Dynamic Planar Point Location with Optimal Query Time

Roberto Tamassia and Franco P. Preparata

Technical Report No. CS-89-24

March 1989

Dynamic Planar Point Location with Optimal Query Time *

Franco P. Preparata

Coordinated Science Laboratory
University of Illinois
1101. W. Springfield Avenue
Urbana, IL 61801

Roberto Tamassia

Department of Computer Science
Brown University
Box 1910
Providence, RI 02912

Abstract

We present a new dynamic technique for locating a point in a convex planar subdivision whose n vertices lie on a fixed set of N horizontal lines. The supported update operations are insertion/deletion of vertices and edges, and (horizontal) translation of vertices. Our method achieves query time $O(\log n + \log N)$, space $O(N + n \log N)$, and insertion/deletion time $O(\log n \log N)$. Hence, for $N = O(n)$, the query time is $O(\log n)$, which is optimal. The proposed technique, based on the *trapezoid method*, provides an efficient solution to many significant applications where the most frequent operation is the point location query, while updates are more rarely executed.

(March 1989)

* This work was carried out at the University of Illinois and was supported in part by National Science Foundation Grant ECS-84-10902, the Joint Services Electronics Program under Contract N00014-84-C-0149, and a grant from Cadre Technologies, Inc..

1. Introduction

The location of a point in a planar subdivision is formulated as follows: Given a planar subdivision $\mathcal{P}$ with n vertices, determine to which region of $\mathcal{P}$ a query point q belongs. If the query point q is on some vertex/edge of $\mathcal{P}$, then that vertex/edge is reported.

Most of the past research on planar point location [2-4, 7-9, 12, 14, 15] has been targeted to the *static* case, where the subdivision is fixed, and point location queries have to be answered on-line. Recently, however, increasing attention has been devoted to *dynamic* point location. Overmars [11] proposed a technique for the case where the n vertices of $\mathcal{P}$ lie on a fixed set of N horizontal lines, the boundary of each region has a bounded number of edges, and only edges can be inserted or deleted. This approach yields $O(N+n\log N)$ space requirement, and $O(\log n \log N)$ query and update times, and can be extended to support insertions and deletions of vertices in $O(\log n \log N)$ amortized time at the expense of deploying a rather complicated and not very practical data structure. The dynamic point location technique of Fries and Mehlhorn [5, 6, 10] allows both edge and vertex updates in general subdivisions. Their approach achieves $O(n)$ space, $O(\log^2 n)$ query time, and $O(\log^4 n)$ amortized update time. If only insertions are considered, the update time is reduced to $O(\log^2 n)$ [10, pp. 135-143]. The authors [13, 16] have recently presented a fully dynamic technique for the case of monotone subdivisions based on the chain-method of Lee-Preparata [8], with the following performance: the data structure uses space $O(n)$; the query time is $O(\log^2 n)$; the times for insertion/deletion are $O(\log n)$ for a vertex and $O(\log^2 n + k)$ for a chain with k edges, all worst-case.

In this paper we present a new dynamic point location technique for the case of convex subdivisions whose vertices lie on a fixed set of N horizontal lines (as in Overmars' method). The supported update operations are insertion/deletion of vertices and edges, and (horizontal) translation of vertices. Our method achieves query time $O(\log n + \log N)$, space $O(N+n\log N)$, and insertion/deletion time $O(\log n \log N)$. Hence, whenever $N = O(n)$, the query time is $O(\log n)$, which is optimal. In addition to its good theoretical performance, our technique is also easy to implement and very efficient in practice. It is based on the *trapezoid method* of Preparata [12], which has been experimentally shown to be one of the fastest point location methods among those with asymptotically optimal query time [3]. Our technique is best suited for the many significant applications where the most frequent operation is the point

location query, while updates are more rarely executed.

The rest of this paper is organized as follows. In Section 2 we formally state the problem. Section 3 describes the geometric foundations of the technique. Sections 4 and 5 present the dynamic data structure and the query algorithm. The update algorithms are discussed in Sections 6 and 7.

2. Preliminaries

A *monotone polygon* is a polygon r such that its boundary is partitionable into two monotone chains γ_1 and γ_2, called the *left chain* and *right chain* of r, respectively. Chains γ_1 and γ_2 share two vertices, referred to as $HIGH(r)$ and $LOW(r)$, respectively, with $y(HIGH(r)) > y(LOW(r))$. A *convex polygon* is a monotone polygon r such that, for any two vertices u and v of r, the segment (u,v) lies entirely inside r.

A *planar subdivision* $\mathcal{P}$ is a partition of the plane into polygons, called the *regions* of $\mathcal{P}$. A *monotone subdivision* is a planar subdivision whose regions are monotone polygons. Monotone subdivisions form an important class of planar subdivisions, since they include *convex subdivisions*, whose regions are convex polygons, and *triangulations*, whose regions are triangles.

Let $L = \{l_0, l_1, \cdots, l_N\}$ be a set of horizontal lines, in this order from bottom to top. The lines of L partition the plane into horizontal strips, called *elementary slabs*, two of which are (the bottom and top) half-planes. A *slab* is either an elementary slab or the union of two contiguous slabs. We consider a convex subdivision $\mathcal{P}$ whose n vertices are on the lines of L (see Fig. 1). This implies that all finite edges of $\mathcal{P}$ occur in the plane strip between l_0 and l_N, and that all infinite edges are horizontal rays.

The following update operations on $\mathcal{P}$ are defined:

INSERTPOINT $(v, e; e_1, e_2)$:

Split the edge $e = (u,w)$ into two edges $e_1 = (u,v)$ and $e_2 = (v,w)$, by inserting vertex v, which lies on some line of L.

REMOVEPOINT $(v, e_1, e_2; e)$:

Let v be a vertex of degree 2 whose incident edges, $e_1 = (u,v)$ and $e_2 = (v,w)$, are on the same straight line. Remove v and replace e_1 and e_2 with edge $e = (u,w)$.

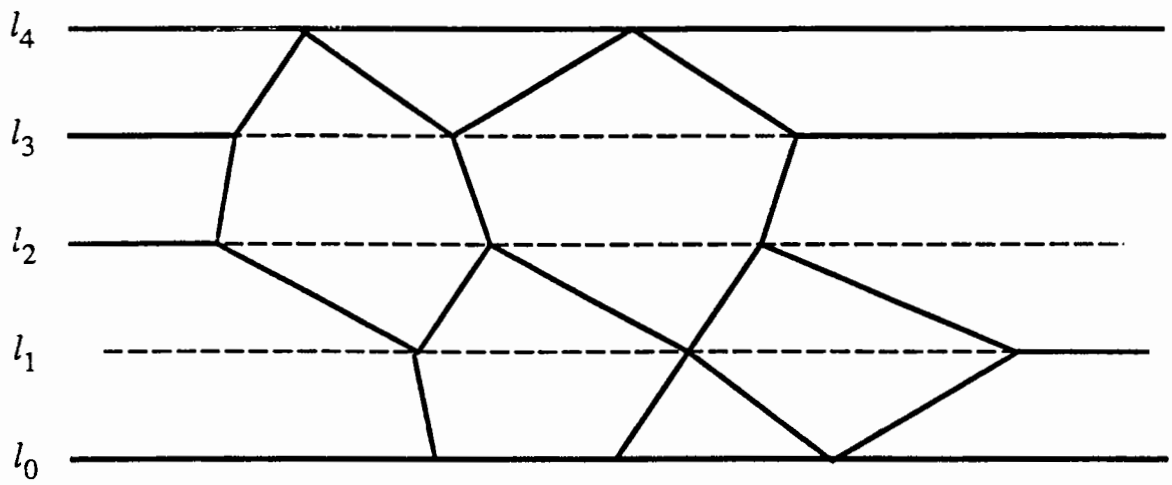


Figure 1 Example of convex subdivision with vertices on the set L of horizontal lines.

INSERTSEGMENT $(e, v_1, v_2, r; r_1, r_2)$:

Add the edge $e = (v_1, v_2)$, with $y(v_1) \leq y(v_2)$, to $\mathcal{P}$ inside region r of $\mathcal{P}$, which is decomposed into regions r_1 and r_2, with r_1 and r_2, respectively, to the left and to the right of e, directed from v_1 to v_2.

REMOVESEGMENT $(e, v_1, v_2, r_1, r_2; r)$:

Remove edge e and merge the regions r_1 and r_2 formerly on the two sides of e into region r. [The operation is allowed only if the subdivision $\mathcal{P}'$ so obtained is convex.]

MOVEPOINT $(v; x)$:

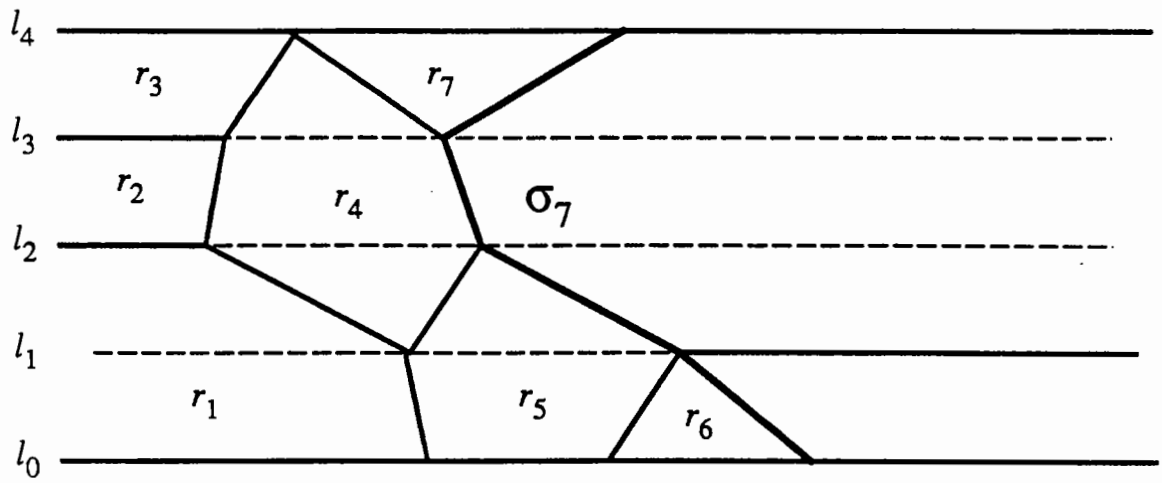
Horizontally translate a vertex v from its present location to abscissa x. [The operation is allowed only if the subdivision $\mathcal{P}'$ so obtained is convex and topologically equivalent to $\mathcal{P}$.]

The following theorem demonstrates the adequacy of the above repertory:

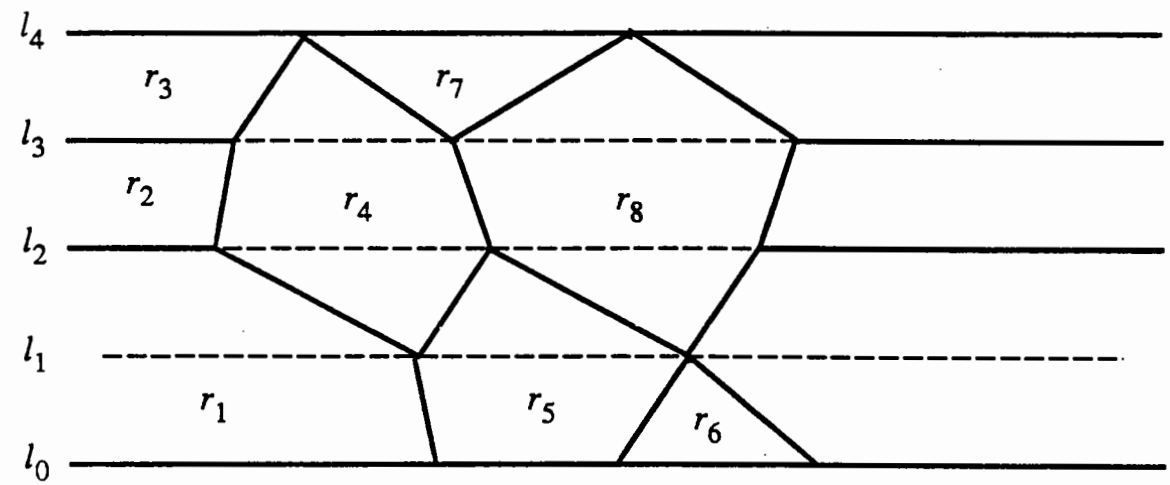
Theorem 1 An arbitrary convex subdivision $\mathcal{P}$ with n vertices can be assembled starting from the empty subdivision, and disassembled to obtain the empty subdivision, by a sequence of $O(n)$ operations from the above repertory.

Proof: It suffices to constructively demonstrate the assembly of $\mathcal{P}$, since its disassembly is obtainable by replacing each operation with its inverse and by reversing their order. Given regions r_1 and r_2 in $\mathcal{P}$, r_1 is said to be *left-adjacent to* r_2 ($r_1 \ll r_2$) if their boundaries share an edge e, and r_1 and r_2 are respectively to the left and to the right of e. (If r_1 and r_2 share a horizontal edge, then $r_1 \ll r_2$ if r_1 is below e.) Clearly, relation $\ll$ is acyclic. Consider an order $(r_1, r_2, \cdots, r_f)$ of the regions of $\mathcal{P}$ which is consistent with "$\ll$". We build $\mathcal{P}$ by adding its regions one by one, according to the chosen order. At the generic step of the construction, after adding region r_i we call $\mathcal{P}_i$ the resulting subdivision. (Obviously, $\mathcal{P}_f = \mathcal{P}$.) In $\mathcal{P}_i$ there is a monotone polygonal line σ_i, which forms the boundary of the set of regions $\{r_1, r_2, \cdots, r_i\}$ (refer to Fig. 2). Consider the (external) angles to the right of σ_i at each nonextreme vertex of σ_i: each angle $> \pi$ is bisected by a horizontal ray to ensure the property of convexity for $\mathcal{P}_i$. Consider now region r_{i+1}: obviously, the left chain of r_{i+1} is already present in $\mathcal{P}_i$, and let the right chain of r_{i+1} be $v'_1, \cdots, v'_s$. By means of *INSERTSEGMENT* we insert the edge $e = (LOW\ (r_{i+1}), HIGH\ (r_{i+1}))$. (This operation is performed as many times as there are regions.) Next, with the sequence *INSERTPOINT*, *INSERTSEGMENT*, *MOVEPOINT* we insert vertex v'_2 and edge (v'_1, v'_2) into their final position, and partition the region to the right by a horizontal ray to preserve convexity. This process is performed for $v'_3, \cdots, v'_{s-1}$. (This sequence of operations is performed at most once for each vertex of $\mathcal{P}$.) Finally, by *REMOVESEGMENT* we remove any horizontal ray originating at v'_1 and v'_s if the corresponding external angles have become $\leq \pi$. (This operation is performed at most twice per region.) The resulting subdivision is $\mathcal{P}_{i+1}$ and exhibits the same invariant as regards the region to the right of σ_{i+1}. The comments about the frequency of each executed operation complete the proof. □

Let r be a convex polygon. Polygon r is said to be *1-supported* if it has an edge e such that any horizontal line intersecting r intersects also e, i.e., r contains the edge $(LOW\ (r), HIGH\ (r))$ (see Fig. 3(a)). Polygon r is said *2-supported* if it has two edges, e_1 and e_2, such that any horizontal line intersecting r intersects at least one of e_1 and e_2 (see Fig. 3(b)). (Notice that a 1-supported polygon is also 2-supported.) A convex subdivision is said to be 1-supported (respectively 2-supported) if all its regions are 1-supported (respectively 2-supported).



(a)



(b)

Figure 2 Example for the proof of Theorem 1. **(a)** Subdivision $\mathcal{P}_7$. **(b)** Subdivision $\mathcal{P}_8$.

An important property of i-supportedness ($i = 1,2$) is expressed by the following straightforward lemma:

Lemma 1 Let convex polygon r be cut by a horizontal line into two convex polygons r_1 and r_2. We have:

(i) If r is 1-supported, then both r_1 and r_2 are 1-supported.

(ii) If r is 2-supported, then at least one of r_1 and r_2 is 1-supported.

In the description of the data structure for dynamic point location we will assume that the subdivision $\mathcal{P}$ is 2-supported. As shown in the following, this is not restrictive. Let $\mathcal{P}^*$ be the subdivision obtained from $\mathcal{P}$ by adding to each region r which is not 1-supported the edge from *LOW* (r) to *HIGH* (r). (Notice that $\mathcal{P}$ may have degenerate regions of measure zero.) Clearly $\mathcal{P}^*$ is 1-supported. The answer to a point location query in $\mathcal{P}^*$ can be immediately converted to an answer in $\mathcal{P}$. Regarding updates, we have

Lemma 2 For each update operation performed on $\mathcal{P}$ there exists a sequence of at most four operations which update $\mathcal{P}^*$ so that after each intermediate operation the resulting subdivision is 2-supported (possibly 1-supported).

Proof: Consider operation *INSERTSEGMENT* ($e, v_1, v_2, r; r_1, r_2$) performed on $\mathcal{P}$. The sequence of operations in $\mathcal{P}^*$ is as follows (see Fig. 4):

(1) insert edge $s_1 = (LOW\ (r_1), HIGH\ (r_1))$;

(2) insert edge $s_2 = (LOW\ (r_2), HIGH\ (r_2))$;

(3) remove edge $s = (LOW\ (r), HIGH\ (r))$, unless $s = s_1$ or $s = s_2$;

(4) insert edge e.

(We assume that an insertion is not performed if the edge to be inserted already exists.) Similar arguments hold for operations *REMOVESEGMENT*.

Regarding operation *INSERTPOINT* ($v, e; e_1, e_2$), let r_1 and r_2 be the regions to the left and right of r, respectively. The sequence of operations in $\mathcal{P}^*$ is as follows:

(1) insert vertex v;

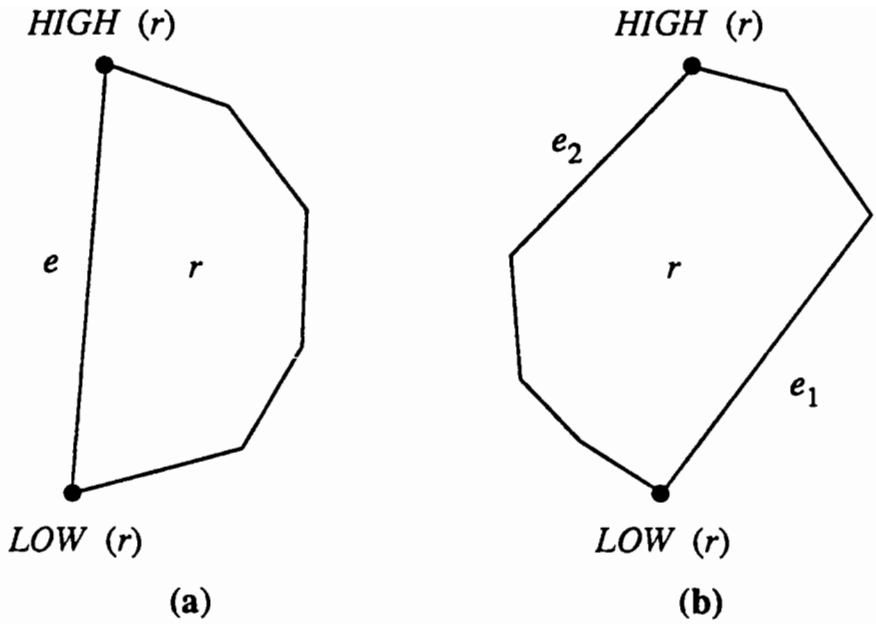


Figure 3 (a) A 1-supported polygon. (b) A 2-supported polygon.

(2) insert edge $s_1=(LOW\ (r_1),HIGH\ (r_1))$;

(3) insert edge $s_2=(LOW\ (r_2),HIGH\ (r_2))$.

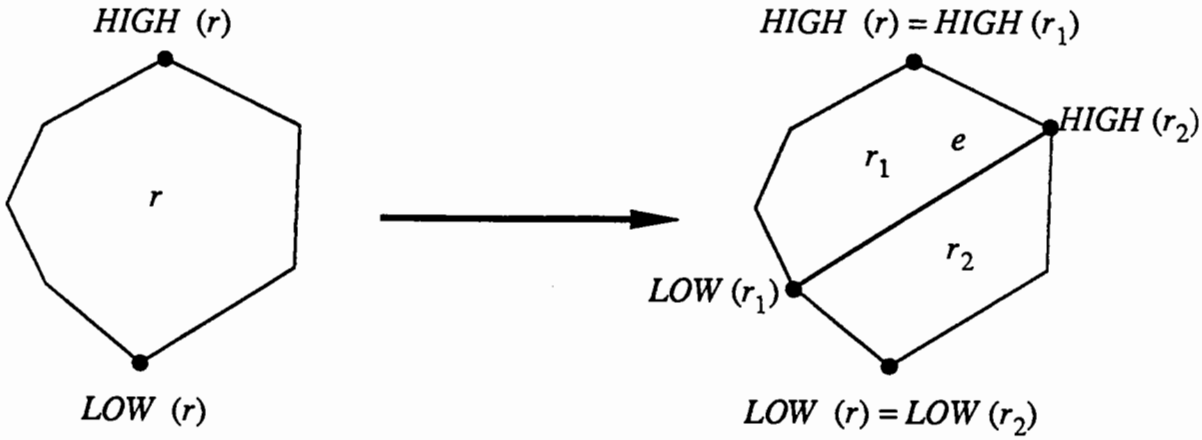
Again, the case of operation *REMOVEPOINT* is similar.

Finally, operation *MOVEPOINT* does not affect the supportedness of any region of $\mathcal{P}$, so that no further operation is required in $\mathcal{P}^*$. □

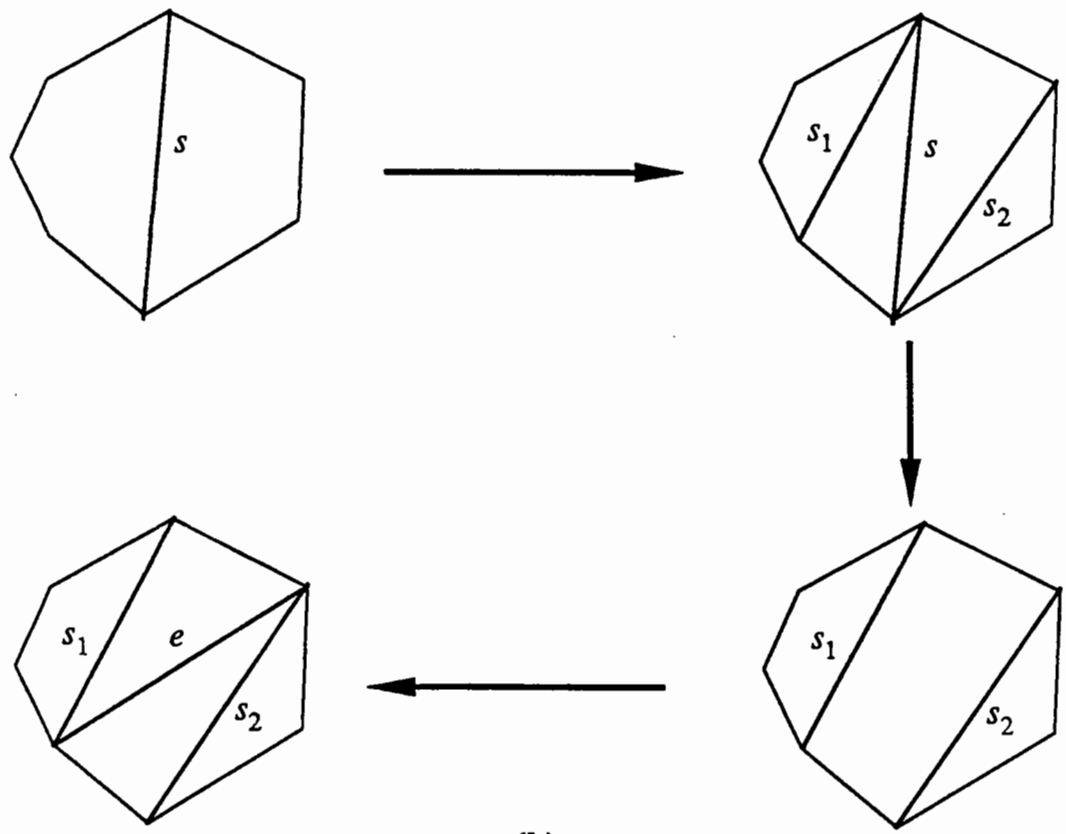
3. Trapezoids

Throughout this section we denote by $\mathcal{P}$ a 2-supported convex subdivision whose n vertices are on the lines of L. We show that the *trapezoid* method for point location [12] can be modified to perform efficiently updates on $\mathcal{P}$.

The trapezoid method is based on a recursive decomposition of the subdivision $\mathcal{P}$ into components, called trapezoids. A *trapezoid* τ is a convex subdivision whose external boundary



(a)



(b)

Figure 4 Example for the proof of Lemma 2. **(a)** Operation *INSERTSEGMENT* on $\mathcal{P}$. **(b)** Sequence of operations on $\mathcal{P}^*$.

is a quadrilateral with two (not necessarily bounded) horizontal sides, which belong to the lines of L. We denote the four sides of τ by *LEFT* (τ), *RIGHT* (τ), *BOT* (τ), and *TOP* (τ). Let *BOT* (τ) be on line l_i, and *TOP* (τ) on l_j. The *median line* of τ, denoted *MEDIAN* (τ), is the line l_k with $k = \lfloor (i+j)/2 \rfloor$. The subdivision $\mathcal{P}$ is the trapezoid with *BOT* ($\mathcal{P}$)=l_0, *TOP* ($\mathcal{P}$)=l_N, and having *LEFT* ($\mathcal{P}$) and *RIGHT* ($\mathcal{P}$) at infinity.

A *spanning edge* of trapezoid τ is an edge of τ with endpoints on the top and bottom sides of τ. A spanning edge s partitions trapezoid τ into two trapezoids, denoted τ_L and τ_R, with s = *RIGHT* (τ_L) = *LEFT* (τ_R). This is called a *vertical cut* (see Fig. 5(a)). If trapezoid τ does not have spanning edges, we decompose it by means of a horizontal line through its median line, and obtain trapezoids τ_B and τ_T, with *MEDIAN* (τ) = *TOP* (τ_B) = *BOT* (τ_T). This is called a *horizontal cut* (see Fig. 5(b)). A trapezoid can always be decomposed unless it is empty. We assume that a vertical cut takes precedence over a horizontal cut, so that the decomposition process is uniquely determined to within the specification of the sequence of consecutive vertical cuts of the same trapezoid.

The above decomposition process can be conveniently represented by means of a string notation. Namely, a vertical cut can be represented by the transformation $\tau \rightarrow \tau_L\ s\ \tau_R$, where s is the spanning edge; and a horizontal cut by the transformation $\tau \rightarrow (\tau_B)\ l\ (\tau_T)$, where l is the median line of τ. Finally, an empty trapezoid τ is eliminated using the transformation $\tau \rightarrow \varepsilon$, where ε is the null string. Hence, the recursive decomposition of a trapezoid τ can be viewed as a sequence of rewritings using one of the above rules. In this framework, trapezoids correspond to nonterminal symbols (variables), while spanning edges and lines correspond to terminal symbols. The initial string is the nonterminal symbol τ. The final string is a sequence of spanning edges, lines, and parentheses. For example, the recursive decomposition of the trapezoid shown in Fig. 6 is described by the string

$$((e_1 e_3)\, l_1\, (e_2 e_4 e_7) e_8 e_9)\, l_2\, (e_2 (e_4)\, l_3\, (e_5 e_6) e_{10} e_{11}).$$

Notice that the parentheses have been introduced in the horizontal decomposition to eliminate ambiguities in the parsing of such strings.

The *canonical decomposition* of a trapezoid τ with spanning edges is defined by (see Fig. 7):

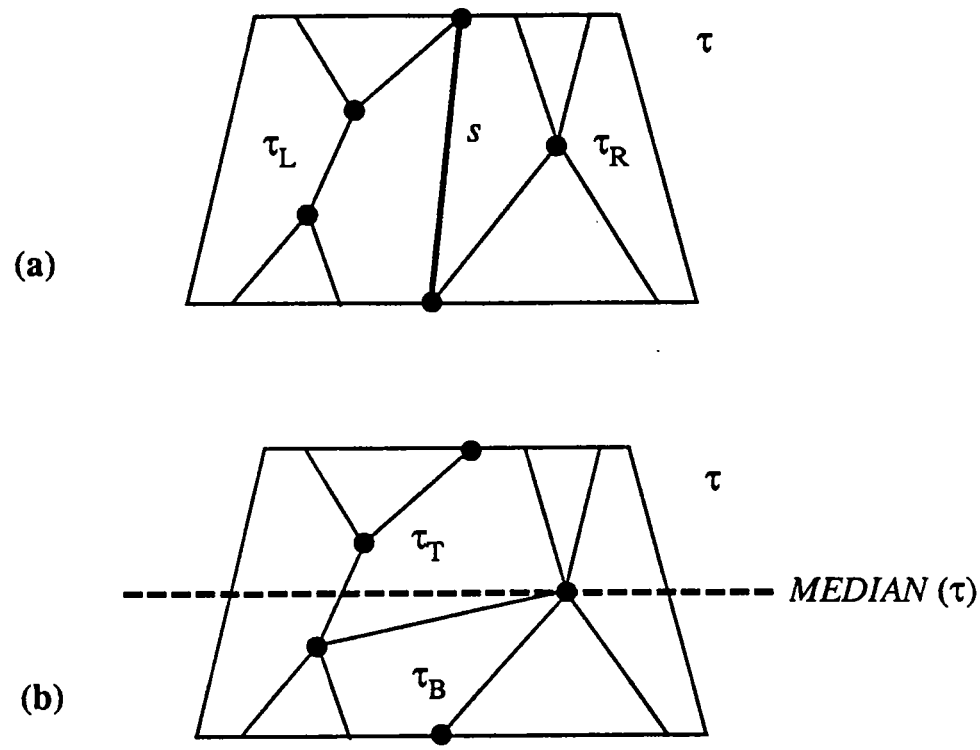


Figure 5 (a) Example of vertical cut. (b) Example of horizontal cut.

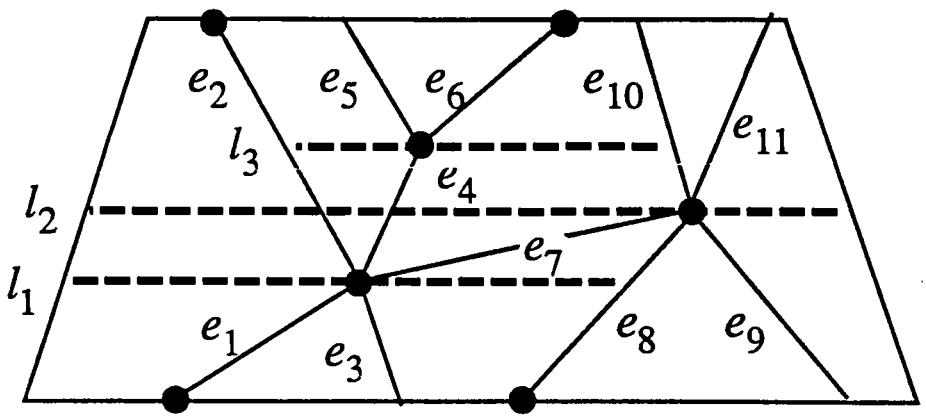


Figure 6 Recursive decomposition of a trapezoid.

$$\tau \rightarrow \tau_0\sigma_1\tau_1\sigma_2 \cdots \sigma_k\tau_k$$

where

(*i*) each τ_i, $i = 1, \cdots, k-1$, is a nonempty trapezoid without spanning edges;

(*ii*) each of τ_0 and τ_k either is empty or is a nonempty trapezoid without spanning edges; and

(*iii*) each σ_j, $j = 1, \cdots, k$, is a maximal sequence of spanning edges that delimit empty trapezoids.

4. Data Structure

The data structure for point location makes use of the so-called *biased binary tree* [1], which is a dynamic binary search tree used to represent a sorted list of weighted items. Specifically, the i-th item in the sorted list, with weight w_i, is represented by the i-th leaf of the tree. Let $w = \sum_{i=1}^{k} w_i$ be the total weight of the items in the tree. Some important properties of biased binary trees, which are relevant to this work, are listed below:

(1) The depth of the i-th leaf is at most $\log(w/w_i)+2$.

(2) Dynamic operations such as insertion/deletion of an item, change of weight of an item, and split/splice of biased trees, can be performed in time $O(\log w)$. (For the splice operation, w is the total weight of the trees to be spliced).

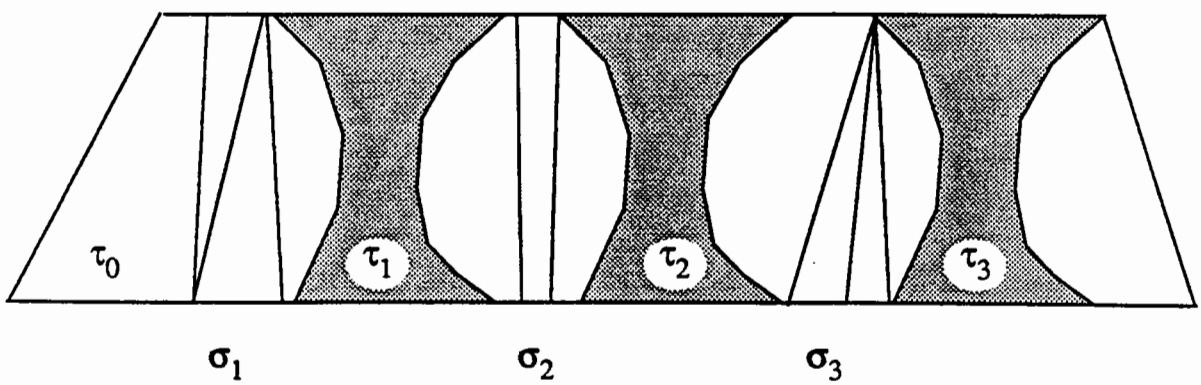


Figure 7 Canonical decomposition of a trapezoid.

The point location data structure consists of a *main component* and of an *auxiliary component*. The main component is a binary tree $\mathcal{T}(\mathcal{P})$, called *trapezoid tree*, which represents the recursive decomposition of the subdivision $\mathcal{P}$.

The nodes of the trapezoid tree $\mathcal{T}(\mathcal{P})$ are of three types (see Fig. 8):

(1) ∇-*node:* each such node is associated with a nonempty trapezoid τ which is partitioned by a horizontal cut; it stores the line *MEDIAN* (τ).

(2) *O-node:* each such node is associated with a nonempty trapezoid τ which is partitioned by vertical cuts; it stores a maximal sequence σ of spanning edges in the canonical decomposition of τ, represented by a balanced tree.

(3) ε-*node:* each such node is associated with an empty trapezoid, and is a leaf of the trapezoid tree.

In addition, each node μ stores an integer *weight* (μ), which denotes the number of vertices in the interior of the trapezoid associated with μ.

Each node of the trapezoid tree is uniquely associated with a trapezoid, so that we will sometimes use the same name for a node of the tree and for the associated trapezoid.

The trapezoid tree $\mathcal{T}(\tau)$ for a nonempty trapezoid τ is recursively defined as follows (see Fig. 9):

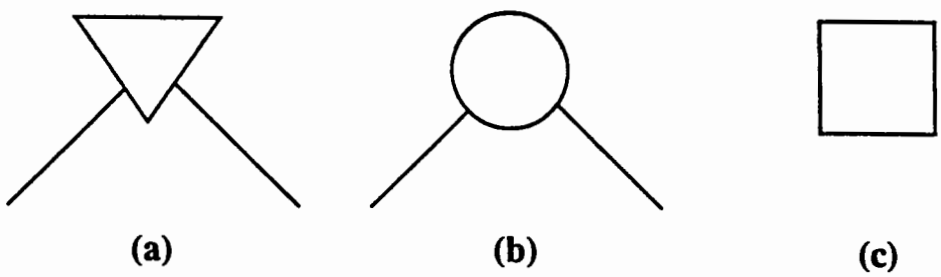


Figure 8 Nodes of a trapezoid tree: (a) ∇-node; (b) *O*-node; (c) ε-node.

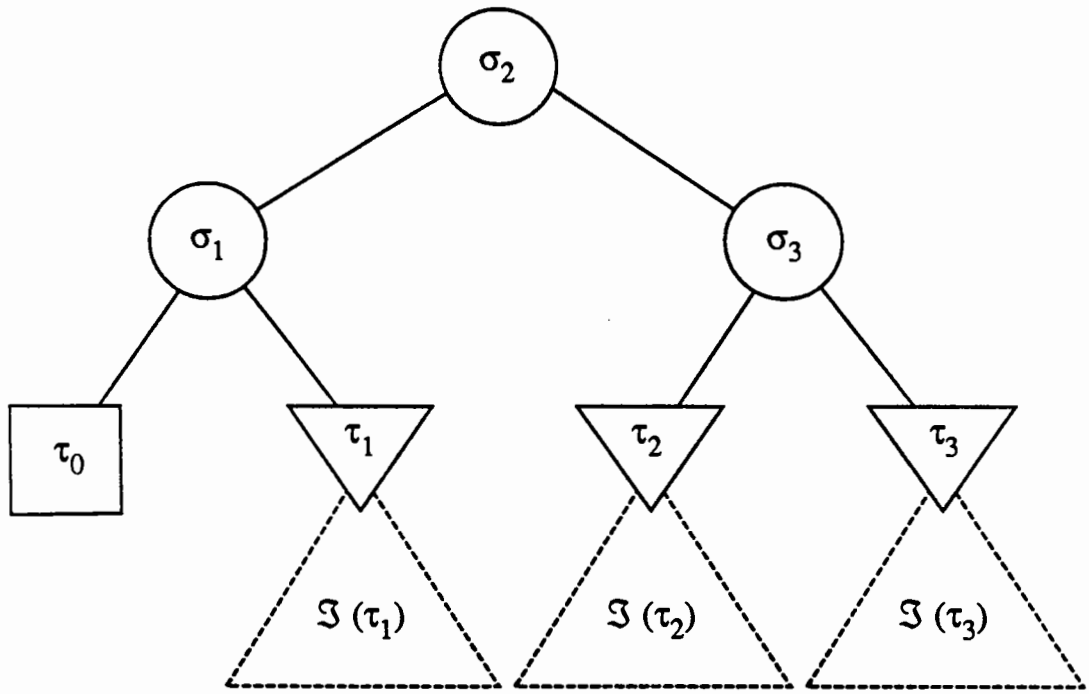


Figure 9 Trapezoid tree for the trapezoid of Fig. 7.

(1) If τ has no spanning edges, the root of $\mathcal{T}(\tau)$ is a ∇-node, and the left and right subtrees of $\mathcal{T}(\tau)$ are trapezoid trees for τ_B and τ_T, respectively.

(2) Otherwise (τ has spanning edges), we consider the canonical decomposition of τ, $\tau_0\sigma_1\tau_1\sigma_2 \cdots \sigma_k\tau_k$, and define $T(\tau)$ as a biased binary tree for the items $\tau_0, \cdots, \tau_k$. Tree $T(\tau)$ is called the *decomposition tree* of τ. The i-th leaf of $T(\tau)$ is a ∇-node for trapezoid τ_i. For the special case when $i = 0$ or $i = k$ and τ_i is empty, leaf i is a ε-node, and has nominal unit weight. The i-th internal node of T, denoted μ_i, stores the sequence σ_i of spanning edges, and is associated with the trapezoid formed by the union of the trapezoids associated with the leaves of the subtree of $T(\tau)$ rooted at μ_i. Each node of $T(\tau)$ is also equipped with pointers *pred* and *succ*, which form a doubly-linked list of the nodes of $T(\tau)$ in their left-to-right order. Finally, the trapezoid tree $\mathcal{T}(\tau)$ is obtained by replacing each leaf τ_i of $T(\tau)$ with the trapezoid tree $\mathcal{T}(\tau_i)$.

It is interesting to observe that each edge e of the subdivision $\mathcal{P}$ may be stored in the sequences of spanning edges of several O-nodes of tree $\mathcal{T}(\mathcal{P})$. Each such representative is called a *fragment* of edge e. The decomposition of e into fragments is performed according to the well-known segment-tree scheme [14, pp. 13-15], so that the number of fragments of each edge is at most $2\log N - 1$.

Theorem 2 The space requirement of the trapezoid tree $\mathcal{T}(\mathcal{P})$ is $O(n \log N)$.

Proof: Let the *order* of a ∇-node μ be the number of ∇-nodes on the path from μ to the root of $\mathcal{T}(\mathcal{P})$, including μ. Because of the median line decomposition rule, the order of μ is at most $\log N$. Also, for any $p \leq \log N$, there are at most n nodes of order p because each such node corresponds to a trapezoid with at least one vertex. Hence, the total number of ∇-nodes in $\mathcal{T}(\mathcal{P})$ is at most $n \log N$. Regarding O-nodes, we observe that each edge is decomposed according to the well-known segment-tree scheme, so that the total space requirement for the O-nodes and their sequences of spanning edges is $O(n \log N)$. Finally, we can charge at most two ε-nodes to each ∇-node, so that the total space for the trapezoid tree $\mathcal{T}(\mathcal{P})$ is $O(n \log N)$. □

Theorem 3 The depth of the trapezoid tree $\mathcal{T}(\mathcal{P})$ is $O(\log n + \log N)$.

Proof: Without significant loss of generality, assume that N is a power of 2. Let $height(\tau)$ be the number of elementary slabs spanned by a trapezoid τ. Clearly, $height(\tau) \leq N$. We show that for a ∇-node of $\mathcal{T}(\mathcal{P})$ with associated trapezoid τ, the depth of $\mathcal{T}(\tau)$ is at most $\log weight(\tau) + 3 \log height(\tau) + 1$. (For simplicity of notation, we assume that $\log 0 = 1$.) The proof is by induction on $height(\tau)$. The base of the induction, $height(\tau) = 2$, is immediate, since the left and right subtrees of $\mathcal{T}(\tau)$ consist each of a O-node and two ε-nodes. For the inductive step, define $w = weight(\tau)$ and $h = height(\tau)$. Consider the deeper of the two subtrees of τ, say, the one associated with trapezoid τ_B. Let τ_i be a leaf of the decomposition tree $T(\tau_B)$, and define $w_i = weight(\tau_i)$ and $h_i = height(\tau_i)$. Since $T(\tau_B)$ is a biased binary tree, leaf τ_i has depth bounded by $\log(w/w_i) + 2$. By the inductive hypothesis, the depth of the trapezoid tree $\mathcal{T}(\tau_i)$ is at most $\log w_i + 3 \log h_i + 1$. Hence, the total depth of the trapezoid tree $\mathcal{T}(\tau)$ is at most

$$(\log(w/w_i) + 2) + 1 + (\log w_i + 3 \log h_i + 1).$$

Since $h_i = h/2$, this is exactly $\log w + 3\log h + 1$. □

The auxiliary component of the data structure, called *dictionary*, contains a record for each vertex, region, and edge of $\mathcal{P}$. The record for a vertex v stores the name of v and its coordinates. The record for a region r stores the name of r, pointers to the records of vertices $LOW(r)$ and $HIGH(r)$, and pointers to two balanced search trees, $lchain(r)$ and $rchain(r)$, respectively, representing the edges of the left and right chain of r in bottom-to-top order. The record for an edge e stores the name of e, pointers to the records of the endpoints of e, denoted $LOW(e)$ and $HIGH(e)$, and pointers $lface(e)$ and $rface(e)$ to the representatives of e in the trees $rchain(r_1)$ and $lchain(r_2)$ such that r_1 and r_2 are the regions on the left and right sides of e, respectively. Clearly, the dictionary takes $O(n)$ space. The dynamic maintenance of the dictionary in the various update operations can be easily performed in $O(\log n)$ time, and will not be further described.

5. Query

The query algorithm is essentially the same as in the static case [12]. It consists of tracing a path in $\mathcal{T}(\mathcal{P})$. The actions taken at the current node μ are as follows:

(1) If μ is a ε-node, we stop.

(2) If μ is a ∇-node, we discriminate the query point q with respect to the median line $MEDIAN(\tau)$ of the trapezoid τ associated with μ. We proceed to the left or right child of μ depending upon whether q is below or above $MEDIAN(\tau)$. (If q is on $MEDIAN(\tau)$ we proceed to the left child.)

(3) If μ is a O-node, we discriminate q with respect to the sequence of spanning edges associated with μ, $\sigma(\mu) = s_1 \cdots s_p$. If q lies to the left of edge s_1, we set $s_R := s_1$ and proceed to the left child of μ. If q lies to the right of s_p, we set $s_L := s_p$ and proceed to the right child of μ. If q lies between two edges of $\sigma(\mu)$, say s_i and s_{i+1}, we set $s_L := s_i$ and $s_R := s_{i+1}$, and stop.

The region r reported is the one which lies on the right side of edge s_L and on the left side of edge s_R.

The time spent is $O(1)$ for the intermediate nodes on the search path, and $O(\log n)$ for the last node. The final computation of r from s_L and s_R can be done by accessing the representatives of s_L and s_R in the trees $lchain(r)$ and $rchain(r)$ pointed by $rface(s_L)$ and $lface(s_R)$, and walking up to the root of such trees, which takes $O(\log n)$ time. Since the height of tree $\mathcal{T}(\mathcal{P})$ is logarithmic, we conclude

Theorem 4 The complexity of a point location query is $O(\log n + \log N)$.

6. Insertion of Edges

First, we consider operation *INSERTSEGMENT* $(e, v_1, v_2, r; r_1, r_2)$. The insertion algorithm is described by means of a recursive procedure, whose actions, depending on the type of the current node μ, are as follows.

1. If μ is a O-node associated with the string $\sigma(\mu) = s_1 \cdots s_p$, we locate e in $\sigma(\mu)$.

1.1 If e is to the left of s_1, we recursively call the procedure on the left child of μ.

1.2 If e is to the right of s_p, we recursively call the procedure on the right child of μ.

1.3 Otherwise, e lies between two edges of $\sigma(\mu)$, say s_i and s_{i+1}. In this case, we insert e into $\sigma(\mu)$ between s_i and s_{i+1}, corresponding to the string transformation

$$s_1 \cdots s_i \, s_{i+1} \cdots s_p \rightarrow s_1 \cdots s_i \, e \, s_{i+1} \cdots s_p .$$

Note that this case can only occur when e spans the slab of $\sigma(\mu)$, for, in the interior of the corresponding region of the plane, there are no vertices of $\mathcal{P}$.

2. If μ is a ε-node, we know that it is either the leftmost or rightmost leaf of a decomposition tree. In the former case, edge e is added to the left of the sequence of spanning edges of the O-node $next(\mu)$ immediately following μ. In the latter case, edge e is added to the right of the sequence of spanning edges of the O-node $pred(\mu)$ immediately preceding μ.

3. If μ is a ∇-node associated with trapezoid τ, we determine first whether e spans τ or not.

3.1 If e does not span τ, then if $LOW(e)$ is below $MEDIAN(\tau)$ we recursively call the procedure on the left child of μ, and if $HIGH(e)$ is above $MEDIAN(\tau)$ we recursively call the procedure on the right child of μ. (Note: here is where the edge may be subjected to the segment-tree fragmentation.)

3.2 If e spans τ, we determine the leftmost and rightmost regions of τ, i.e., the region r_L to the right of edge *LEFT* (τ) and the region r_R to the left of *RIGHT* (τ). Such regions can be computed by observing that *LEFT* (τ) is the last spanning edge in the sequence $\sigma(pred(\mu))$, and *RIGHT* (τ) is the first spanning edge in the sequence $\sigma(next(\mu))$. We have three further subcases:

3.2.1 If $r = r_L$, i.e., edge e is to the left of each vertex of τ, then edge e is added to the right of the sequence of spanning edges of the O-node $pred(\mu)$ immediately preceding μ, and no further action is required. (If $pred(\mu) = nil$, a new O-node and a new ε-node are created, and the decomposition tree containing leaf μ is rebalanced.) This case corresponds to the syntactic transformation $\tau \rightarrow e\ \tau$.

3.2.2 If $r = r_R$, i.e., e is to the right of each vertex of τ, then edge e is added to the left of the sequence of spanning edges of the O-node $next(\mu)$ immediately following μ, and no further action is required. (If $next(\mu) = nil$, a new O-node and a new ε-node are created, and the decomposition tree containing leaf μ is rebalanced.) This case corresponds to the syntactic transformation $\tau \rightarrow \tau\, e$, and is symmetric to case 3.2.1.

3.2.3 If $r \neq r_L$ and $r \neq r_R$, we have that e lies inside τ, which will cause a more substantial update of the data structure (see Fig. 10). In fact, while τ was formerly horizontally decomposed, it has now to be vertically decomposed by means of a vertical cut along edge e. First, we call recursively the procedure on both the left and right children of μ. At this point, both trapezoids τ_B and τ_T have a vertical cut along edge e, corresponding to the string

$$(\tau_{11}\sigma_{11} e\, \sigma_{12}\tau_{12})\, l\, (\tau_{21}\sigma_{21} e\, \sigma_{22}\tau_{22}).$$

To comply with the rule that vertical cuts take precedence over horizontal cuts, we have to restructure the tree according to the string

$$(\tau_{11}\sigma_{11})\, l\, (\tau_{21}\sigma_{21})\, e\, (\sigma_{12}\tau_{12})\, l\, (\sigma_{22}\tau_{22}).$$

This is done as follows:

(1) Let μ_L and μ_R be the left and right child of μ, respectively. Find node μ_1 of $T(\mu_L)$ associated with the sequence $\sigma_1 = \sigma_{11} e\, \sigma_{12}$. Similarly, find node μ_2 of $T(\mu_R)$ associated with the sequence $\sigma_2 = \sigma_{21} e\, \sigma_{22}$.

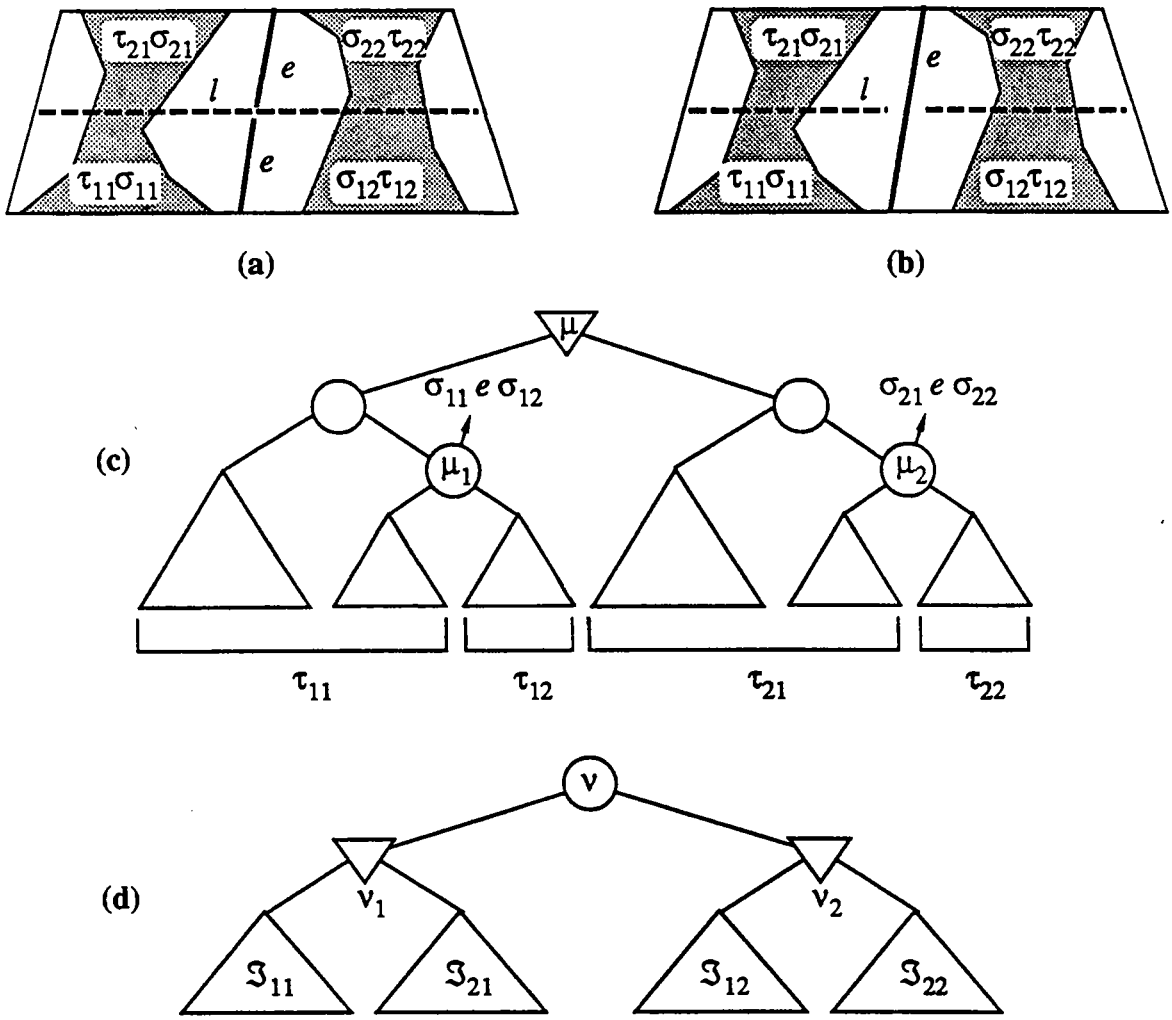


Figure 10 Update of the data structure in Case 3.2.3 of *INSERTSEGMENT*. **(a,c)** Preliminary restructuring (edge e is inserted in both children of μ). **(b,d)** Final restructuring.

(2) Split $T(\mu_L)$ and $T(\mu_R)$ at nodes μ_1 and μ_2, respectively. This yields trees $\mathcal{T}_{11}$, $\mathcal{T}_{12}$, $\mathcal{T}_{21}$, and $\mathcal{T}_{22}$.

(3) Split the sequences $\sigma(\mu_1)$ and $\sigma(\mu_2)$ at edge e. This yields the sequences σ_{11}, σ_{12}, σ_{21}, and σ_{22}.

(4) If $\sigma_{11}\neq\emptyset$, create a *O*-node μ_{11} for it, and add μ_{11} and a ε-node to the right of $\mathcal{T}_{11}$. If $\sigma_{21}\neq\emptyset$, create a *O*-node μ_{21} for it, and add μ_{21} and a ε-node to the right of $\mathcal{T}_{21}$. If $\sigma_{12}\neq\emptyset$,

create a O-node μ_{12} for it, and add μ_{12} and a ε-node to the left of $\mathcal{T}_{12}$. If $\sigma_{22} \neq \emptyset$, create a O-node μ_{22} for it, and add μ_{22} and a ε-node to the left of $\mathcal{T}_{22}$.

(5) Create a ∇-node ν_1 (associated with the portion of τ to the left of e), with left subtree $\mathcal{T}_{11}$ and right subtree $\mathcal{T}_{21}$.

(6) Create a ∇-node ν_2 (associated with the portion of τ to the right of e), with left subtree $\mathcal{T}_{12}$ and right subtree $\mathcal{T}_{22}$.

(7) Replace μ with a O-node ν having $\sigma(\nu) = e$, left child ν_1, and right child ν_2. Finally, rebalance the decomposition tree containing node ν.

This completes the description of the algorithm for operation *INSERTSEGMENT*.

We now analyze the time complexity of operation *INSERTSEGMENT*. Let r^* be the intersection of region r and the horizontal slab spanned by segment e (see Fig. 11). Notice that, since r is 2-supported, so is r^*. The insertion of e into the trapezoid tree can be viewed as a visit of a subgraph of $\mathcal{T}(\mathcal{P})$ (see Fig. 12). Specifically, this subgraph consists of an initial path

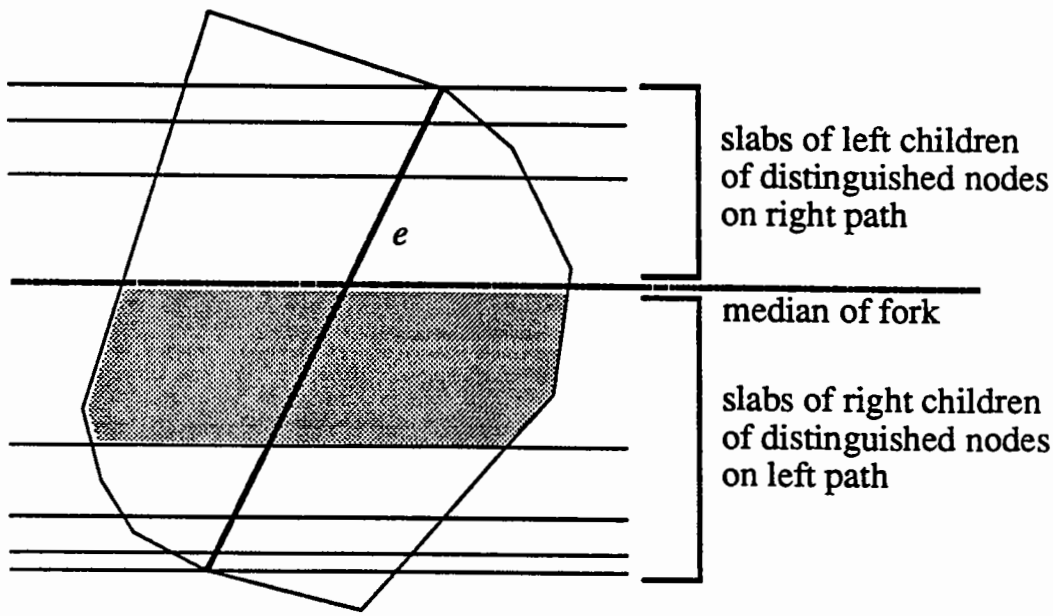


Figure 11 Region r^* in the analysis of operation *INSERTSEGMENT*.

originating at the root and consisting of O-nodes such that e is always external to the sequence of spanning edges (Case 1.1 or 1.2), and of ∇-nodes such that e never crosses the median of the corresponding slab (Case 3.1). For either type of node, $O(1)$ time is spent at each node. The visit of this initial path terminates at a ∇-node (referred to as the *fork*), where, for the first time, e crosses the median.

At the fork, e is passed on to both children, i.e., the visit continues with two distinct paths, called the *left spine* and *right spine*. Let us consider the left spine (the right spine is similarly analyzed). It is easily realized that this path contains a set of distinguished ∇-nodes such that, for trapezoid τ associated with any of them, $y(TOP(\tau)) \leq y(HIGH(e))$ and $y(BOT(\tau)) < y(LOW(e)) \leq y(MEDIAN(\tau))$. This implies that e spans the trapezoid τ_T associated with the right subtree of the distinguished ∇-node τ. The decomposition tree $T(\tau_T)$ is called an *allocation subtree* for edge e. The slabs spanned by the roots of all allocation subtrees of e form a partition of r^* into a set of convex polygons, of which it is easily realized that at most one is 2-supported while all the others are 1-supported (see Fig. 11).

Each distinguished ∇-node of the left spine is a branching point for the visit, since the procedure is called recursively on both children. The left child is on the left spine, while the right child is the root of an allocation subtree for e. Within the allocation subtree the visit continues along a path, called *allocation trail* until we reach a node, called *allocation target* for which one of the Cases 1.3, 2, or 3.2 occurs. The time spent at each node of the left spine and at each node of the allocation trail, target excluded, is $O(1)$.

Consider now an allocation target μ of e. If μ is an ε-node, then its handling takes $O(\log n)$ time (Case 2). If μ is an O-node, then in $O(\log n)$ time e is inserted into the corresponding $\sigma(\mu)$ (Case 1.3). Finally, if μ is a ∇-node, we are in Case 3.2. If the intersection of r with the slab of μ is a 1-supported polygon, then we have either Case 3.2.1 or Case 3.2.2, since e is adjacent to a spanning edge of this slab. For such cases, the processing of μ takes time $O(\log n)$. Else (when the intersection of r with the slab of μ is a 2-supported polygon), we are in Case 3.2.3. From Lemma 1, we have that this case can occur for at most one allocation target, called henceforth *special target.*

Let μ be the special target, and let τ be the associated trapezoid. In the processing of μ, we recursively call the procedure on both children of μ, so that the visit branches again into two

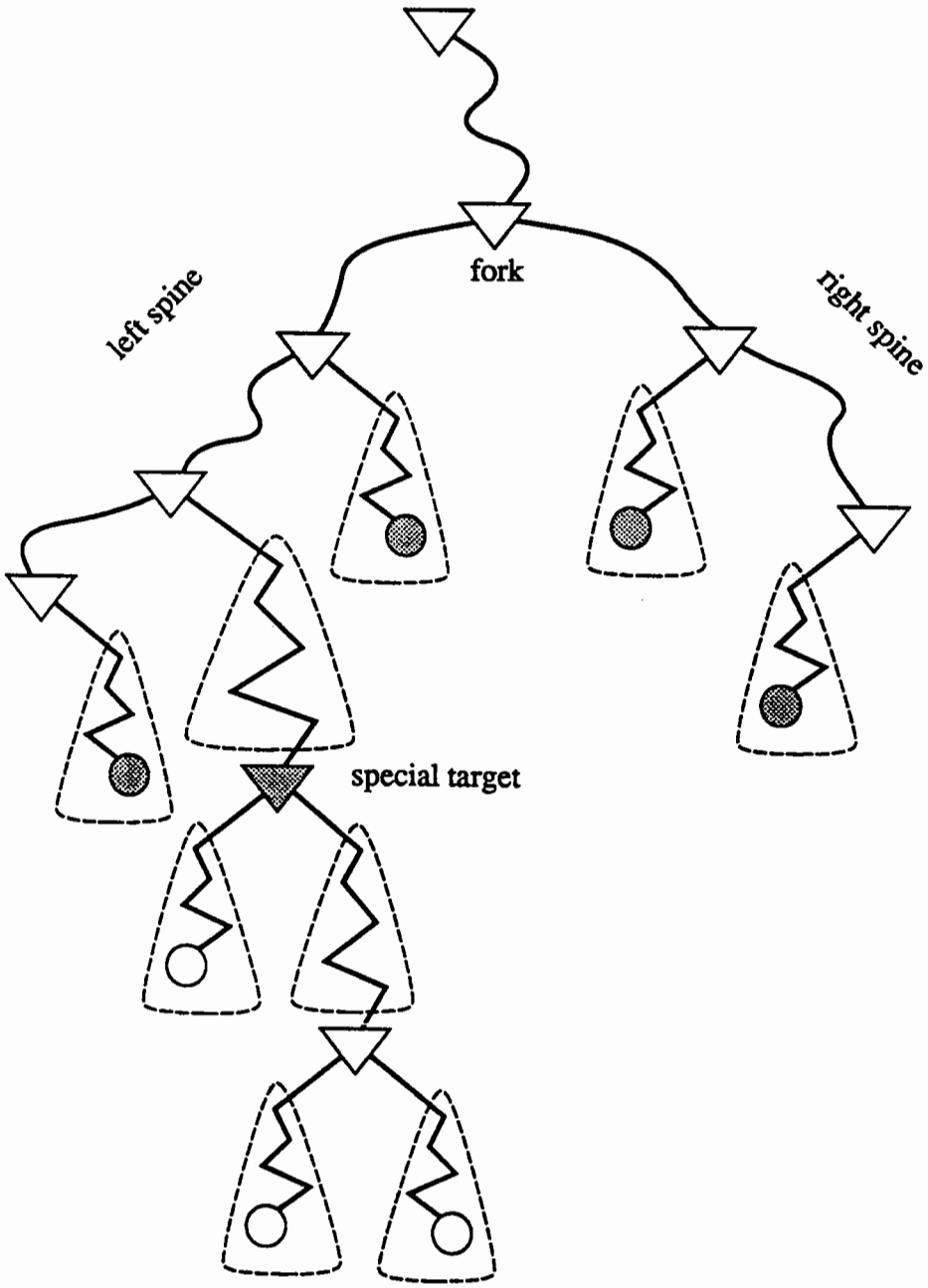


Figure 12 Subgraph of the trapezoid tree visited by the *INSERTSEGMENT* algorithm. The target nodes are shaded.

paths. Since edge e spans both τ_B and τ_T, each path will encounter a node for which one of the Cases 1.3, 2, or 3.2 occurs. However, by Lemma 1, at most one of the polygons obtained by intersecting region r with τ_B and τ_T is 2-supported, so that Case 3.2.3 can occur again for at most one such node. Finally, after the recursive calls have been executed, we perform the restructuring specified in the description of Case 3.2.3, which takes $O(\log n)$ time. Let p be the order of μ, i.e., the number of nodes on the path from μ to the root. The work done for completing the visit originating at μ requires in the worst case time $O(\log n)$ plus the time for executing one recursive call to the procedure on a node of order $p-1$. Since the order of a node is at most $\log N$, this sums up to $O(\log n \log N)$ time.

We summarize the preceding discussion as follows:

(i) The total time spent at the nodes of the path from the root to the fork and at the nodes of the left and right spine is $O(\log n + \log N)$.

(ii) The total time spent in traversing each allocation trail is $O(\log n)$, for a total time $O(\log n \log N)$.

(iii) The time spent for processing each nonspecial target node is $O(\log n)$, for a total time $O(\log n \log N)$.

(iv) The time spent for processing the special target and its subtree is $O(\log n \log N)$.

We therefore conclude

Theorem 5 The complexity of operation *INSERTSEGMENT* $(e, v_1, v_2, r; r_1, r_2)$ is $O(\log n \log N)$.

Remark. The underlying segment-tree structure of the method effects the fragmentation into a logarithmic number of subsegments of an edge to be inserted. Each of these subsegments, however, may not be insertable as a single entity, if it does not occur next to existing spanning edges of the corresponding slab. Note, however, that such a spanning edge would always exist if each region of $\mathcal{P}$ were a 1-supported polygon. Relaxing such strong constraint to 2-supportedness, we effectively create a "channel" (a sort of "virtual diagonal") between *LOW* (r) and *HIGH* (r) of each polygon r of $\mathcal{P}$, which guarantees that at most one segment fragment (the one reaching the special target) is further subdivided into a number of subsegments at most

logarithmic in N, before being restored to its original length as specified by the string transformation given for Case 3.2.3.

7. Other Update Operations

The algorithm for operation *REMOVESEGMENT* $(e, v_1, v_2, r_1, r_2; r)$ is similar to the one for the *INSERTSEGMENT* operation, and for brevity we omit its description.

Theorem 6 The complexity of operation *REMOVESEGMENT* $(e, v_1, v_2, r_1, r_2; r)$ is $O(\log n \log N)$.

We consider now operation *INSERTPOINT* $(v, e; e_1, e_2)$. The algorithm consists of two phases. The first phase is performed by a recursive procedure consisting of the following steps:

(1) Using the query algorithm, we search for a node μ of the trapezoid tree such that v lies in the trapezoid τ of μ, and e is a spanning edge of $\sigma(\mu) = s_1, \cdots, s_p$, i.e., $e = s_i$ for some i.

(2) We perform some restructurings at node μ (to be specified later), which correspond to inserting v onto line *MEDIAN* (τ).

(3) We recursively call the procedure if v is not on line *MEDIAN* (τ).

The restructurings performed in Step 2 depend on the position of edge e in the sequence $\sigma(\mu)$. We distinguish four cases:

1. If $e = s_i$ is a nonextreme element of the sequence $\sigma(\mu)$, i.e. $i \neq 1, p$, the restructuring is dictated by the string transformation

$$s_{i-1}\, e\, s_{i+1} \rightarrow s_{i-1}\, (e)\, l\, (e)\, s_{i+1}.$$

Omitting the details, node μ is replaced in the decomposition tree by nodes μ_1, μ', and μ_2, where μ_1 and μ_2 are O-nodes with sequences $\sigma(\mu_1) = s_1, \cdots, s_{i-1}$ and $\sigma(\mu_2) = s_{i+1}, \cdots, s_p$, and μ' is a ∇-node. In turn, μ' is the root of an elementary decomposition tree, whose left and right subtree consist each of a O-node with sequence e, and of two ε-nodes.

2. If $e = s_1$ is the leftmost element of $\sigma(\mu)$ and $p \geq 2$, the restructuring is dictated by the string transformation

$$(\tau_1)\, l\, (\tau_2)\, e\, s_2 \rightarrow (\tau_1\, e)\, l\, (\tau_2\, e)\, s_2.$$

Let $\mu' = pred(\mu)$. Node μ' is associated with trapezoid τ' to the left of e. Edge e is inserted into the left and right children of μ' as in the algorithm for *INSERTSEGMENT*.

3. If $e = s_p$ is the rightmost element of $\sigma(\mu)$ and $p \geq 2$, the restructuring is dictated by the string transformation

$$s_{p-1}\, e\, (\tau_1)\, l\, (\tau_2) \rightarrow s_{p-1}\, (e\, \tau_1)\, l\, (e\, \tau_2).$$

This case is symmetric to Case 2 above.

4. If $e = s_1 = s_p$ is the unique element of $\sigma(\mu)$, the restructuring is dictated by the string transformation

$$(\tau_{11})\, l\, (\tau_{21})\, e\, (\tau_{12})\, l\, (\tau_{22}) \rightarrow (\tau_{11}\, e\, \tau_{12})\, l\, (\tau_{21}\, e\, \tau_{22}).$$

This restructuring merges the two trapezoids associated with the ∇-nodes $\mu_1 = pred(\mu)$ and $\mu_2 = next(\mu)$ into a unique trapezoid, represented by a new node μ'. The tasks to be performed are analogous to the ones for Case 3.2.3 of the edge insertion algorithm, and omitted for brevity.

In Step 3, if v is below *MEDIAN* (τ) we recursively call the procedure on the left subtree of μ', while if v is above *MEDIAN* (τ) we recursively call the procedure on the right subtree of μ'.

A detailed time complexity analysis of the first phase can be conducted using techniques similar to the ones presented in the preceding section. In summary, the recursion depth is at most $O(\log N)$, and the time for Step 2 is $O(\log n)$ for all but (possibly) one node, for which it is $O(\log n \log N)$. Hence the first phase takes time $O(\log n \log N)$.

The second phase rebalances the decomposition trees embedded in $\mathcal{T}(\mathcal{P})$ for which the weight of some leaf has increased by one. There are at most $O(\log N)$ such trees, so that the second phase also takes time $O(\log n \log N)$.

We therefore conclude

Theorem 7 The complexity of operation *INSERTPOINT* $(v, e; e_1, e_2)$ is $O(\log n \log N)$.

The algorithm for operation *REMOVEPOINT* $(v, e_1, e_2; e)$ is similar to the one for operation *INSERTPOINT*, and we have

Theorem 8 The complexity of operation *REMOVEPOINT* $(v, e_1, e_2; e)$ is $O(\log n \log N)$.

Finally, as regards operation *MOVEPOINT* $(v; x)$, we observe that it does not affect the trapezoid tree. Hence, it can be simply carried out by updating the dictionary in time $O(\log n)$.

In conclusion, we have proved the following theorem:

Theorem 9 Let $\mathcal{P}$ be a convex subdivision whose n vertices lie on a fixed set L of N horizontal lines. There exists a dynamic point location data structure for $\mathcal{P}$ with space $O(N + n \log N)$ and query time $O(\log n + \log N)$, which allows for insertion/deletion of vertices and edges in time $O(\log n \log N)$, all time bounds being worst-case.

References

[1] S.W. Bent, D.D. Sleator, and R.E. Tarjan, "Biased Search Trees," *SIAM J. Computing*, vol. 14, no. 3, pp. 545-568, 1985.

[2] D.P. Dobkin and R.J. Lipton, "Multidimensional Searching Problems," *SIAM J. Computing*, vol. 5, no. 2, pp. 181-186, 1976.

[3] M. Edahiro, I. Kokubo, and T. Asano, "A New Point-Location Algorithm and its Practical Efficiency – Comparison with Existing Algorithms," *ACM Trans. on Graphics*, vol. 3, no. 2, pp. 86-109, 1984.

[4] H. Edelsbrunner, L.J. Guibas, and J. Stolfi, "Optimal Point Location in a Monotone Subdivision," *SIAM J. Computing*, vol. 15, no. 2, pp. 317-340, 1986.

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

[6] O. Fries, K. Mehlhorn, and S. Naeher, "Dynamization of Geometric Data Structures," *Proc. ACM Symp. on Computational Geometry*, pp. 168-176, 1985.

[7] D.G. Kirkpatrick, "Optimal Search in Planar Subdivisions," *SIAM J. Computing*, vol. 12, no. 1, pp. 28-35, 1983.

[8] D.T. Lee and F.P. Preparata, "Location of a Point in a Planar Subdivision and its Applications," *SIAM J. Computing*, vol. 6, no. 3, pp. 594-606, 1977.

[9] R.J. Lipton and R.E. Tarjan, "Applications of a Planar Separator Theorem," *Proc. 18th IEEE Symp. on Foundations of Computer Science*, pp. 162-170, 1977.

[10] K. Mehlhorn, *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, Springer-Verlag, New York, 1984.

[11] M. Overmars, "Range Searching in a Set of Line Segments," *Proc. ACM Symp. on Computational Geometry*, pp. 177-185, 1985.

[12] F.P. Preparata, "A New Approach to Planar Point Location," *SIAM J. Computing*, vol. 10, no. 3, pp. 473-483, 1981.

[13] F.P. Preparata and R. Tamassia, "Fully Dynamic Techniques for Point Location and Transitive Closure in Planar Structures," *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pp. 558-567, 1988.

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

[15] N. Sarnak and R.E. Tarjan, "Planar Point Location Using Persistent Search Trees," *Communications ACM*, vol. 29, no. 7, pp. 669-679, 1986.

[16] R. Tamassia, "Dynamic Data Structures for Two-Dimensional Searching," Ph.D. Thesis (Technical Report ACT-100), Coordinated Science Laboratory, Univ. of Illinois at Urbana-Champaign, 1988.