Chapter 6
A Unified Approach to Dynamic Point Location, Ray Shooting, and Shortest Paths in Planar Maps*

Yi-Jen Chiang† Franco P. Preparata† 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).

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, 27] 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, 29].

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 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 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 in a dynamic setting. Sublinear-time techniques are known only for ray-shooting queries [1, 7], with $O(\sqrt{n}\,\text{polylog}(n))$ 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]). In the static case, a connected map can be reduced to monotone (or, *normal-*

*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, and by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014–91–J–4052, ARPA order 8225.

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

ized) 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 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: (a) 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. (b) 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. (c) A *location structure* that represents a recursive decomposition of the normalized map into trapezoidal regions, and supports point-location queries in optimal time.

The fundamental constituents of our data structures are monotone chains and trapezoids determined by edges and horizontal lines through vertices. The normalization structure provides the unifying framework for the three applications considered. In fact, this structure 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 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. It should be underscored that the elementary data structures employed are particularly simple, so that 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).*

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 $O(\sqrt{n}\,\text{polylog}(n))$ 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.

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. 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 $\mathcal{M}$ currently being considered.

The *trapezoidal decomposition* of a connected 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*. 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, 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 r and paths in $\delta(r)$.) 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, in particular biased binary trees [3] and $BB[\alpha]$-trees [20]. 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.

3 Dynamics of Trapezoidal Decompositions

Given a connected map $\mathcal{M}$, our objective is to efficiently maintain a normalization (refinement into a monotone map) of $\mathcal{M}$ under a dynamic regimen of insertions/deletions of vertices and edges.

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 r.

We imagine to represent $\delta(r)$ as a dynamic tree $\Delta(r)$ [28] (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 may 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).

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

1. All light arcs (Rule 1).

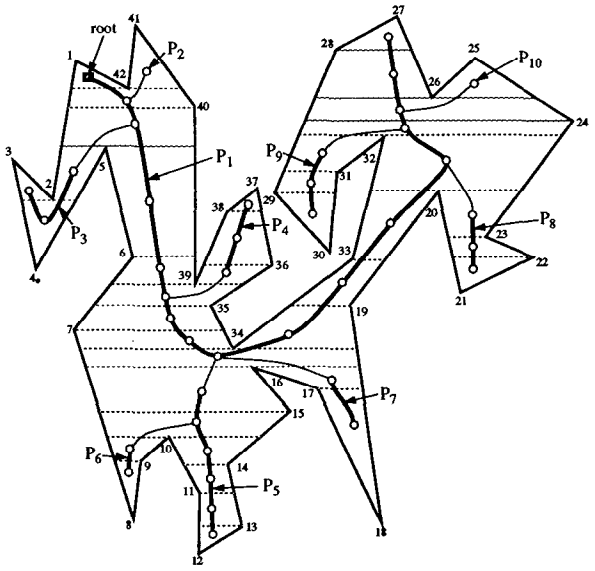

Figure 1: Example of a region r and its dynamic tree $\Delta(r)$ (P_1,...,P_{10} are solid paths).

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

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

LEMMA 3.2. *Each directed path of the dynamic tree $\Delta(r)$ contains at most $\log_2 m$ light arcs.*

COROLLARY 3.1. *Altering the weight of a node of $\Delta(r)$ can cause at most $2\log_2 m$ arcs to change their classification.*

Since insertion or deletion of an edge may substantially modify the set of trapezoids but alters only slightly the structure of region boundaries, we adopt a data structure, called *double-thread data structure*, to represent a solid path P of $\Delta(r)$ by two "threads", which respectively correspond to two chains whose union is the boundary of the polygon associated with P.

Each arc α of $\Delta(r)$ intersects its dual splitter issuing from some vertex v of r; we associate α with v. By associating each node of $\Delta(r)$ to the arc issuing from it, we may represent $\Delta(r)$ equally well by the arcs instead of nodes of $\Delta(r)$, where we maintain the arcs by the corresponding vertices. Recall that $\Delta(r)$ is represented by a collection of solid paths connected via dashed arcs; if a solid path P is followed by a dashed arc α, we also include α as an arc on P in our representation. Each vertex v associated with P is classified as follows: walking along P toward the root, v is classified *left* if it lies to the left of P, and *right* otherwise.

We represent a solid path P with two *path trees* *lthread*(P) and *rthread*(P), each implemented as a two-

level balanced binary tree. Note that the arcs of P can be partitioned into maximal monotone (with respect to the y-axis) subpaths $Q_1, Q_2, \cdots, Q_k$. 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. Path 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 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*. We also establish a bidirectional link between the roots of $lthread(P)$ and $rthread(P)$. An example is shown in Fig. 2.

Any node on P might be pointed to (via dashed arcs) by some other solid paths. Suppose that P' points to P via an 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 $P = P_1$), 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. 2). 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 $\Delta(r)$, P' points to some node μ of P where μ joins two monotone subpaths Q_i and Q_{i+1}; we make μ correspond to the coupler H_i and let both roots of $lthread(P')$ and $rthread(P')$ point to H_i of $lthread(P)$ or of $rthread(P)$ according to whether P' is to the left or right of P (see Fig. 2).

Now we define the weight of the 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* or *rtree*), then the weight of λ is the sum of the weights of the leaves in T, otherwise leaf λ is a coupler H_i and its weight is the sum of the weights of the path trees pointing to λ (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 $I_v = [y', y'']$ 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: we walk from v to the root of $ltree(Q_i)$, follow the pointer to the root of $rtree(Q_i)$, and perform a binary search in $rtree(Q_i)$ to locate I_v by y-coordinates. The same approach applies when v is stored in $rtree(Q_i)$. Clearly, this operation can be carried out in logarithmic time.

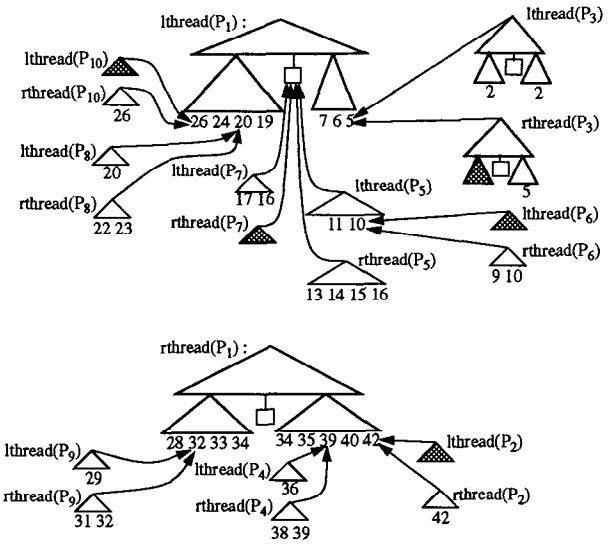

Figure 2: Double-thread data structures for the solid paths in Fig. 1.

In summary, the weight $w(\mu)$ of node μ in $\Delta(r)$ 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.

LEMMA 3.3. *The space complexity of the normalization structure for an n-vertex map is $O(n)$.*

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.

THEOREM 3.1. *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 *}.*

ATTACHVERTEX and DETACHVERTEX can be simulated by a sequence of $O(1)$ operations taken among the first four of the repertory and a point-location query. In the rest of this section, we describe how to implement the first four operations of the repertory.

We begin by considering some elementary dynamic tree operations *expose, conceal* and *evert* introduced in [28], in terms of which the operations of the above repertory can be immediately expressed. The operation of moving the root of $\Delta(r)$ to any arbitrary node is referred to as *evert*. 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.

LEMMA 3.4. *The update of the double-thread data structure as required by the operations expose, conceal and evert can each be performed 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 3.5. *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.*

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

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 3.5 (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).
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).

Step 3 takes $O(\log m)$ time, and all the other steps globally involve a fixed number of *evert*, *expose* and *conceal* operations, so that INSERTEDGE takes $O(\log^2 m)$ time. Operation REMOVEEDGE is the inverse operation of INSERTEDGE, and can be done in $O(\log^2 m)$ time.

THEOREM 3.2. *Each of the operations* INSERTEDGE, REMOVEEDGE, INSERTVERTEX, REMOVEVERTEX *takes time $O(\log^2 m)$ to update the normalization structure.*

4 Shortest Path Queries

In this section we illustrate how the normalization data structure can be augmented with a *hull structure* to answer the following queries:

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

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

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 paths PATH(q_1, q_2) with $q_1 \in s_1$ and $q_2 \in s_2$ (see Fig. 3.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 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 of α and β do not intersect, then PATH(a_1, a_2) is the convex hull of the subchain of α clockwise 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, p_1') and of the convex hull 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) 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. 3.a).

If an hourglass is represented by the length of the string and the (two to four) convex chains forming the rest of its boundary, 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 $O(1)$ 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.

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].

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[α]-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 structure so that hourglasses can be dynamically maintained (see Fig. 3). 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 $\mathcal{Y}$. 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 SLEEVE(μ) the sleeve of Q', with s_1 and s_2 the splitters

that delimit SLEEVE(μ), 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:

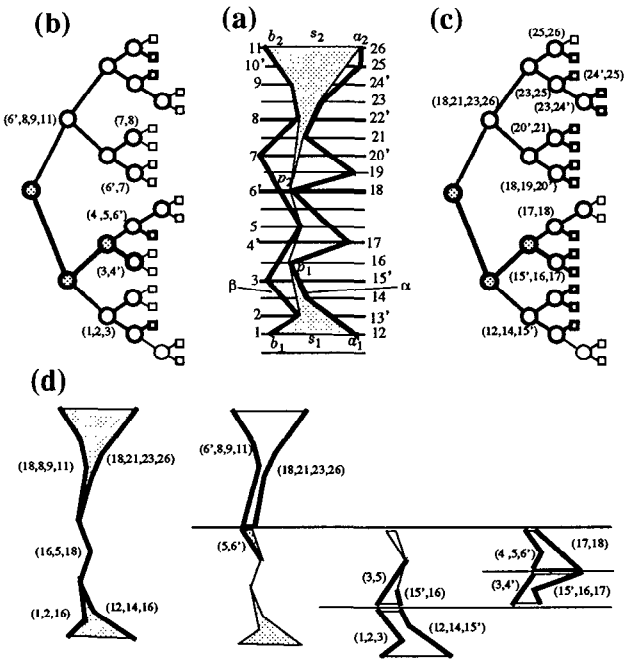

Figure 3: Example of representation of hourglasses in the nodes of *ltree*(Q) and *rtree*(Q) of a monotone path Q. **(a)** The sleeve of Q: the parallel lines drawn on it represent set Y; the points on the sleeve with labels of the type i' delimit fragments of the same edge; the hourglass between the extreme splitters of the sleeve is shown grey-filled. **(b)** Pruned-tree *ltree*(Q): the nodes of *ltree*(Q) are 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.

- 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 the HOURGLASS(s_1, s_2).
- If HOURGLASS(μ) is closed and μ is not the root, then μ stores the apices and the length of the string of HOURGLASS(s_1, s_2), plus the subchains of the funnels of HOURGLASS(s_1, s_2) that are not stored at the ancestors of μ.

Trees *lthread*(P) and *rthread*(P) are isomorphic. Also, an internal node μ in the higher level of *lthread*(P) stores the length and endpoints of the string of HOURGLASS(s_1, s_2), and similarly for *rthread*(P).

LEMMA 4.1. *The space requirement of the hull structure is* $O(n \log n)$.

Queries 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) 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 Fig. 4):

LEMMA 4.2. *An elementary split or join of two solid path trees in the normalization structure augmented with the hull structure takes time* $O(\log n)$.

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 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.*

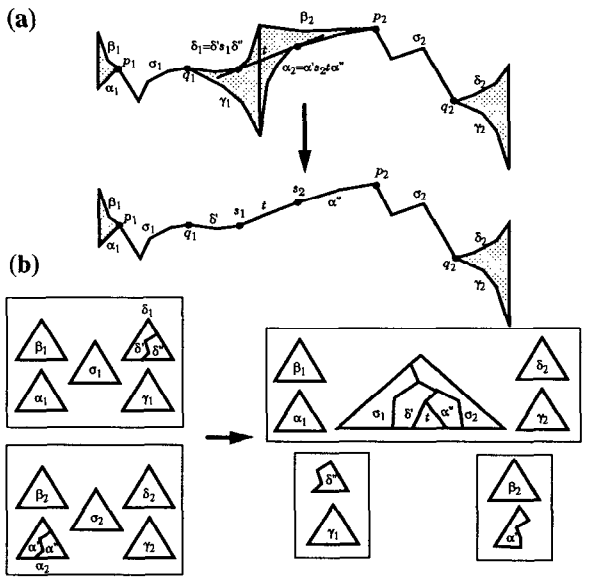

Figure 4: Example of update of the secondary structures in an elementary join of two solid paths. **(a)** Geometric construction of the hourglass. **(b)** Construction of the representation of the root hourglass by means of split and join operations on the chain-trees in the representation of the hourglasses of the children nodes.

Now, we discuss operation INSERTVERTEX$(v, e; e_1, e_2)$. 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 $\mathcal{Y}$ 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'. The rotation only affects the subtrees of the path trees rooted at the nodes pointed by y', which can be rebuilt from scratch in time proportional to their size.

LEMMA 4.4. *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 $O(\ell \log \ell)$.*

By the properties of BB[α]-trees, we have that the amortized rebalancing time of the *Y-tree* $\mathcal{Y}$ in a sequence of update operations is $O(\log^2 n)$.

THEOREM 4.1. *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).*

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 a 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 called *trapezoid tree*.

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. 5):

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. 5). Each node of $\mathcal{T}$ is associated with a trapezoid τ of the decomposition and stores the partitioning object of τ. Nodes of $\mathcal{T}$ are classified into three categories: $\bigcirc$-*node* (vertical cut), ∇-*node* (horizontal cut), and $\square$-*node* (terminal trapezoid of the decomposition).

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 $\mathcal{T}$ has $O(n \log n)$ leaves. Since each node stores a constant amount of information, $\mathcal{T}$ uses $O(n \log n)$ space.

Instead of imposing a balance requirement on $\mathcal{T}$, we implement $\mathcal{T}$ as a dynamic tree [28], i.e., $\mathcal{T}$ is decomposed into solid paths (which should not be

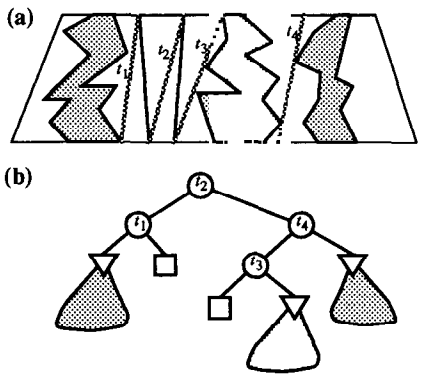

Figure 5: **(a)** Decomposition of a trapezoid and **(b)** associated trapezoid tree.

confused with the solid paths in the normalization structure), connected by dashed arcs. Each solid path is associated with a path-tree, implemented as a biased search tree [3]. 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. 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 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 against the partitioning object of μ. By this process, we will finally reach a leaf of $\mathcal{T}$ which identifies an edge of the region containing q.

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

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

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 *ltree*(Q) and *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 *ltree*(R) and *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.

Regarding updates, 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. 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 have the following lemma.

LEMMA 5.2. *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 $O(\log^2 n)$.

THEOREM 5.1. *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).*

6 Ray Shooting

In this section we consider ray-shooting queries:

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

Without loss of 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. 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) as follows:

1. Find the topmost splitter s_2 of 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).
2. Find the first intersection p of (q, d) with SLEEVE(R).

3. If p is not on s_2, return p and stop; otherwise (p is on s_2) there are two subcases: if s_2 is the top lid of SLEEVE(Q), then return p and stop. Else, set $s_1 := s_2$, $q := p$, and repeat the process. Note that this situation can occur at most twice, since s_2 is the topmost splitter such that HOURGLASS(s_1, s_2) is open.

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.

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 supports updates of the map in time $O(\log^3 n)$ (worst-case for edge updates, and amortized for vertex updates).*

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*, 250–258, 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," *International J. of Computational Geometry and Applications*, Vol. 2, No. 3, 311–333, 1992. (Prelim. version: *Proc. 7th ACM Symposium on Computational Geometry*, 61–70, 1991.)

[9] Y.-J. Chiang and R. Tamassia, "Dynamic Algorithms in Computational Geometry," *Proceedings of the IEEE*, Vol. 80, No. 9, 1992.

[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—Comparison 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, Saarbrücken, 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 J. 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. Computer Systems Sciences*, 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] Sarnak, N. and R.E. Tarjan, "Planar Point Location Using Persistent Search Trees," *Communications ACM*, Vol. 29, 669–679, 1986.

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

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