms*, Austin, Texas, January 1993. 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 grants DAAL03-91–G–0035 and DAAH04–93–0134, and by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014–91–J–4052, ARPA order 8225.

† Department of Computer Science, Brown University, Providence, RI 02912–1910. E-mail addresses: {yjc,franco,rt}@cs.brown.edu.

a dynamic environment is restricted to monotone maps [8]. For a survey of dynamic point-location techniques and other dynamic algorithms in computational geometry, see 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 in a dynamic setting, although a polylog-time ray-shooting technique by Reif and Sen [28], designed for monotone polygons, may be extensible to the general case. Sublinear-time techniques are known only for ray-shooting queries [1, 7], with $O(\sqrt{n}\,\mathrm{polylog}(n))$ query/update time; they 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, yet the exclusion of more general maps is a severe shortcoming. In the static case, a connected 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.

In this paper we combine 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 connected 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 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 structure—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 naturally augmented with node-appended secondary structures to support shortest-path and ray-shooting queries. It can also be supplemented with a distinct, but tightly coupled, location structure designed for efficient point-location. The main normalization structure and its two auxiliary components act in a tightly integrated fashion: 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 1.1. *There exists a fully dynamic data structure that supports point-location, ray-shooting, and shortest-path queries in a connected 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 $O((k+1)\log^3 n)$ 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 connected 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 connected 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 connected 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 *connected planar map* $\mathcal{M}$ is a subdivision of the plane into polygonal regions whose underlying planar graph is connected. The map is augmented with two vertical rays, one directed toward $y=+\infty$, the other toward $y=-\infty$, respectively issuing from the vertices of $\mathcal{M}$ with maximum and minimum y-coordinates. Thus, all but two regions of $\mathcal{M}$ are bounded simple polygons. In the following, n denotes the number of vertices of the planar map $\mathcal{M}$ currently being considered. Also, we assume that no two vertices of $\mathcal{M}$ have the same y-coordinate; the degenerate cases can be handled by standard techniques and will not be discussed in this paper.

In the plane we have an orthogonal frame of reference (x, y). A polygonal chain γ is *monotone* if any horizontal line intersects it in a single point or in a single interval or not at all. 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 connected map $\mathcal{M}$ is obtained by drawing from each vertex v of $\mathcal{M}$ two horizontal rays that either remain unbounded or 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. Let s_i, $i = 1, 2$, denote either a splitter or an extreme vertex of region r. Then $\text{SLEEVE}(s_1, s_2)$ 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.

Three special types of data structures will be used in this paper: biased binary trees [3], $BB[\alpha]$-trees [20], and dynamic trees [30].

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 $\frac{1}{4} < \alpha \leq 1 - \frac{\sqrt{2}}{2}$) 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 $O(\log^2 n)$.

Dynamic trees [30] are designed to represent a forest of rooted trees, with each edge directed toward the root of its tree (and called an *arc*). Some important operations (among others) supported by dynamic trees include:

link(μ, ν): Add an arc from μ to ν, thereby making μ a child of ν in the forest. This operation assumes that μ is the root of one tree and ν is a node of another tree.

cut(μ): Delete the arc from μ to its parent, thereby dividing the tree containing μ into two trees.

evert(μ): Make μ the root of its tree by reversing the path from μ to the original root.

Each arc of the trees is classified as *solid* or *dashed*, so that each tree is partitioned into a collection of *solid paths*, connected by dashed arcs. A solid path is maintained by a data structure called a *path tree*. Using biased binary trees [3] as the standard implementation of path trees, each of the above operations takes $O(\log n)$ time, where n is the size of the tree(s) in the forest involved.

3. The dynamics of trapezoidal decompositions. Given a connected 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 region r, bounded or unbounded. In the following, we denote by m the current number of vertices in r.

We imagine to represent $\delta(r)$ as a dynamic tree $\Delta(r)$ [30] (see Fig. 1). We choose an arbitrary node of $\delta(r)$ as the *root*, which immediately forces a direction on each edge, referred to hereafter as an *arc* and directed toward the root. Since we have chosen to dualize each trapezoid to a point in its interior, the y-component of each arc has a well-defined sign. An arc is usually denoted either by a single letter or by an ordered pair (origin, destination). The arcs are classified as follows: letting $w(\mu)$, *weight* of μ, denote the number of nodes in the subtree rooted at node μ, an arc (μ, ν) is classified *heavy* if $w(\mu) \geq \frac{1}{2}w(\nu)$, and *light* otherwise. Consequently, at most one heavy arc enters a node of $\Delta(r)$. Note that the attributes {light, heavy} pertain uniquely to the weight structure of the dynamic tree $\Delta(r)$.

Arcs are also classified as *solid* or *dashed* to enforce the property that *at most one solid arc enters a node* of $\Delta(r)$. The maximal paths of consecutive solid arcs (possibly consisting of a single node) are called *solid paths*, and each corresponds to a sleeve of r. Note that the attributes {dashed, solid} pertain to a given, but otherwise arbitrary, decomposition of r into sleeves.

The weight structure and the sleeve decomposition are tied by the following *weight invariant*, which holds before and after the execution of data structure operations (queries or updates):

"heavy arcs are solid and light arcs are dashed."

However, during the execution of operations, we may change heavy arcs to dashed and light arcs to solid, and thus loose the original correspondence. The weight invariant is restored at the completion of each operation.

Region r contains a set of splitters, called *lids*, which are the duals of the following arcs:

Rule 1. All dashed arcs.

Rule 2. Any two consecutive solid arcs whose y-components have opposite signs.

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

LEMMA 3.1. *The set of lids partitions r into a collection of monotone polygons.*

Proof. Let c be a cusp of polygon r. We consider the two arcs of $\Delta(r)$ which are the duals of the two splitters issuing from c. If at least one of them is dashed (see Fig. 2(a)), then there is at least one lid issuing from cusp c corresponding to the dashed arc (Rule 1). If on the other hand both arcs are solid, then one must have a positive y-component and the other a negative one, or otherwise they would enter the same node of $\Delta(r)$ and thus would violate the property that at most one solid

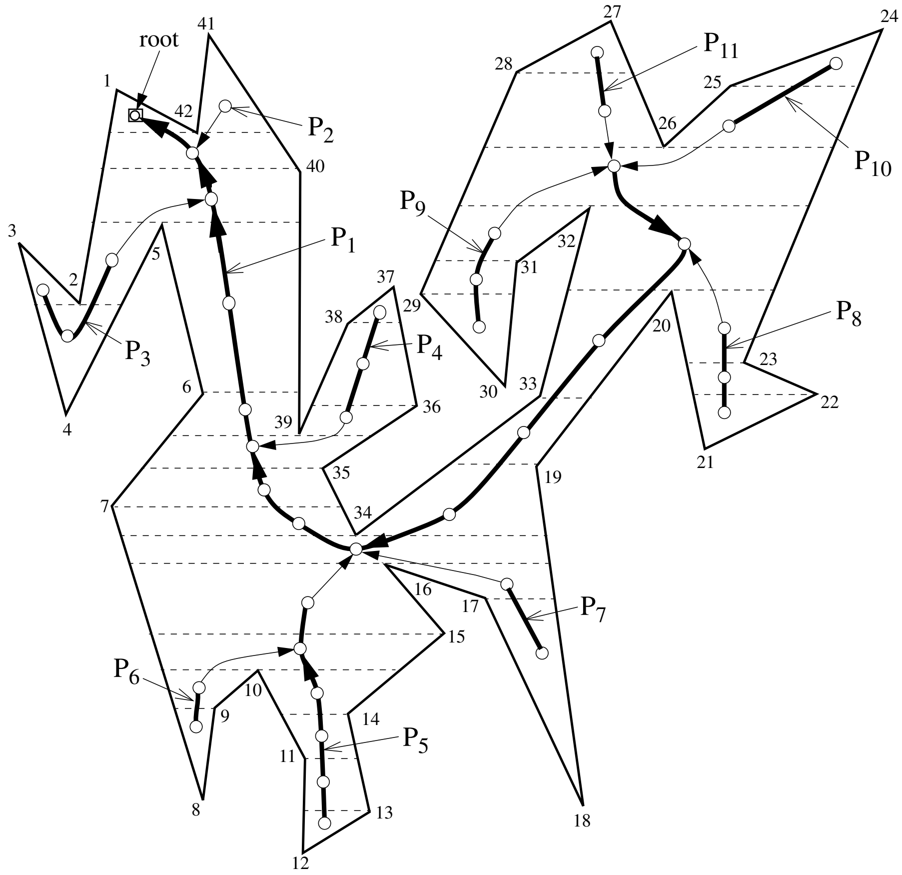

FIG. 1. *Example of a region* r *and its dynamic tree* $\Delta(r)$ *(*$P_1, \ldots, P_{11}$ *are solid paths).*

arc enters a node (see Fig. 2(b)). Then these two arcs are consecutive solid arcs with y-components of opposite signs, and there are two lids from c corresponding to these arcs (Rule 2). Hence there is always at least one horizontal lid issuing from each cusp c of r, thereby achieving a decomposition of r into monotone polygons. □

LEMMA 3.2. *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 $w(child) < \frac{1}{2}w(parent)$. □

COROLLARY 3.3. *Any straight line drawn in region* r *crosses* $O(\log m)$ *lids.*

Proof. The weight invariant is always preserved before and after the execution of data structure operations. Each lid then corresponds to either (i) a light arc (Rule 1, since dashed arcs are light), or (ii) a solid arc at which the solid path containing this arc changes monotonicity with respect to the y-axis (Rule 2). By Lemma 3.2, any straight line l drawn in r crosses $O(\log m)$ lids of type (i). Now consider the lids of type (ii). Lemma 3.2 also implies that l goes through $O(\log m)$ solid paths. Observe that each solid path P can be partitioned into maximal monotone subpaths, and l can go through at most one such monotone subpath, thus crossing at most two lids of solid arcs of P. It follows that the number of lids of type (ii) crossed by l is also $O(\log m)$. □

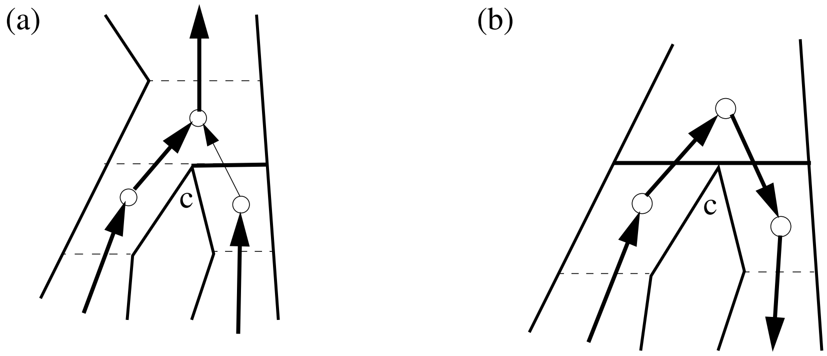

FIG. 2. *Proof of Lemma 3.1.*

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 $\Delta(r)$ by two "threads"; these two threads respectively correspond to two chains whose union is the boundary of the sleeve associated with the solid path. The proposed structure is referred to as *double-thread data structure* for region r, denoted by $DT(r)$.

Each arc α of $\Delta(r)$ can be drawn to intersect 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 with two arcs: 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 choose to 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. As a consequence, each solid path P is represented by two binary trees $lthread(P)$ and $rthread(P)$, referred to as *thread 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 dashed arc α (every solid path except the one terminating at the root of $\Delta(r)$ has this property), 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 (on the basis of the signs of their y-components) subpaths $Q_1, Q_2, \cdots, Q_k$. Our thread trees $lthread(P)$ and $rthread(P)$ are each implemented as a two-level (called lower and upper) balanced binary tree (i.e., the roots of lower-level trees are leaves of the upper-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 of Q_i in their path order. Thread tree $rthread(P)$ is analogously organized, with $rtree(Q_i)$ storing the right vertices. The roots of $ltree(Q_i)$ and $rtree(Q_i)$ are bidirectionally linked. In the upper 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. A bidirectional link also exists between the roots of $lthread(P)$ and $rthread(P)$. An example is shown in Fig. 5(a).

Any node on P might be pointed to (via dashed arcs) by some other solid paths in the dynamic tree $\Delta(r)$. Suppose that P' points to P via an arc α' associated with vertex v'. Two situations may now occur: (i) vertex v' is also associated with an arc of P (e.g., see paths P_2, P_3 and P_4 in Fig. 1 with $P = P_1$). Then v' is a left or right vertex of P (thus stored as a lower-level leaf of $lthread(P)$ or of $rthread(P)$). We establish a pointer from each root of $lthread(P')$ and $rthread(P')$ to that lower-level leaf v' (see Fig. 5(b)). The possible instances of this situation are illustrated in Figure 3(b, d). (ii) vertex v' is not associated with an arc of P (e.g., see paths P_5 and P_7 in Fig. 1 with $P = P_1$). This occurs if P changes monotonicity (by crossing both splitters of a cusp c) at the node reached by arc α'. In this case, in order to provide a destination for the pointers from the roots of $lthread(P')$ and $rthread(P')$, we introduce an auxiliary leaf, called a *coupler* (usually denoted by letter H), inserted between the two consecutive subtrees (both either $ltree$'s or $rtree$'s) of the thread tree not containing cusp c (see Fig. 5(b)). The possible instances of this situation are illustrated in Figure 3(c, e).

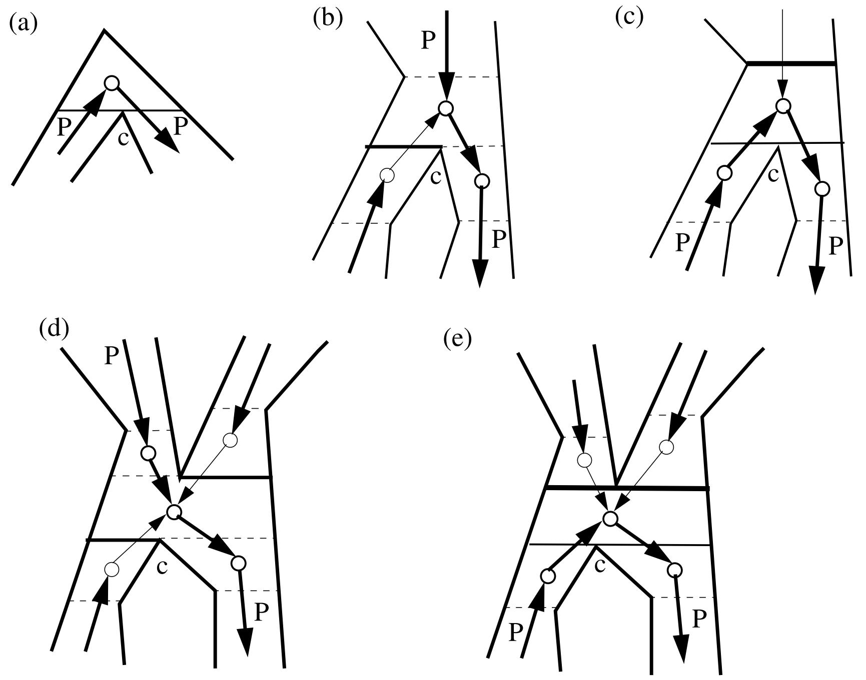

FIG. 3. *All possible cases in which a solid path P crosses a splitter issuing from a cusp c. Note that P does not change monotonicity (i.e., crosses only one splitter issuing from c) in (b) and (d), and P changes monotonicity (i.e., crosses both splitters issuing from c) in (a), (c) and (e).*

Note that a pointer destination may be needed when a solid path P begins (Fig. 4(b, c) and Fig. 1 for $P = P_1$). In this case we adopt the convention to insert a coupler preceding either $ltree(Q_1)$ or $rtree(Q_1)$, where Q_1 is the initial monotone subpath of P (see Fig. 5(b)). The overall data structure $DT(r)$ consists therefore of two rooted trees of indegree at most 4 (see Fig. 5(b)).

We now define a new parameter of nodes of $DT(r)$ (*DT-nodes*), called *charge*, which will be used to maintain the weights of the nodes of the dynamic tree $\Delta(r)$. Each DT-node corresponding to a vertex of r (a leaf of a lower-level tree) is labelled *distinguished*; the charge of a DT-node is the number of the distinguished nodes in

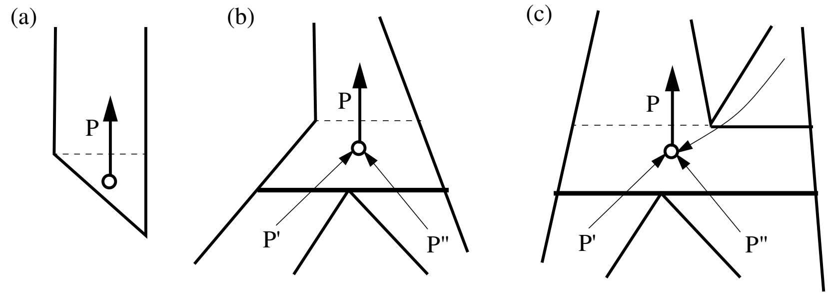

FIG. 4. *All possible cases in which a solid path P starts. Note that a coupler of P is needed to provide a destination of P' and P'' in (b) and (c).*

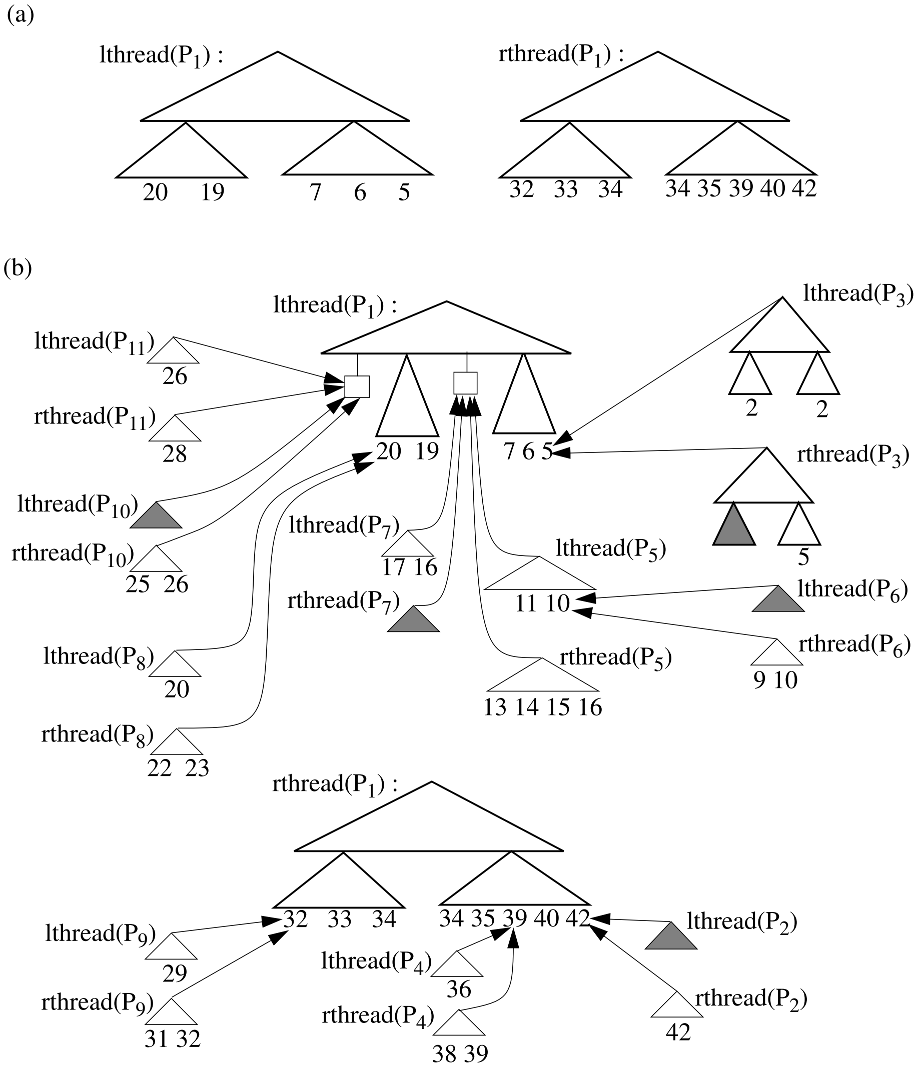

FIG. 5. *Double-thread data structure $DT(r)$ for region r in Fig. 1: (a) basic thread trees for P_1; (b) complete structure of $DT(r)$. The bidirectional pointers linking pairs of corresponding thread trees and thread subtrees are omitted.*

the subtree of which it is the root.

According to its definition, the weight $w(\mu)$ of a node μ of $\Delta(r)$ is the number of the nodes in the subtree of which it is the root, or, equivalently, the number of the arcs in this subtree plus the arc α issuing from μ. It is immediate that, denoting by v the vertex associated with arc α and by Q_i the monotone subpath containing α, this number is obtained as the sum of two items: (1) the sum of the charges of all lower-level leaves (actually leaves or couplers) up to and including v in the thread tree containing v, and (2) in the other thread tree, the sum of the charges of all lower-level leaves *preceding* v^*, where v^* is the first vertex on monotone subpath Q_i whose splitter follows the splitter issuing from v, or if v^* does not exist (because Q_i terminates at v), the sum of the charges of all lower-level leaves up to and including the last leaf of the appropriate subtree of Q_i (either $ltree(Q_i)$ or $rtree(Q_i)$). For example, let us look at $w(\mu_1)$ and $w(\mu_2)$ in Figure 6. For $w(\mu_2)$, $v^* = s$, thus $w(\mu_2)$ is the sum of the charges of all lower-level leaves of $rthread(P)$ from left up to and including v which corresponds to μ_2, and the charges of all lower-level leaves of $lthread(P)$ up to and including coupler H; for $w(\mu_1)$, v^* does not exist, and thus $w(\mu_1)$ is the sum of the charges of all lower-level leaves of $rthread(P)$ up to and including v which corresponds to μ_1, and the charges of all lower-level leaves of $lthread(P)$ up to and including u. Clearly, we can locate v^* or decide its nonexistence in logarithmic time, using the y-coordinate of v to perform a binary search on either $ltree(Q_i)$ or $rtree(Q_i)$ of the thread tree that does not contain v.

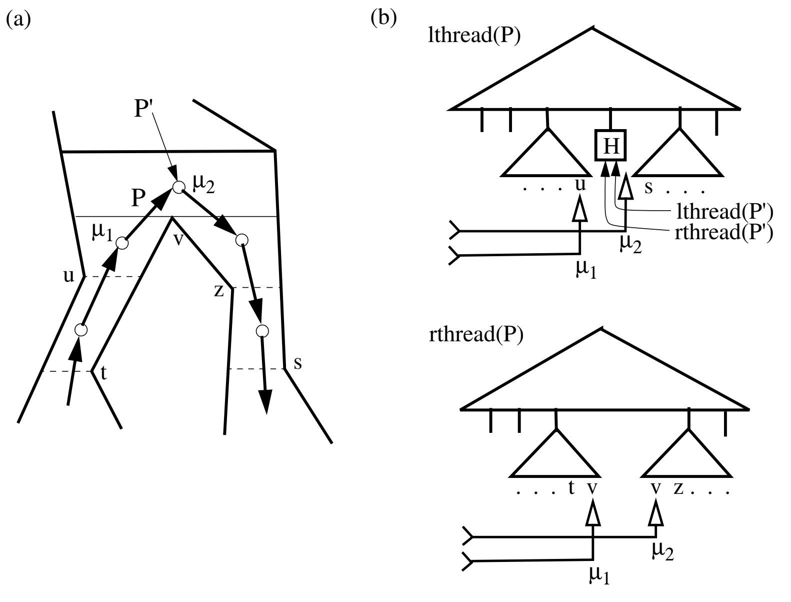

FIG. 6. *Weights $w(\mu_1), w(\mu_2)$ of nodes μ_1, μ_2 of the dynamic tree $\Delta(r)$: $w(\mu_2)$ is the sum of the charges of all lower-level leaves of $rthread(P)$ from left up to and including the occurrence of v which corresponds to μ_2, and the charges of all lower-level leaves of $lthread(P)$ up to and including coupler H; $w(\mu_1)$ is the sum of the charges of all lower-level leaves of $rthread(P)$ up to and including the occurrence of v which corresponds to μ_1, and the charges of all lower-level leaves of $lthread(P)$ up to and including u.*

The preceding discussion establishes the following lemma.

LEMMA 3.4. *The space complexity of the normalization structure for an n-vertex map is $O(n)$.*

Our data structure has an auxiliary component, called *dictionary*. The *dictionary* stores the names of vertices, edges and regions, so that their representatives occurring in various places in the normalization structure, hull structure and location structure (see Sections 4 and 5), etc., can be efficiently accessed. The edges of a region r are also maintained in the dictionary by a balanced binary tree according to their circular order, with the root of the tree storing the name of r. We store with each edge e two pointers respectively to its left and right representatives in such trees, so that given e, the region r to its left (resp. right) can be found by accessing its left (resp. right) representative and walking up to the root of the tree of r. It is easy to see that accessing and updating the dictionary can be performed in logarithmic time, and that the dictionary does not affect the space complexity of our data structure. Therefore we omit any further discussion of the dictionary in the rest of the paper.

3.3. Update operations. We define the following update operations on a connected 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 3.5. *An arbitrary connected 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 *}.*

Now 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 (which also provides the trapezoid containing v_2), 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 *expose, conceal* and *evert* introduced in [30], 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 [30]. Operation *expose*(μ), 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, the inverse operation *conceal*(P) is used 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. 7). 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).

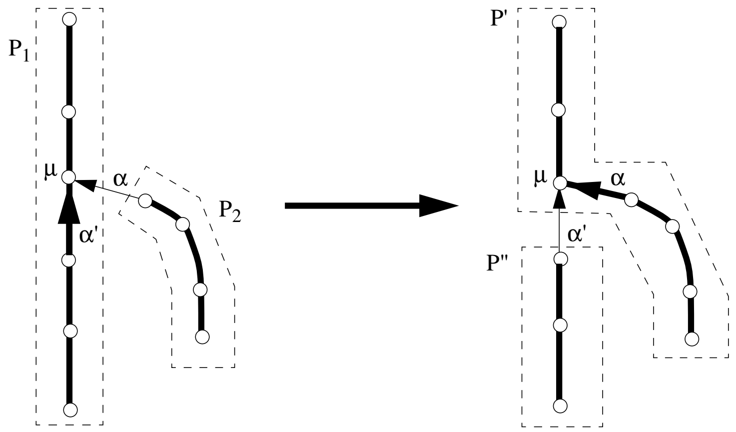

FIG. 7. *Example of splice*$(P_1, P_2; P', P'')$.

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*. Operation *splice* may require either the insertion or the deletion of a coupler (see, for example, splicing P_4 to P_1 (insertion) and P_5 to P_1 (deletion) in Fig. 1). Since a constant number of splits/concatenations have to be performed, we have the following lemma.

LEMMA 3.6. *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 3.2 (each accessible by climbing to the root of an $O(\log m)$-depth thread tree), *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 light arcs of P which have to be made dashed and the heavy arcs (if any) incident to P which have to be made solid 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. The heavy arcs incident to P can then be identified (and made solid) in a straightforward way: each time a light arc (μ, ν) is found, we check all (up to three) arcs incident to ν to see if any one of them is heavy; finally, we also apply this checking process to the arcs incident to the bottommost node of P. So the main issue for performing $conceal(P)$ is how to find the topmost light arc.

Before describing its adaptation to the double-thread data structure, we briefly review the standard implementation of operation *conceal* as proposed by Sleator and Tarjan [30]. Let the dynamic-tree-nodes of solid path P be stored left-to-right as the leaves of a balanced binary tree $T(P)$, called in [30] a *path tree*. Each leaf ζ of $T(P)$ stores $local\_weight(\zeta)$, defined as the sum of the local weights of all dashed-arc children (which are the roots of some other path trees) of ζ, if any, plus one (to account for ζ itself). For each internal node η, $local\_weight(\eta)$ is defined as the sum of the local weights of its children. Note the similarlity between the *local weights* of nodes in path tree $T(P)$ and *charges* of nodes in thread trees $lthread(P)$ and $rthread(P)$ defined in Section 3.2. Actually, parameters local weight and charge are identical except for their usages in computing $w(\mu)$—the weight of a dynamic-tree-node μ: in $T(P)$, $w(\mu)$ is the left to right prefix-sum of the local weights of the leaves, whereas in thread trees, $w(\mu)$ is contributed by the prefix-sums of the charges of the leaves in *both* $lthread(P)$ and $rthread(P)$ (see Section 3.2). Let T_η be the subtree of $T(P)$ rooted at internal node η. Denoting by λ the rightmost leaf in T_η, and by ξ the leaf adjacent to λ on the left, variable $lefttilt(\eta)$ is defined by $lefttilt(\eta) \stackrel{\triangle}{=} w(\xi) - local\_weight(\lambda)$. We recall that arc (ξ, λ) of P is light if and only if $w(\xi) < \frac{1}{2}w(\lambda)$, i.e., $w(\xi) < \frac{1}{2}(w(\xi) + local\_weight(\lambda))$, which yields $lefttilt(\eta) < 0$.

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

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 η, parameter $leftmin(\eta)$ (< 0) is known. Let η' and η'' be the left and right children of η, respectively, and ξ be the leftmost leaf of $T_{\eta''}$. From the definition

$$netleft(\eta'') = leftmin(\eta'') - leftmin(\eta)$$

we obtain $leftmin(\eta'')$. If $leftmin(\eta'') < 0$, then we proceed to η''. Otherwise, we compare $local\_weight(\eta')$ and $local\_weight(\xi)$. If $local\_weight(\eta') < local\_weight(\xi)$, then the arc leading to ξ is the sought light arc; else, we compute $leftmin(\eta') = leftmin(\eta) + netleft(\eta')$ (which is necessarily < 0) and proceed to η' (this establishs the inductive step). By this process akin to binary search, the topmost light arc can be found in $O(\log m)$ time. Recall that by Lemma 3.2, 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 thread trees $lthread(P)$ and $rthread(P)$ independently as two path trees, with parameter *charge* playing the role of *local weight*. 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 (ξ, λ) in P assures the existence of a light arc (ξ', λ) in either $lthread(P)$ or $rthread(P)$ that contains leaf λ, where ξ' is the left-neighboring leaf of λ. Indeed, in $T(P)$, the sum $w(\xi)$ of the local weights up to and including ξ satisfies $w(\xi) < local\_weight(\lambda)$; but in the appropriate thread tree (i.e., either $lthread(P)$ or $rthread(P)$ that contains λ), the sum w' of the charges up to and including ξ' is only a fraction of $w(\xi)$ ($w(\xi)$ is contributed by *both* $lthread(P)$ and $rthread(P)$), so that $w' \leq w(\xi) < local\_weight(\lambda) = charge(\lambda)$, and (ξ', 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 (ξ', λ), we perform a binary search in the paired thread tree to locate the point just before λ, at the same time accumulate the total charge w'' up to and including this point in that tree, then compute $w(\xi)$ by adding w'' to w', and check if $w(\xi) < charge(\lambda)$ $(= local\_weight(\lambda))$. 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 also at most $\log_2 m + 1$ heavy arcs incident to P), and then split and join P accordingly—each of these operations within $O(\log m)$ time. This leads to the following lemma.

LEMMA 3.7. *The update of the double-thread data structure as required by the operation conceal can be performed in $O(\log^2 m)$ time.*

Operation $evert(\mu)$, for an arbitray node μ of $\Delta(r)$, moves the root of $\Delta(r)$ to μ while preserving the weight invariant. If we can reverse the direction of a solid path, then $evert(\mu)$ can be carried out as follows: we perform $expose(\mu)$ to obtain a solid path P from μ to the original root, reverse the direction of P (which effectively moves the root to μ), and then perform $conceal(P)$ to comply with the weight invariant. We add a "direction" bit to each node of the thread trees, so that when we reverse the direction of a solid path P, the direction bit of the root of $lthread(P)$ is complemented, indicating that the meanings of left and right subtrees of $lthread(P)$ are interchanged; similarly for the direction bit of the root of $rthread(P)$. Also, these two complemented bits indicate that $lthread(P)$ means $rthread(P)$ and *vice versa*. Given the direction bits and operations *expose* and *conceal*, 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 $expose(\mu)$ are equivalent, and similarly for *evert*.

LEMMA 3.8. *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 thread 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 3.9. *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 then there are two such splitters; by viewing edge $e = (v_1, v_2)$ as issuing from v_i, s_i is taken as the left splitter of v_i if e goes toward left (and as the right splitter otherwise), so that SLEEVE(s_1, s_2) is the smallest monotone sleeve that contains e.
2. Construct SLEEVE(s_1, s_2) and the corresponding solid path P, by performing $evert(\delta(s_1))$ and then $expose(\delta(s_2))$.
3. Insert edge e by performing $part(P, e; P_1, P_2)$, so that there are new solid paths P_1 and P_2 respectively in new regions r_1 and r_2.
4. For each of the (up to three) solid paths previously pointing to the head of P, make it point to the head of P_1 if it lies in r_1, and to the head of P_2 if it lies in r_2; similarly for the solid paths previously pointing to the tail of P. Note that P_1 and P_2 have the same orientation as P.
5. Create a new dynamic tree $\Delta(r_1)$ for r_1, by putting the root at the end of P_1 that is closer to v_1 (which does not change the direction of P_1), then performing operation $conceal(P_1)$; similarly create a new dynamic tree $\Delta(r_2)$. Note that the *conceal* operations readily splice the solid paths pointing to the heads and tails of P_1 and of P_2 if necessary.

We analyze the time complexity of the above operation. Steps 1, 3 and 4 take $O(1)$ time, and the other steps globally involve a fixed number of *evert*, *expose* and *conceal* operations, so that the total time required 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_1 and then expose v_2 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 insert v with $charge(v) = 1$ into $lthread(P)$ and $rthread(P')$ for some solid paths P and P' of different regions r and r', where both $lthread(P)$ and $rthread(P')$ contain two end-points v_1 and v_2 of e. In dynamic tree $\Delta(r)$, we perform *expose* on the one of v_1 and v_2 that is farther from the root to obtain a solid path, and then perform *conceal* on this path; in $\Delta(r')$ we perform exactly the same operations. It is easy to see that operation INSERTVERTEX is 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 can be modified, by appending secondary data structures collectively called *hull structure*, to answer the following queries:

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 as a chain of segments.

First, by point location (see Section 5) we can check whether q_1 and q_2 belong to r. Note that we need to specify within which region the shortest path is sought to avoid ambiguities when both q_1 and q_2 belong to edges of the subdivision. We now show that the above queries can be supported 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.

The notion of *hourglass* is central to our current problem. We adopt the terminology 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 counterclockwise 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 shortest paths PATH(q_1, q_2, r) with $q_1 \in s_1$ and $q_2 \in s_2$ (see Fig. 8.b). It is known that the boundary of HOURGLASS(s_1, s_2) is the concatenation of s_1, PATH(a_1, a_2, r), s_2, and PATH(b_2, b_1, r). Let α be the subchain of r counterclockwise from a_1 to 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 inside r of α and β do not intersect, then PATH(a_1, a_2, r) is the convex hull of the subchain of α clockwise from a_1 to a_2, and similarly for PATH(b_2, b_1, r).

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$(a_1, a_2, r) \cap$ PATH$(b_1, b_2, r) =$ PATH(p_1, p_2, r). Without loss of generality, assume that p_1 is in α. Then PATH(a_1, p_1, r) is the convex hull inside r of the subchain of α from a_1 to p_1, while PATH(b_1, p_1, r) is the union of segment (p_1, p_1') and the convex hull inside r of the subchain of β from b_1 to p_1', where p_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, r) and PATH(b_i, p_i, r) ($i = 1$ or 2) is called a *funnel* [19]. 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. 8.b).

If we represent an hourglass by its string and the (two to four) convex chains forming the rest of its boundary, and for each polygonal chain represented we also store its length, then 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 the line containing 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 three items are needed, i.e.:

(i) The choice of an appropriate implementation of the trees *ltree* and *rtree* introduced in Section 3.2;

(ii) The appending of secondary data structures (collectively called "hull structure") to the nodes of *ltree*'s and *rtree*'s. The *hull structure* stores at the nodes of *ltree*'s and *rtree*'s 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 as required by edge insertion or deletion;

(iii) A separate BB[α]-tree $\mathcal{Y}$ (called *Y-tree*) that determines a hierarchical partition of the plane into horizontal strips, according to which *ltree*'s and *rtree*'s are implemented.

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 and has inorder thread pointers. 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 the degree of each vertex of $\mathcal{M}$ 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[α]-tree $\mathcal{Y}$, called *Y-tree*, which is added as a separate tree into the data structure. 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 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 subtree 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 structure so that hourglasses can be dynamically maintained (see Fig. 8). We denote with Q a maximal monotone subpath of a solid path P 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

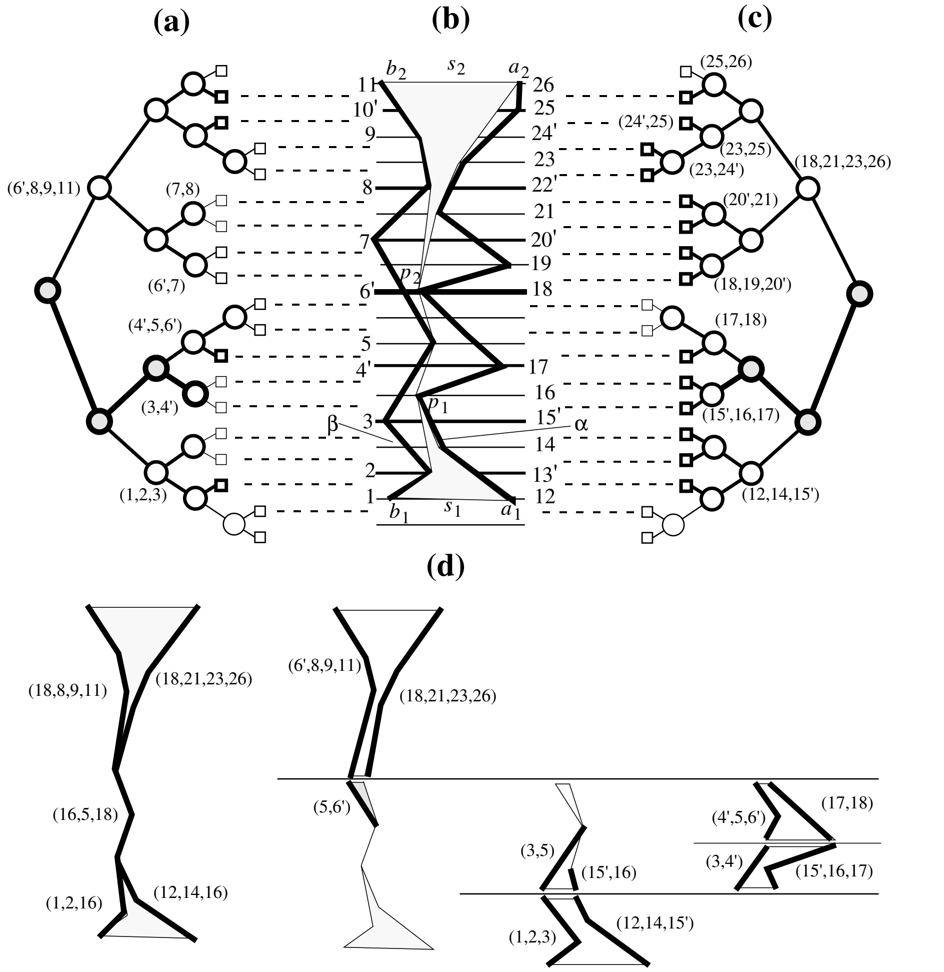

FIG. 8. *Example of representation of hourglasses in the nodes of ltree(Q) and rtree(Q) of a monotone path Q.* **(b)** *The sleeve of Q (directed from left to right): the parallel lines drawn on it represent set Y; the points on the sleeve with labels of the type i' delimit fragments of the same edge; the hourglass between the extreme splitters of the sleeve is shown grey-filled.* **(a)** *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 (a) 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.*

$\mathcal{Y}$. Also, if μ is not a leaf, then we establish a back pointer from y to μ. We do not set up back pointers from y to leaves of $ltree(Q)$ (or of $rtree(Q)$) in order to obtain efficient updates, as we shall see later. Consider the subpath Q' of Q associated with the subtree of $ltree(Q)$ rooted at μ. We denote with SLEEVE(μ) the sleeve of Q', with s_1 and s_2 the splitters that delimit SLEEVE(μ), with HOURGLASS(μ) the hourglass of s_1 and s_2 (namely, HOURGLASS(s_1, s_2)), and with CHAIN(μ) the "left chain" of SLEEVE(μ), 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 HOURGLASS(μ) is open and HOURGLASS(ν) is closed, then μ stores in a secondary data structure the right convex hull of CHAIN(μ).
- If both HOURGLASS(μ) and HOURGLASS(ν) are open, then μ stores in a secondary data structure only the endpoints of CHAIN(μ) and the portion of the right hull of CHAIN(μ) that is not stored at an ancestor of μ.
- If HOURGLASS(μ) is closed and μ is the root of *ltree*(Q), then μ stores in secondary data structure the (up to five) components of HOURGLASS(μ).
- If HOURGLASS(μ) is closed and μ is not the root, then μ stores the apices and the length of the string of HOURGLASS(μ), plus the subchains of the funnels of HOURGLASS(μ) that are not stored at the ancestors of μ.

• The upper levels (see Section 3.2) of thread trees *lthread*(P) and *rthread*(P) are essentially identical (except for the couplers). Also, an internal node μ in the upper level of *lthread*(P) stores the length and the endpoints of the string of HOURGLASS(μ). The corresponding node of *rthread*(P) stores exactly the same information.

LEMMA 4.1. *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 $ltree$'s and $rtree$'s. 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 node in $ltree$ or $rtree$. We claim that the size of S is $O(n \log n)$. By standard segment-tree arguments, the number of edge-fragments in S is $O(n \log n)$. For the tangent segments, consider the hourglasses HOURGLASS(μ), HOURGLASS(μ') and HOURGLASS(μ'') of a node μ and its children μ' and μ''. Note that SLEEVE(μ') and SLEEVE(μ'') share a common splitter, say s_2, and the other splitters s_1 of SLEEVE(μ') and s_3 of SLEEVE(μ'') lie on opposite sides of s_2. It follows that HOURGLASS(μ) = HOURGLASS(s_1, s_3) is obtained from HOURGLASS(μ') = HOURGLASS(s_1, s_2) and HOURGLASS(μ'') = HOURGLASS(s_2, s_3) by $O(1)$ common tangent computations, and thus each node μ contains $O(1)$ tangent segments. Again, by segment-tree arguments, the total number of nodes in $ltree$'s and $rtree$'s is $O(n \log n)$, hence the total number of tangent segments in S is $O(n \log n)$. Also, each segment of S is stored $O(1)$ 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 may be two such nodes for each segment (recall that edges have two "sides", and the corresponding nodes in the paired $ltree$ and $rtree$ may have duplicate information). We conclude that the secondary 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, using the point-location machinery of 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), after computing in time $O(\log n)$

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. 9):

LEMMA 4.2. *An elementary split or join of two thread 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 thread 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. 9). 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 length of the string in the nodes of the upper levels of the *lthread* and *rthread* trees. □

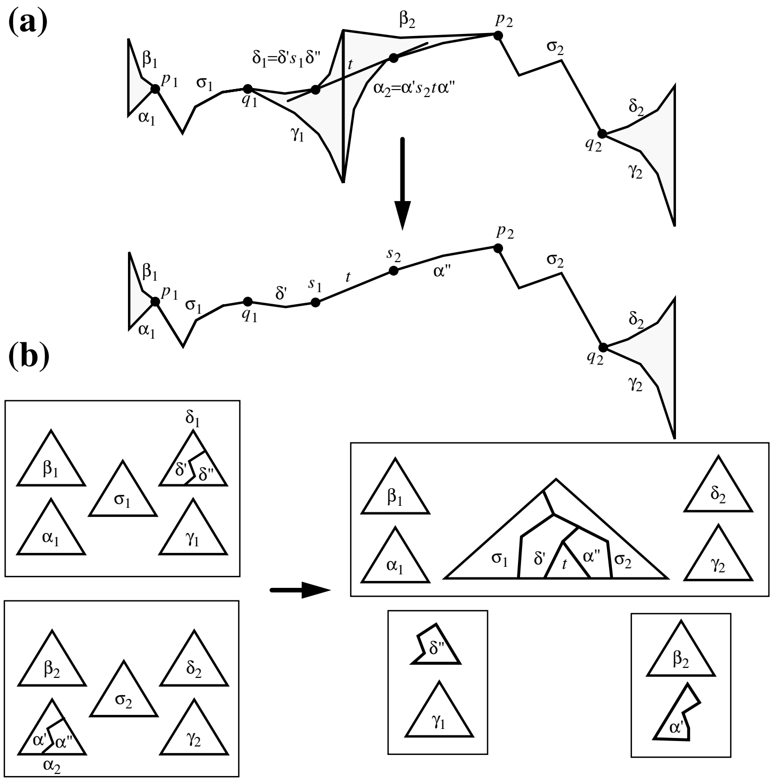

FIG. 9. *Example of update of the secondary structures in an elementary join of two solid paths.* **(a)** *Geometric construction of the hourglass.* **(b)** *Construction of the representation of the root hourglass by means of split and join operations on the chain-trees in the representation of the hourglasses of the children nodes.*

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. The lemma below follows from Lemmas 3.8 and 4.2.

LEMMA 4.3. *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 a new node $y(v)$ into $\mathcal{Y}$. Let μ be one of the two nodes in the *ltree* and *rtree* 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 a pruned tree. After the insertion of v, the fragment of e stored in μ is further partitioned into $O(\log n)$ fragments (but the total number of fragments of e is still $O(\log n)$) according to the subtree of $\mathcal{Y}$ rooted at y ($y(v)$ has already been inserted into this subtree); we allocate these edge fragments into a new tree T_μ, expand leaf μ to T_μ, establish a pointer from y to μ, and rename all fragments of e to e_1 or e_2 appropriately.

The insertion of $y(v)$ into $\mathcal{Y}$ may cause rebalancing operations in $\mathcal{Y}$ carried out by means of rotations. A rotation between a node y' and its child y'' implies that horizontal cuts at y'' now take priority over horizontal cuts at y'. It is easy to see that the rotation only affects the subtrees of the *ltree*'s and *rtree*'s rooted at the nodes pointed to by y'. We rebuild such subtrees from scratch, which can be done in time proportional to their size. Note that prior to the rotation, a leaf μ of *ltree* or *rtree* corresponding to y' stores an edge fragment that spans the canonical vertical interval I of y', and thus μ is not affected by the rotation (except that after the rotation we have to redirect the original pointer of μ to y' so that it now points to y'', since y'' now corresponds to I). Since there may be a large number of such leaves μ that do not require rebuilding, we do not establish a back pointer from y' to leaf μ in our data structure (as we have already seen), so that inefficient checking for the necessity of rebuilding is avoided. Also, the redirection of all pointers of leaves μ from y' to y'' can be done efficiently when we rotate y' with y'': we switch the contents of the *physical* nodes y' and y'' to interchange the roles of the physical nodes y' and y'' (and then carry out the rotation appropriately by $O(1)$ elementary splits and joins), so that all these pointers are effectively redirected, though no actual changes are made to the pointers. Now we show that the rebuilding of the subtrees of *ltree*'s and *rtree*'s caused by a rotation in $\mathcal{Y}$ can be performed efficiently.

LEMMA 4.4. *Let* y *be a node of* $\mathcal{Y}$ *whose subtree has* ℓ *leaves. The subtrees of ltree's and rtree's with the hull structure appended whose roots are pointed to by node* y *have total size* $O(\ell \log \ell)$, *and can be built in time* $O(\ell \log \ell)$.

Proof. The subtree of $\mathcal{Y}$ rooted at y has exactly $2\ell - 1$ 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 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 4.5. *The amortized rebalancing time of the Y-tree* $\mathcal{Y}$ *in a sequence of update operations is* $O(\log^2 n)$.

We conclude:

THEOREM 4.6. *Shortest path queries* PATHLENGTH(q_1, q_2, r) *and* PATH(q_1, q_2, r) *in an* n*-vertex connected 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 $O(\log^3 n)$ (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 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 $O((\ell + 1) \log^3 n)$ and $O((\ell + 1) \log^3 n + k)$, 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 the 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. 10):

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, we decompose τ by 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. 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. 10). Each node of $\mathcal{T}$ is associated with a trapezoid τ of the decomposition and the partitioning object (a tangent or a horizontal line) of τ, and stores the representation of such object. Nodes of $\mathcal{T}$ are classified into three categories (and the association): a $\bigcirc$-*node* (a vertical cut), a ∇-*node* (a horizontal cut), and a $\square$-*node* (a terminal trapezoid of the decomposition and its edge fragment).

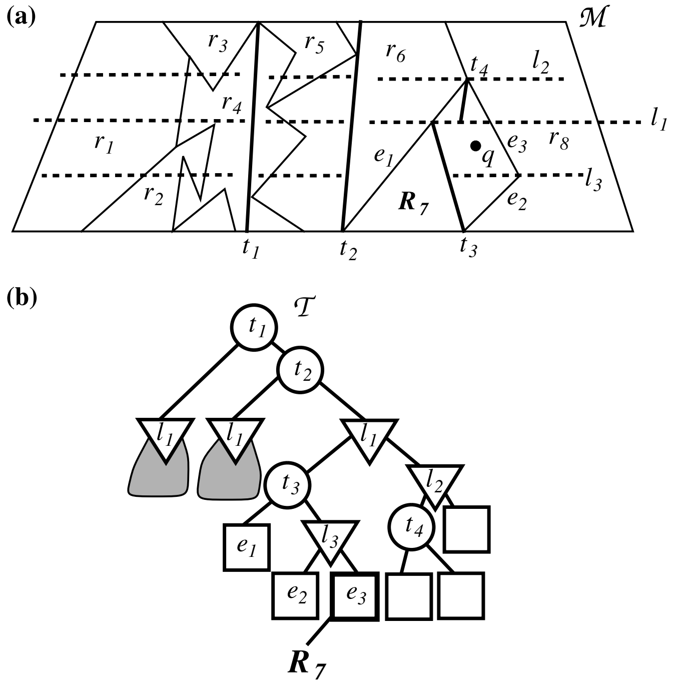

FIG. 10. *Example of the construction of trapezoid tree $\mathcal{T}$ for map $\mathcal{M}$.* (**a**) *Recursive decomposition of $\mathcal{M}$ by vertical and horizontal cuts.* (**b**) *Trapezoid tree $\mathcal{T}$ associated with the decomposition in part (a).*

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 [30], i.e., $\mathcal{T}$ is decomposed into solid paths (which should not be confused with the solid paths in the normalization structure), connected by dashed arcs (see Fig. 11). Each solid path is associated with a path tree, implemented as a biased search tree [3]. Note that the sequence of nodes of a solid path of $\mathcal{T}$ identifies a sequence of nested trapezoids. For example, in path tree $T(P_1)$ of Fig. 11(c), leaf t_1 identifies the trapezoid of the entire map $\mathcal{M}$, and leaf t_4 identifies the trapezoid whose right side is at infinity and whose other sides are t_2, l_1 and l_2. A point-location query starts at the root of the path tree of the topmost solid path of $\mathcal{T}$ (e.g., the root of $T(P_1)$ in Fig. 11(c)). 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 against 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 a solid path is stored bottom-to-top in the left-to-right leaves of its path tree). When we reach a leaf of a path tree (which represents a node μ of $\mathcal{T}$), we always exit on a dashed arc, 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 against the partitioning object of μ. For example, in Fig. 11(c), when we reach leaf l_1 of $T(P_1)$, we know that the next node to visit is the root of $T(P_2)$, since that is the only exit; when we reach leaf l_3 of $T(P_2)$, we discriminate q with l_3 and move down right to $T(P_5)$ by the fact that q is above l_3. By this process, we will finally reach a leaf of a path tree with no exit (representing a leaf of $\mathcal{T}$), which identifies an edge of the region containing q.

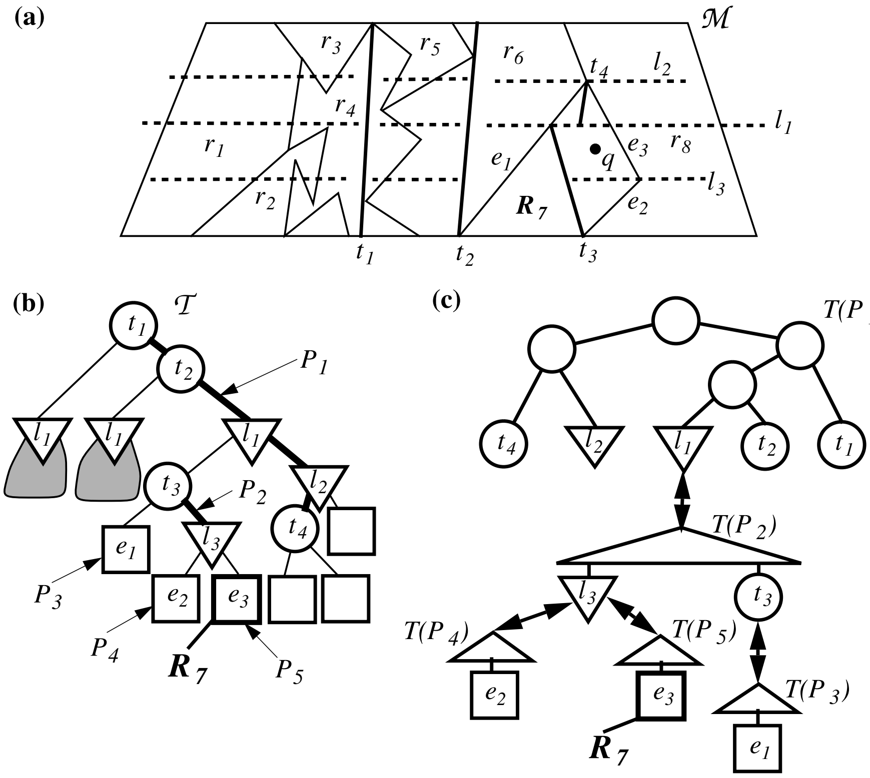

FIG. 11. *Representing trapezoid tree $\mathcal{T}$ by a dynamic tree.* (**a**) *The same decomposition of $\mathcal{M}$ as in Fig. 10(a).* (**b**) *Decomposing trapezoid tree $\mathcal{T}$ of Fig. 10(b) into solid paths $P_1, P_2, \cdots$.* (**c**) *Actual data structure representing $\mathcal{T}$, where $T(P_i)$ is the path tree for solid path P_i in $\mathcal{T}$. The left-to-right leaves of $T(P_i)$ represent bottom-to-top nodes of P_i, which in turn correspond to smaller-to-bigger nested trapezoids.*

Using biased search trees [3] as the standard implementation of path trees, we have

LEMMA 5.1. *The time complexity for a point location query is $O(\log n)$.*

Proof. Let $(\nu_1, \mu_1), \cdots, (\nu_\ell, \mu_\ell)$ be the sequence of dashed arcs 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 the path 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 ◯-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 □-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 thread 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. 12–13. 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 4.4, we can prove the following lemma.

LEMMA 5.2. *Let y be a node of $\mathcal{Y}$ whose subtree has ℓ leaves. Then the subtrees of $\mathcal{T}$ whose roots are pointed to by y have total size $O(\ell \log \ell)$, and can be built in time $O(\ell \log \ell)$.*

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

THEOREM 5.3. *Pont location queries* LOCATE(q) *in an n-vertex connected 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 $O(\log^3 n)$ (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.

We show that the dynamic point location data structure in the previous section also supports ray-shooting queries in worst-case time $O(\log^3 n)$. Without loss of

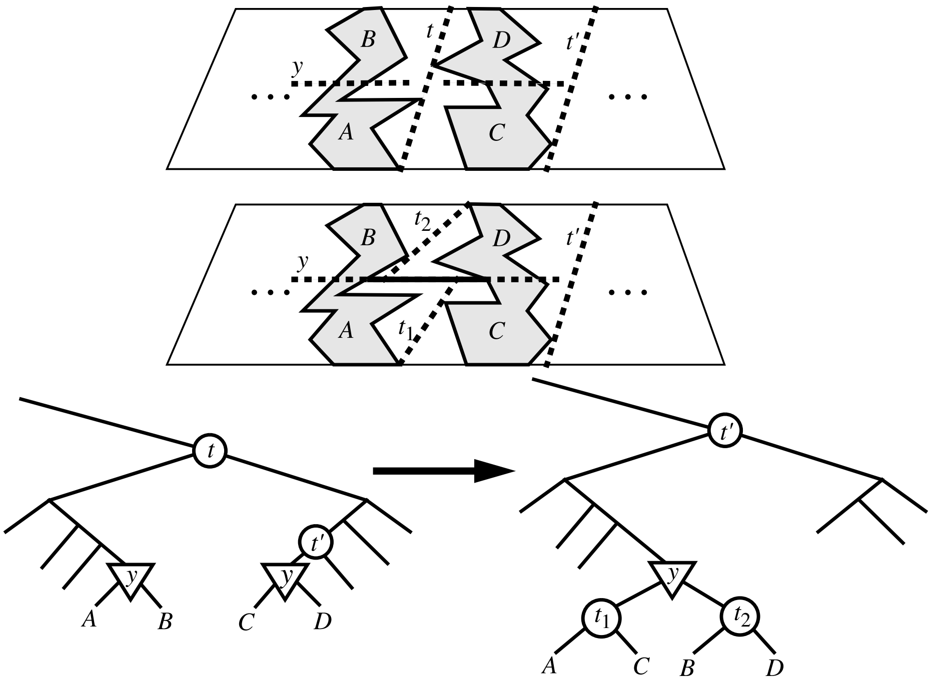

FIG. 12. *Update of the trapezoid tree in consequence of an elementary split of a monotone path in the normalization structure.*

generality, assume that (q, d) is oriented upwards. The ray-shooting algorithm is as follows:

First, we perform LOCATE(q) to determine the region r containing q. 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 LOCATE(q) also identifies the monotone sleeve SLEEVE(Q) of r containing q and the splitter s_1 of SLEEVE(Q) immediately below q. We find the first intersection q' of (q, d) with the boundary of 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 SLEEVE(Q)) we apply the algorithm recursively to the new ray (q', d).

We find the first intersection q' of (q, d) with the boundary of SLEEVE(Q) by the process below:

1. Find the topmost splitter s_2 in SLEEVE(Q) such that HOURGLASS(s_1, s_2) is open, by means of $O(\log n)$ elementary splits and joins of subpaths of Q that yield a new monotone path R such that SLEEVE$(s_1, s_2) =$ SLEEVE(R). Note that the boundary of SLEEVE(R) is part of the boundary of SLEEVE(Q) except for possibly s_1 and s_2, where s_2 is part of the boundary of SLEEVE(Q) if and only if s_2 is the top lid of SLEEVE(Q).
2. Find the first intersection p of (q, d) with the boundary of SLEEVE(R).
3. If p is not on s_2, or if p is on s_2 but s_2 is the top lid of SLEEVE(Q), then p is on the boundary of SLEEVE(Q) and thus the desired intersection q'. Return p and stop.
4. Else (p is on s_2 and s_2 is not the top lid of SLEEVE(Q)), set $s_1 := s_2$, $q := p$, and go to Step 1. Note that this situation can occur at most twice, since s_2 is the topmost splitter above s_1 such that HOURGLASS(s_1, s_2) is open, and any

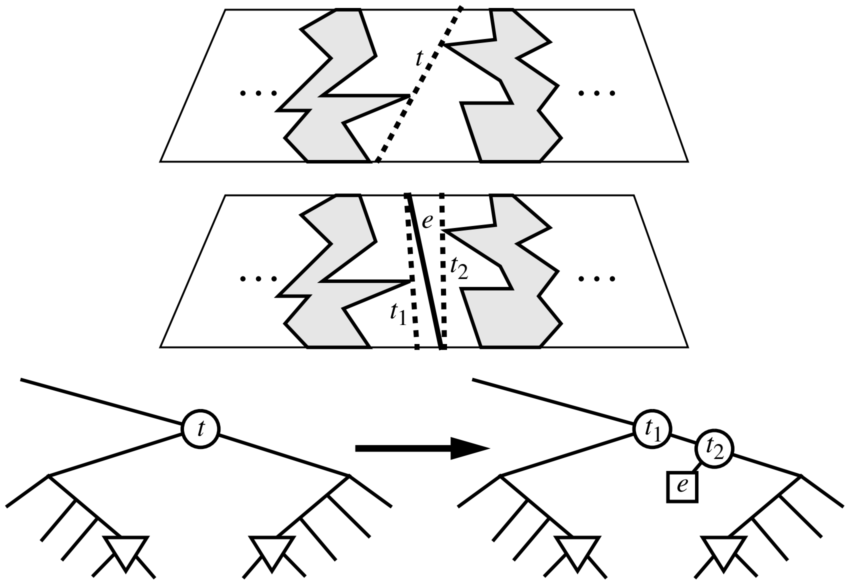

FIG. 13. *Update of the trapezoid tree caused by parting a monotone path in the normalization structure because of an edge insertion.*

straight line can completely go through at most one such hourglass, with the bottom and top portions of the line possibly in the two (below and above) adjacent hourglasses (see Fig. 14).

In step 2, the first intersection p of (q, d) with the boundary of 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'', where CHAIN(μ') is below CHAIN(μ''), we determine the intersection of (q, d) with the convex hull of CHAIN(μ). 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: we then compute the complete convex hulls of CHAIN(μ') and CHAIN(μ''), and repeat the process on μ' or on μ'' depending on whether or not (q, d) intersects with the convex hull of CHAIN(μ'), respectively.

The computation of point q' can be done in $O(\log^2 n)$ time: step 1 performs $O(\log n)$ elementary joins and splits of solid subpaths of Q, each in $O(\log n)$ time by Lemma 4.2; step 2 takes $O(\log^2 n)$ time, with $O(\log n)$ time on each node visited during the binary search; finally, the steps are executed at most three times by step 4. The number of recursive calls to compute a sequence of such points q' is $O(\log n)$ since the query ray intersects $O(\log n)$ lids by Corollary 3.3. 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 6.1. *Ray-shooting queries* SHOOT(q, d) *in an n-vertex connected planar map can be performed in worst-case time $O(\log^3 n)$ using a fully dynamic data structure that uses space $O(n \log n)$ and supports updates of the map in time $O(\log^3 n)$ (amortized for vertex updates).*

Theorem 6.1 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 $O((k + 1) \log^3 n)$.

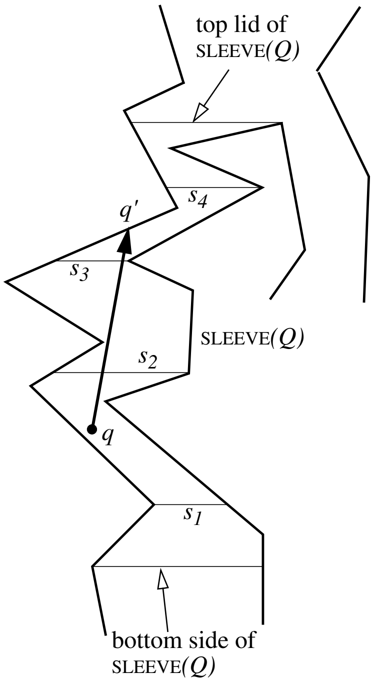

FIG. 14. *The situation in step 4 of the process for computing q' can occur at most twice. For $i = 1, 2, 3$, s_{i+1} is the topmost splitter above s_i such that* HOURGLASS(s_i, s_{i+1}) *is open. As shown, the situation of step 4 occurrs twice when (q, d) hits s_2 and s_3, respectively. Note that (q, d) can not reach s_4, or otherwise* HOURGLASS(s_2, s_4) *would be open and s_3 would not be the topmost splitter above s_2 such that* HOURGLASS(s_2, s_3) *is open.*

REFERENCES

[1] P. K. AGARWAL AND M. SHARIR, *Applications of a new partition scheme*, Discrete Comput. Geom., 9 (1993), pp. 11–38.

[2] H. BAUMGARTEN, H. JUNG, AND K. MEHLHORN, *Dynamic point location in general subdivisions*, in Proc. 3rd ACM-SIAM Sympos. on Discrete Algorithms, 1992, pp. 250–258.

[3] S. W. BENT, D. D. SLEATOR, AND R. E. TARJAN, *Biased search trees*, SIAM J. Comput., 14 (1985), pp. 545–568.

[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 Comput. Geom., 4 (1989), pp. 551–581.

[6] S. W. CHENG AND R. JANARDAN, *New results on dynamic planar point location*, SIAM J. Comput., 21 (1992), pp. 972–999.

[7] ———, *Space efficient ray shooting and intersection searching: algorithms, dynamizations, and applications*, in Proc. 2nd ACM-SIAM Sympos. on Discrete Algorithms, 1991, pp. 7–16.

[8] Y.-J. CHIANG AND R. TAMASSIA, *Dynamization of the trapezoid method for planar point location in monotone subdivisions*, Internat. J. Comput. Geom. Appl., 2 (1992), pp. 311–333.

[9] ———, *Dynamic algorithms in computational geometry*, Proc. IEEE, G. Toussaint, ed., 80 (1992), pp. 1412–1434.

[10] D. P. DOBKIN AND R. LIPTON, *Multidimensional searching problems*, SIAM J. Comput., 5 (1976), pp. 181–186.

[11] M. I. EDAHIRO, I. KOKUBO AND T. ASANO, *A new point-location algorithm and its practical efficiency—comparison with existing algorithms*, ACM Trans. Graphics, 3 (1984), pp. 86–109.

[12] H. EDELSBRUNNER, L. J. GUIBAS, AND J. STOLFI, *Optimal point location in a monotone*

subdivision, SIAM J. Comput., 15 (1986), pp. 317–340.

[13] O. Fries, *Zerlegung einer planaren unterteilung der ebene und ihre anwendungen*, M.S. thesis, Inst. Angew. Math. and Inform., Univ. Saarlandes, Saarbrücken, Germany, 1985.

[14] O. Fries, K. Mehlhorn, and S. Näher, *Dynamization of geometric data structures*, in Proc. 1st ACM Sympos. on Computational Geometry, 1985, pp. 168–176.

[15] M. T. Goodrich and R. Tamassia, *Dynamic trees and dynamic point location*, in Proc. 23rd ACM Sympos. on Theory of Computing, 1991, pp. 523–533.

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

[17] D. G. Kirkpatrick, *Optimal search in planar subdivisions*, SIAM J. Comput., 12 (1983), pp. 28–35.

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

[19] ———, *Euclidean shortest paths in the presence of rectilinear barriers*, Networks, 14 (1984), pp. 393–410.

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

[21] M. H. Overmars, *Range searching in a set of line segments*, in Proc. 1st ACM Sympos. on Computational Geometry, 1985, pp. 177–185.

[22] M. H. Overmars and J. van Leeuwen, *Maintenance of configurations in the plane*, J. Comput. Syst. Sci., 23 (1981), pp. 166–204.

[23] F. P. Preparata, *A new approach to planar point location*, SIAM J. Comput., 10 (1981), pp. 473-483.

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

[25] F. P. Preparata and R. Tamassia, *Fully dynamic point location in a monotone subdivision*, SIAM J. Comput., 18 (1989), pp. 811–830.

[26] ———, *Dynamic planar point location with optimal query time*, Theoret. Comput. Sci., 74 (1990), pp. 95–114.

[27] F. P. Preparata, J. Vitter, and M. Yvinec, *Computation of the axial view of a set of isothetic parallelepipeds*, ACM Trans. Graphics, 9 (1990), pp. 278–300.

[28] J. H. Reif and S. Sen, *An efficient output-sensitive hidden-surface removal algorithm and its parallelization*, in Proc. 4th ACM Sympos. on Computational Geometry, 1988, pp. 193–200.

[29] N. Sarnak and R. E. Tarjan, *Planar point location using persistent search trees*, Commun. ACM, 29 (1986), pp. 669–679.

[30] D. D. Sleator and R. E. Tarjan, *A data structure for dynamic trees*, J. Comput. Syst. Sci., 24 (1983), pp. 362–381.

[31] R. Tamassia, *An incremental reconstruction method for dynamic planar point location*, Information Processing Letters, 37 (1991), pp. 79–83.