BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-91-M8

Dynamization of the Trapezoid Method for Planar Point Location

by
Yi-Jen Chiang

Dynamization of the Trapezoid Method for Planar Point Location

by

Yi-Jen Chiang

Department of Computer Science
Brown University

Master's Thesis

Submitted in partial fulfillment of the requirments
for the degree of Master of Science in the
Department of Computer Science at Brown University

May, 1991

This thesis by Yi-Jen Chiang is accepted in its present form by the Department of Computer Science as satisfying the thesis requirement for the degree of Master of Science.

Date May 1, 1991

Roberto Tamassia
Advisor

Abstract

We present a fully dynamic data structure for point location in a monotone subdivision, based on the trapezoid method. The operations supported are insertion and deletion of vertices and edges, and horizontal translation of vertices. Let n be the current number of vertices of the subdivision. Point location queries take $O(\log n)$ time, while updates take $O(\log^2 n)$ time. The space requirement is $O(n \log n)$. This is the first fully dynamic point location data structure for monotone subdivisions that achieves optimal query time.

1 Introduction

Planar point location is a fundamental geometric searching problem and has been extensively studied. Given a subdivision P of the plane into polygonal regions, we want to perform on-line queries that ask for the region of P containing a given query point. In the *static* case, where P is fixed, there are optimal techniques that achieve $O(\log n)$ query time using $O(n \log n)$ preprocessing time and $O(n)$ space [8, 4, 19], where n is the size of P.

Research on dynamic algorithms for geometric problems has received increasing attention in the last years. A survey on the subject appears in [3]. Previous results on dynamic point location data structures, where queries are intermixed with update operations that insert or delete vertices and edges, are summarized below. In the following, we denote with n the current size of the subdivision.

Early work of Overmars [14] and of Fries, Mehlhorn, and Naeher [5, 6] shows that polylogarithmic query and update time can be achieved using data structures derived from the segment-tree and the static chain-method [9].

Preparata and Tamassia [17, 18, 20] present efficient data structures for some important classes of planar subdivisions. Namely, for monotone subdivisions queries and updates can be performed in $O(\log^2 n)$ time using $O(n)$ space [17]. For convex subdivisions with vertices on a fixed set of N horizontal lines, there exists a data structure with space $O(N + n \log N)$, query time $O(\log n + \log N)$, and update time $O(\log n \log N)$ [18]. Finally, for triangulations, one can achieve $O(n)$ space and a tradeoff between query and update time; for example, $O((\log^2 n)/\log\log n)$ query time and $O((\log^3 n)/\log\log n)$ update time, or $O(\log n)$ query time and $O(n^\epsilon \log n)$ update time [20].

Cheng and Janardan [2] present a technique for general subdivisions, derived from priority search trees [10], with $O(n)$ space, $O(\log^2 n)$ query time, $O(\log n)$ time for inserting/deleting a vertex, and $O(k \log n)$ time for inserting/deleting a chain of k edges. However, as emphasized by the authors, this method is mainly of theoretical interest because it involves rather complex manipulations of data structures.

Very recently, Goodrich and Tamassia [7] have shown how to dynamically maintain a monotone subdivision so as to achieve $O(n)$ space, $O(\log^2 n)$ query time, and $O(\log n)$ update time. This method is based on a new, optimal static point-location data structure, which uses interlaced spanning trees, one for the subdivision and one for its graph-theoretic dual, to answer queries. A variation of this method supports updates in a semi-dynamic environment where only insertions are performed. Two semi-dynamic data structures are presented, one with $O(\log n)$ query and insertion time (worst-case for queries and amortized for insertions), and the other with $O(\log n \log\log n)$ query time and $O(1)$ amortized time for insertions.

In this paper we present a fully dynamic data structure for point location in a monotone subdivision, based on the trapezoid method [15]. The operations supported are insertion and deletion of vertices and edges, and horizontal translation of vertices. Point location queries take $O(\log n)$ time, while updates take $O(\log^2 n)$ time. The space requirement is $O(n \log n)$. This is the first fully dynamic point location data structure for monotone subdivisions that achieves optimal query time and polylogarithmic update time.

To compare our result with previous ones, we note that

- The fully dynamic data structures of Cheng-Janardan [2] and Goodrich-Tamassia [7] have slower query time but faster update time and less space requirement (both by a $\log n$ factor).

- The fully dynamic data structure of Preparata-Tamassia [18] has the same performance bounds but is limited to convex subdivisions and to a fixed set of lines on which the vertices can be placed.

- The fully dynamic data structure of Tamassia [20] achieves the same query time but is limited to triangulations and has a much higher update time.

- The semi-dynamic data structure of Goodrich-Tamassia [7] has the same query time and better insertion time, but is limited to insertions only.

It is conceivable that in many practical applications point location queries are the most critical operations, so that our method is especially suitable for them.

Our technique extends to general monotone subdivisions the previous results of Preparata and Tamassia for convex subdivisions with vertices on a fixed set of N horizontal lines [18]. Several new ideas have been used to achieve such an extension. We leave as a challenging open problem the dynamization of the trapezoid method for point location in general (nonmonotone) subdivisions.

The rest of this paper is organized as follows. Section 2 contain preliminary definitions and results. In Section 3 we give a decomposition scheme for monotone subdivisions. In Section 4 we present a data structure for monotone subdivisions with vertices on a fixed set of horizontal lines. We show how to remove this restriction in Section 5, which describes the fully dynamic data structure for monotone subdivisions.

2 Preliminaries

A polygonal chain is *monotone* if any horizontal line intersects it in at most one point. A *monotone polygon* r is such that its boundary can be partitioned into two monotone chains, which share the highest and lowest points of r. A *planar subdivision* P is a partition of the plane into polygons, called the *regions* of P. A *monotone subdivision* P is a planar subdivision where each region is a monotone polygon. We assume that the edges of a monotone subdivision are oriented from the lowest to the highest endpoint. For additional geometric definitions, see [16].

Let $L = \{l_0, l_1, \cdots, l_N\}$ be a set of horizontal lines, in this order from bottom to top, with $l_0 : y = -\infty$ and $l_N : y = +\infty$. The lines of L partition the plane into horizontal strips, called *elementary slabs*; the top-most and bottom-most ones are actually half planes. A *slab* is either an elementary slab or the union of two contiguous slabs. First we consider the point location problem in a monotone subdivision P whose n vertices lie on the lines of L; later on we will generalize the solution such that this restriction can be removed.

We define the following update operations on a monotone subdivision P:

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 along e.

REMOVEPOINT$(v, e_1, e_2; e)$: 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 $e = (u, w)$.

INSERTSEGMENT$(e, v_1, v_2, r; r_1, r_2)$: Insert edge $e = (v_1, v_2)$ into region r such that r is partitioned into two regions r_1 and r_2, with r_1 on the left of e and r_2 on the right of e.

REMOVESEGMENT$(e, v_1, v_2, r_1, r_2; r)$: Remove edge $e = (v_1, v_2)$ and merge the regions r_1 and r_2 formerly on the left and right of e into a new region r. This operation is allowed only if the resulting subdivision is monotone.

MOVEPOINT$(v; x)$: Translate a vertex v horizontally so that its x-coordinate becomes x. This operation is allowed only when v has degree two and the resulting subdivision P' is topologically equivalent to P.

The above repertory is complete for the class of monotone subdivisions:

Theorem 1 *An arbitrary monotone subdivision P with n vertices can be assembled from the empty subdivision, and disassembled to obtain the empty subdivision, by a sequence of $O(n)$* INSERTPOINT, REMOVEPOINT, INSERTSEGMENT, REMOVESEGMENT, *and* MOVEPOINT *operations. Also, such a sequence can be computed from P in $O(n)$ time.*

In the following, we shall make use of *biased binary trees* [1], which are binary search trees whose leaves store weighted items. Let w be the sum of all weights. We have that the depth of a leaf with weight w_i is at most $\log(w/w_i) + 2$, and each of the following update operations can be done in $O(\log w)$ time: change of the weight of an item, insertion/deletion of an item, and split/splice of two biased trees.

3 Decomposing a Subdivision into Trapezoids

Our point location method is based on a recursive decomposition of the subdivision P into trapezoids. A *trapezoid* τ is a quadrilateral with two horizontal sides that lie on the lines of L. We use LEFT(τ), RIGHT(τ), BOT(τ), and TOP(τ) to denote the four sides of τ. Let BOT(τ) and TOP(τ) be on lines l_i and l_j, respectively, then the *median line* of τ, denoted by MEDIAN(τ), is the line l_m with $m = \lfloor (i + j)/2 \rfloor$. The subdivision P is the trapezoid with BOT$(P) = l_0$, TOP$(P) = l_N$, and the other two sides at infinity.

A monotone chain is said to *span* a trapezoid τ if it has a subchain that is inside τ and intersects the boundary of τ in two points on BOT(τ) and TOP(τ), respectively. An edge

e of P that spans τ is called a *spanning edge* of τ. Spanning edge e partitions τ into two trapezoids τ_L and τ_R, with $e = \text{RIGHT}(\tau_L) = \text{LEFT}(\tau_R)$.

A *spanning region* of a trapezoid τ is a region r of P such that (i) both the left and right chains of r span τ, and (ii) r contains a segment that spans τ. Let r' be the portion of r inside τ. We say that r' is a *gap* of τ if r' does not have spanning edges of τ, i.e., each of the left and right chains of r' has at least two edges. It is easy to see that r is a spanning region of τ if and only if the right convex hull of the left chain of r' and the left convex hull of the right chain of r' do not cross (but are allowed to touch), see Fig. 1(b). If r' is a gap, the *spanning tangents* of τ in r' are defined as the two (possibly coincident) spanning segments of τ that are tangent to the left and right chains of r'. A spanning tangent t decomposes τ into two trapezoids τ_L and τ_R, with $t = \text{RIGHT}(\tau_L) = \text{LEFT}(\tau_R)$.

The decomposition of τ by means of a spanning edge or tangent is called a *vertical cut* (see Fig. 1(a,b)). If a trapezoid τ has neither spanning edges nor spanning tangents, we decompose it by its median line $\text{MEDIAN}(\tau)$ into two trapezoids τ_B and τ_T, with $\text{MEDIAN}(\tau) = \text{TOP}(\tau_B) = \text{BOT}(\tau_T)$. This is called a *horizontal cut* (see Fig. 1(c)). A trapezoid can always be decomposed unless it is empty. We let a vertical cut take precedence over a horizontal cut. Therefore the decomposition process is unique up to within the specification of the sequence of consecutive vertical cuts of the same trapezoid and the choice of spanning tangents. An example of recursive decomposition of a trapezoid is shown in Fig. 3(a).

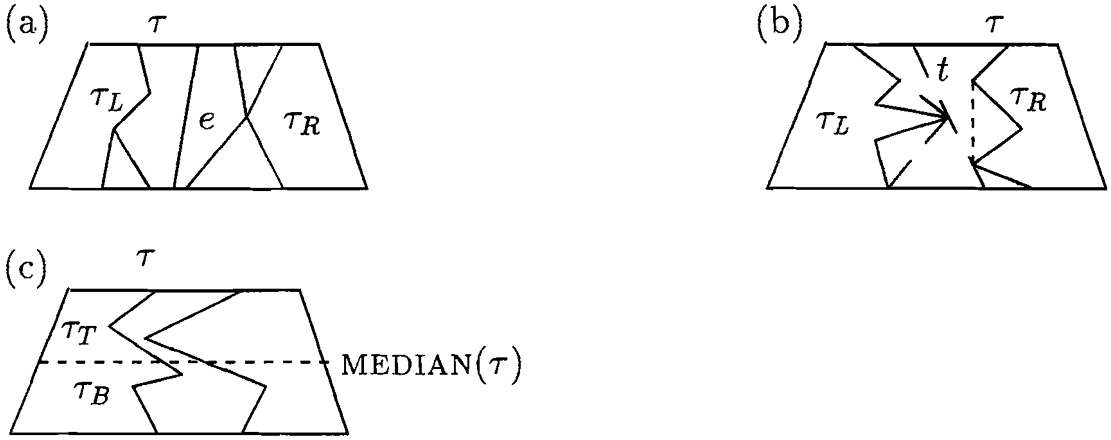

Figure 1: (a) vertical cut by a spanning edge; (b) vertical cut by a spanning tangent; (c) horizontal cut by the median line.

The *canonical decomposition* of a trapezoid τ with spanning edges and spanning tangents is the sequence $\tau_0\sigma_1\tau_1 \cdots \sigma_k\tau_k$, where each τ_i is a trapezoid without spanning edges or tangents (τ_0 and τ_k may be empty), and each σ_j is either a spanning tangent or a maximal sequence of spanning edges that separate empty trapezoids (see Fig. 2).

4 Data Structure for a Fixed Set of Y-Coordinates

In this section we study a restricted version of dynamic point location where all vertices of a monotone subdivision P lie on a fixed set L of N horizontal lines. The point location

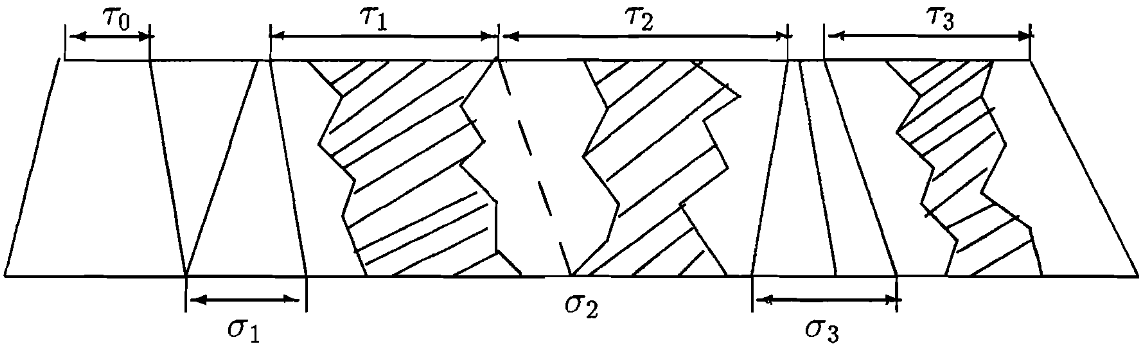

Figure 2: Canonical decomposition of a trapezoid

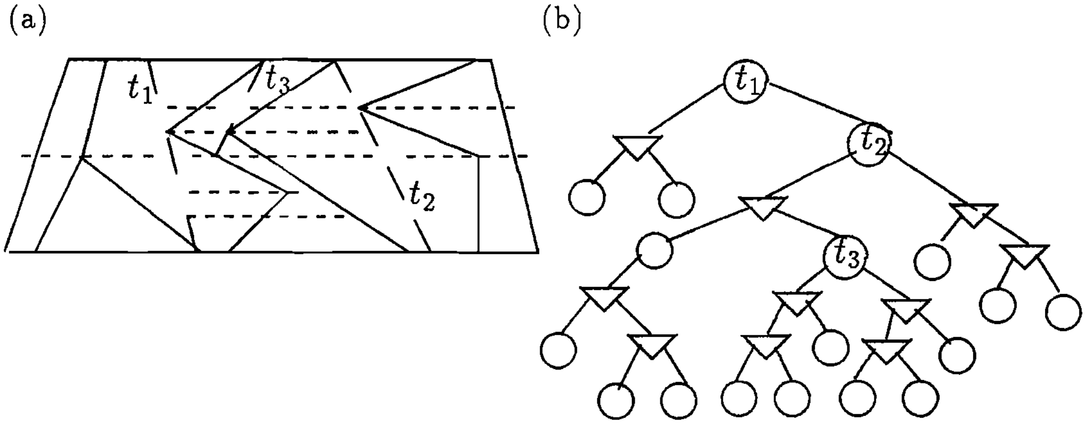

Figure 3: (a) Recursive decomposition of a trapezoid τ. (b) Trapezoid tree for τ with ε-nodes omitted.

data structure for P consists of two main components: the *trapezoid tree* and the *hull trees.* In addition, a dictionary stores the vertices, edges, and regions, so that their associated substructures in the trapezoid tree and hull trees can be efficiently accessed. We denote with n the current number of vertices of P.

4.1 Trapezoid Tree

The *trapezoid tree*, denoted $\mathcal{T}(P)$, is a binary tree that represents the recursive decomposition of the subdivision P into trapezoids. Each node μ represents a trapezoid τ of the decomposition and stores an integer $weight(\mu)$, denoting the number of vertices inside τ. There are four types of nodes: an ε-node corresponds to an empty trapezoid, and is a leaf of the trapezoid tree. The other types (and corresponding cuts) are: O-node (spanning edge), R-node (spanning tangent), and ∇-node (median line). In the figures, we use squares for ε-nodes, circles for O-nodes and R-nodes, and triangles for ∇-nodes.

The trapezoid tree $\mathcal{T}(\tau)$ for a trapezoid τ is recursively defined as follows (see Fig. 3(b)):

1. If τ is empty, then $\mathcal{T}$ consists of a single ε-node.

2. If τ has no spanning edges or tangents, the root of $\mathcal{T}(\tau)$ is a ∇-node storing the median line of τ, and the left and right subtrees of $\mathcal{T}(\tau)$ are the trapezoid trees $\mathcal{T}(\tau_B)$ and $\mathcal{T}(\tau_T)$, respectively.

3. If τ has spanning edges or tangents, let the canonical decomposition of τ be $\tau_0\sigma_1\tau_1 \cdots \sigma_k\tau_k$, and recall that each σ_i is either a spanning tangent or a maximal sequence of spanning edges. We define the *decomposition tree* of τ, denoted by $T(\tau)$, to be a biased binary tree for the items $\tau_0, \cdots, \tau_k$, where the i-th leaf of $T(\tau)$ is a ∇-node for trapezoid τ_i. (If $i \in \{0, k\}$ and τ_i is empty, leaf i of $T(\tau)$ is an ε-node, with unit weight.) The i-th internal node μ_i of T stores σ_i, where if σ_i is a maximal sequence of spanning edges, it is stored in a secondary structure as a balanced search tree. Node μ_i corresponds to the trapezoid formed by the union of the trapezoids associated with the leaves in the subtree of $T(\tau)$ rooted at μ_i. Finally, the trapezoid tree $\mathcal{T}(\tau)$ is obtained by replacing each leaf τ_i of $T(\tau)$ with $\mathcal{T}(\tau_i)$, the trapezoid tree for τ_i. Note the difference between the *trapezoid tree* $\mathcal{T}(\tau)$ and the *decomposition tree* $T(\tau)$.

Theorem 2 *The space complexity of the trapezoid tree $\mathcal{T}(P)$ for a monotone subdivision P with n vertices on N fixed lines is $O(n \log N)$.*

Sketch of Proof: There are $O(n)$ R-nodes, since gaps can be charged to vertices. Also, because of the segment-tree cutting scheme, there are $O(n \log N)$ O-nodes and ∇-nodes. Finally, each O-node contributes at most two ε-nodes. □

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

Sketch of Proof: Let $height(\tau)$ be the number of elementary slabs spanned by a trapezoid τ associated with a ∇-node of $\mathcal{T}(P)$. We show by induction that the depth of $\mathcal{T}(\tau)$ is at most $\log weight(\tau) + 3 \log height(\tau) + 1$. □

4.2 Hull Trees

The horizontal cuts in the decomposition of subdivision P induce a decomposition of each region into several subregions, each having spanning edges or spanning tangents. The set of left and right chains of the subregions of P are called the *elementary chains* of P. We dynamically maintain the convex hull of the elementary chains of P using a variation of the data structure of Overmars and van Leeuwen [12].

For each region r of P, we maintain two *hull trees*, denoted *lchain*(r) and *rchain*(r), representing the left and right chains of r, respectively. The secondary structures at the internal nodes of these trees are used to maintain the convex hulls of the elementary chains of P.

We describe the hull tree *lchain*(r) for the left chain γ of region r. Tree *rchain*(r) is symmetrically defined. We cut γ by a sequence of median lines, starting with $l_{\lfloor N/2 \rfloor}$, until for each resulting slab the portion of γ within that slab is a single edge spanning that slab. Let S be the set of points that are either vertices of γ or the intersections of γ with the cutting lines. Tree *lchain*(r) consists of a modification of the dynamic structure of [12] for the right convex hull of the points of S (see Fig. 4.). The leaves of *lchain*(r) are the points of S, where each point is duplicated. Each internal node ξ of *lchain*(r) is associated with a slab σ and a node μ of $\mathcal{T}(P)$, such that μ is the lowest node of $\mathcal{T}(P)$ whose trapezoid τ contains $\gamma \cap \sigma$. We establish a pointer from μ to ξ. Also, if $r \cap \tau$ spans τ, then node ξ of *lchain*(r) has a secondary structure (a balanced tree) that stores the points of the right convex hull of $\gamma \cap \tau$ (in bottom-to-top order), excluding the points that have been already stored in the secondary data structures of the ancestors of ξ.

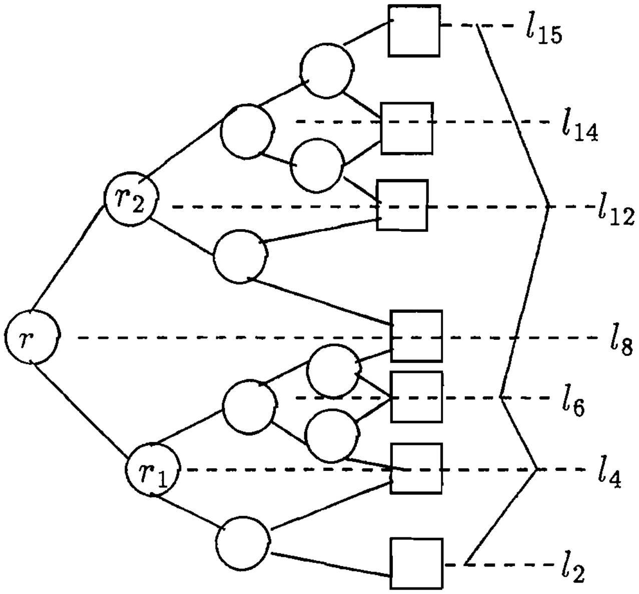

Figure 4: Example of hull tree for the left chain of a region ($N = 16$).

Lemma 1 *The hull trees for the left and right chains of the regions of P have each height $O(\log N)$, and use total space $O(n \log N)$. Also, the hull trees allow to compute in $O(\log n)$ time the spanning tangents of a trapezoid τ induced by a gap of τ.*

The dynamic operation of merging two right (left) convex hulls (corresponding to two consecutive left (right) subchains) into a single one is called *bridging*, and its inverse operation is called *de-bridging*. Extending the results of [12], each bridging/de-bridging can be performed in $O(\log n)$ time.

4.3 Query

Point location for a query point q is performed by tracing down a path in $\mathcal{T}(P)$. Let μ be the node currently visited.

1. If μ is an ε-node then we stop.

2. If μ is a ∇-node with median line l, we discriminate q against l. If q is below or on l then we go to the left child of μ, otherwise (q is above l) we go to the right child.

3. If μ is an R-node with spanning tangent t, we discriminate q against t. If q is on t or to the left of t, we set $s_R \leftarrow \emptyset$ and go to the left child of μ; else (q is to the right of t) we set $s_L \leftarrow \emptyset$ and go to the right child.

4. If μ is an O-node, let $\sigma = s_1 \cdots s_p$ be the corresponding sequence of spanning edges delimiting empty trapezoids. If q is to the left of s_1, then set $s_R \leftarrow s_1$ and go to the left child of μ. Else if q is to the right of s_p, then set $s_L \leftarrow s_p$ and go right. Else, by searching in the balanced tree of σ we find the two edges s_i and s_{i+1} between which q lies, set $s_L \leftarrow s_i$ and $s_R \leftarrow s_{i+1}$, and stop.

When the above process terminates, we know that the region r containing q is immediately to the left of edge s_R and/or to the right of edge s_L (one of these edges may not be defined). Region r can be determined in additional $O(\log N)$ time using the hull trees storing representatives of s_L or s_R in their leaves. Note that the "masking" action in Step 3 (where s_L or s_R is set to $\emptyset$) ensures us to use the information obtained from the final point-edge discrimination, and thus to report r correctly.

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

4.4 Insertion and Deletion of Edges

In this section we consider operation INSERTSEGMENT$(e, v_1, v_2, r; r_1, r_2)$. We briefly describe the update of the trapezoid tree $\mathcal{T}(P)$ and of the hull trees.

The edge-insertion algorithm is a recursive procedure. The actions performed at the current node μ, associated with trapezoid τ, depend on the type of node μ:

1. μ is an R-node corresponding to gap g with spanning tangent t.

   If e spans g, we transform μ into an O-node with spanning edge e. Else, we consider the following subcases:

   (a) e does not intersect t.

   Recursively call the algorithm on the left or right child of μ according to whether edge e is to the left or right of t, respectively.

   (b) e intersects t.

   Construct the other spanning tangent $\bar{t}$ of g. If e does not intersect $\bar{t}$, replace t with $\bar{t}$ in μ, and recursively call the algorithm on the left or right child of μ according to whether e is to the left or right of $\bar{t}$, respectively. If instead e intersects also $\bar{t}$, we have to perform a *horizontal cut*. The operation is illustrated in Fig. 5, where each of the σ_m's denotes an O-node or R-node, and t' and t'' result from cutting t with l. The operation essentially corresponds to replacing node μ and the leaves (∇-nodes) immediately to its left and right with a new ∇-node ν. (Special cases are omitted in this extended abstract). Note that the decomposition trees (biased search trees) T, T_1, and T_2 may need to be rebalanced. After the horizontal cut is performed, we recursively call the algorithm on the new ∇-node ν.

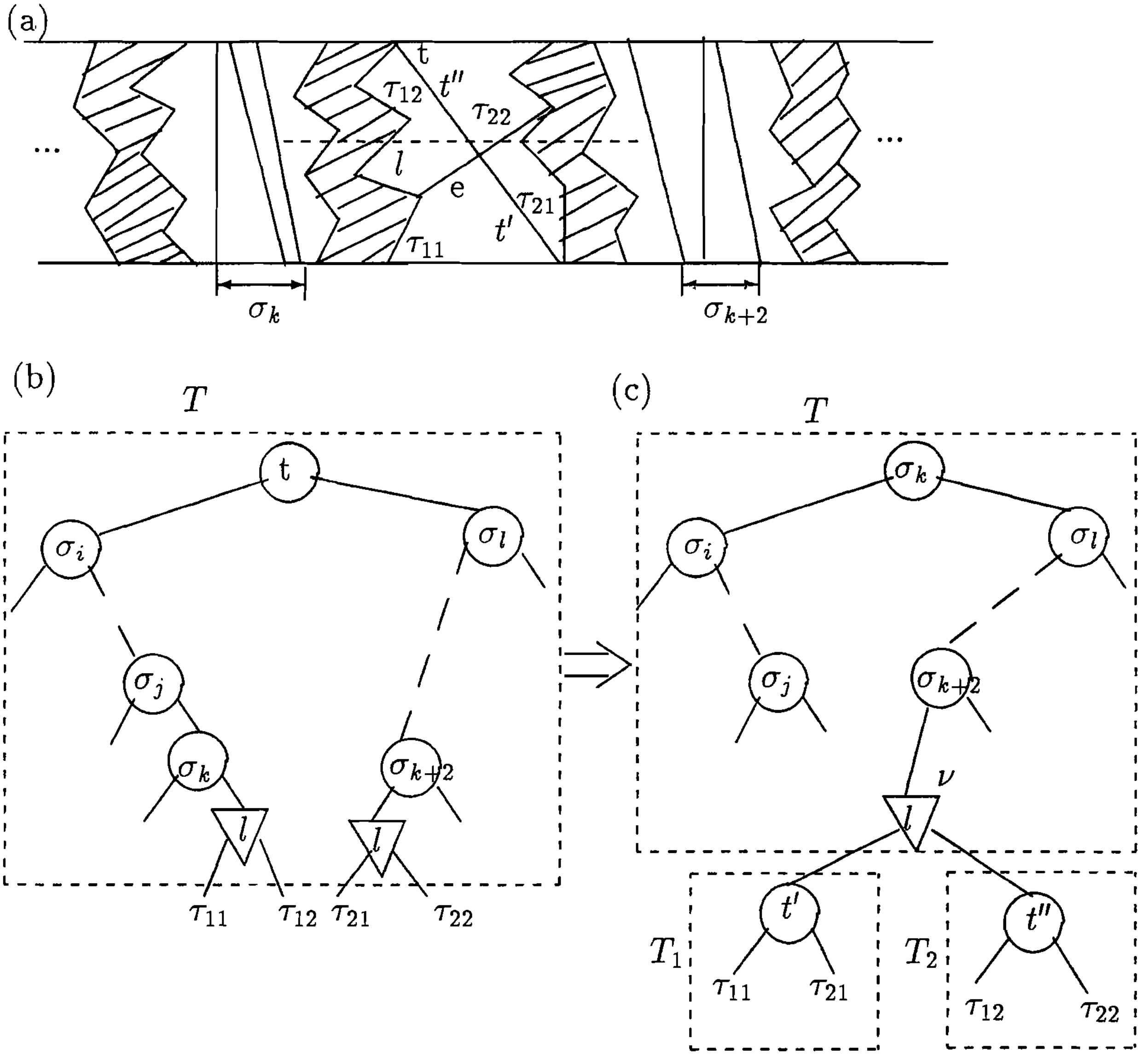

Figure 5: Operation of horizontal cut caused by inserting edge e

2. μ is an O-node with sequence of spanning edges $\sigma = s_1 ... s_p$.

   We distinguish three subcases:

   (a) e is to the left of s_1.

   If e spans τ and r (the region where e is inserted) is the region formerly to the left of s_1, then insert e into the balanced tree associated with σ in the leftmost position. Else recursively call the procedure on the left child of μ.

   (b) e is to the right of s_p.

   This case is analogous to the previous one.

(c) e lies between two edges of σ, say s_i and s_{i+1}. (In this case e must span τ.)
Insert e into the balanced tree of σ between s_i and s_{i+1}.

3. μ is a ∇-node with median line l.

In this case edge e cannot be a spanning edge of τ, or otherwise τ would have a spanning tangent/edge and thus μ would not be a ∇-node. Again, we have three subcases:

(a) e lies below l.
Recursively call the procedure on the left child of μ.

(b) e lies above l.
Recursively call the procedure on the right child of μ.

(c) e is cut by l into e_1 and e_2, with e_1 below e_2.
Recursively call the procedure twice to insert e_1 into the left child of μ and e_2 into the right child of μ.

The time-complexity analysis is based on the following lemma, which provides an upper bound on the number of horizontal cuts.

Lemma 2 *A horizontal cut at an R-node μ causes $O(\log N)$ additional horizontal cuts in the subtree of μ.*

Sketch of Proof: Let τ be the trapezoid associated with node μ, and t its spanning tangent. Let the median line l of τ cut τ into τ_B and τ_T, with τ_B below l and τ_T above l. After inserting edge e, at least one of τ_B and τ_T is spanned by t, depending on whether e intersects t above or below l. Thus, at most one of τ_B and τ_T needs to be horizontally cut again (recall that a horizontal cut occurs only when e intersects *both* spanning tangents). We conclude that the number of horizontal cuts is bounded by the number of ∇-nodes on a root-to-leaf path. □

Theorem 5 *Updating the trapezoid tree $\mathcal{T}(P)$ in consequence of operation* INSERTSEGMENT $(e, v_1, v_2, r; r_1, r_2)$ *can be done in time $O(\log n \log N)$.*

Proof: By the segment tree scheme the algorithm splits e into $O(\log N)$ fragments and allocates them into a set of O-nodes and R-nodes. At each such allocation node we either recompute a spanning tangent (R-node) or insert e into a balanced tree (O-node), which takes $O(\log n)$ time. In general, before the insertion of edge e, region r is horizontally partitioned into subregions $r_1, \ldots, r_m$, each having spanning edges or spanning tangents. Assume that the endpoints of e are in subregions r_i and r_j. Since e is a spanning edge for r_k, $i < k < j$, there cannot be horizontal cuts in r_k during operation INSERTSEGMENT. Thus, only the trapezoids of r_i and r_j may get horizontally cut. By Lemma 2, a total of $O(\log N)$ horizontal cuts are performed. Each horizontal cut takes time $O(\log n)$ to rebalance the decomposition trees T, T_1, and T_2. □

Theorem 6 *Updating the hull trees in consequence of operation* INSERTSEGMENT$(e, v_1, v_2, r; r_1, r_2)$ *can be done in time $O(\log n \log N)$.*

Operation REMOVESEGMENT is the inverse of INSERTSEGMENT and its algorithm is similar (there are *horizontal merges* instead of horizontal cuts).

Theorem 7 *Operations* INSERTSEGMENT *and* REMOVESEGMENT *can each be performed in* $O(\log n \log N)$ *time.*

4.5 Other Update Operations

The algorithm for operation INSERTPOINT($v, e; e_1, e_2$) is outlined as follows:

1. Use the query algorithm to find the O-node μ with sequence of spanning edges $\sigma = s_1 \cdots s_p$ such that $e = s_i$ for some i and v lies in the trapezoid τ of μ.
2. Perform a local restructuring at μ, which essentially corresponds to cutting e horizontally along the median line l of τ into e' and e'' respectively below and above l. Namely, we construct a trapezoid tree $\mathcal{T}_v$ whose root is a ∇-node containing l, and whose subtrees consist each of an O-node with two children ε-nodes, where the left O-node contains e' and the right one e''. Let $\mathcal{T}_L$ and $\mathcal{T}_R$ be the left and right subtrees of μ. The restructuring replaces μ with two O-nodes, μ_1 and μ_2, and tree $\mathcal{T}_v$(details and special cases are omitted).
3. If v lies on l then stop; otherwise, if v is below l, then recursively call the procedure on the left subtree of $\mathcal{T}_v$; if v is above l, recursively call it on the right subtree.

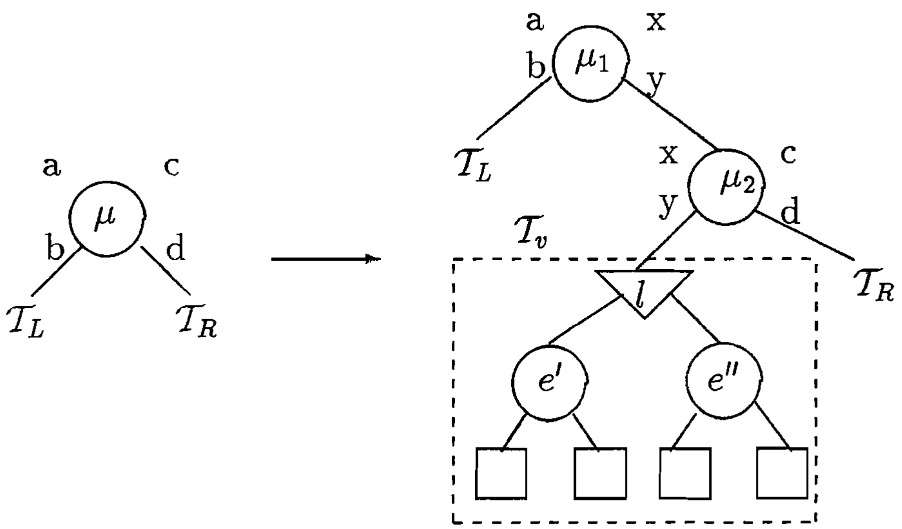

Figure 6: Restructuring of the trapezoid tree in operation INSERTPOINT.

Operation REMOVEPOINT is the inverse of INSERTPOINT, and is performed by a similar algorithm.

Theorem 8 *Operations* INSERTPOINT *and* REMOVEPOINT *can each be performed in time* $O(\log n \log N)$.

Finally, operation MOVEPOINT$(v; x)$ may destroy (or create) spanning tangents in the subregions to the left and right of v, which induces $O(\log N)$ horizontal cuts (or merges) of trapezoids.

Theorem 9 *Operation* MOVEPOINT$(v; x)$ *can be performed in time* $O(\log n \log N)$.

5 Data Structure for General Monotone Subdivisions

In the previous section we have described our dynamic data structure for the case when all n vertices of P lie on a fixed set of N horizontal lines. In this section we extend the technique to remove this constraint.

Without loss of generality we assume that no two vertices of P have the same y-coordinate and 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 P is $O(n)$, the total number of vertices after the expansion is still $O(n)$. Every update operation in the original subdivision P can be simulated with $O(1)$ operations in the modified subdivision with bounded-degree vertices.

5.1 Y-Tree

We make use of another type of binary search tree, called $BB[\alpha]$-tree [11]. Let α be a fixed real, with $\frac{1}{4} < \alpha \leq 1 - \frac{\sqrt{2}}{2}$. Some important properties of a $BB[\alpha]$-tree are listed below:

1. A $BB[\alpha]$-tree with n nodes has height $O(\log n)$.

2. Assume that we augment a $BB[\alpha]$-tree with secondary structures. Let the subtree with root v have k leaves, and let the time for updating the secondary structures after a rotation or double rotation at node v be $O(k \log k)$. Then the amortized rebalancing cost per insertion/deletion is $O(\log^2 n)$, when we perform a sequence of n insertions and deletions into an initially empty $BB[\alpha]$-tree.

We use a $BB[\alpha]$-tree, called *Y-tree* $Y(P)$, for storing the vertices of P sorted by y-coordinate. Each node v of $Y(P)$ also corresponds to the slab containing all the vertices in the subtree of v and bounded by the horizontal lines associated with the ancestor nodes of v immediately preceding and following v in symmetric order.

The trapezoid tree and hull trees of P are then constructed using the tree $Y(P)$ to determine the horizontal cutting scheme. Indeed, each ∇-node μ of the trapezoid tree is associated with the node v of $Y(P)$ such that the trapezoid τ of μ is horizontally cut by the horizontal line through vertex v. Also, the ∇-nodes forming τ_B and τ_T are associated with the left and right children of v in $Y(P)$, respectively. Note that many ∇-nodes of the trapezoid tree may correspond to the same node v of $Y(P)$. We let v have bidirectional pointers to such ∇-nodes, so that the space requirement for $Y(P)$ is $O(n \log n)$.

With these modifications, the trapezoid tree and the hull trees have height $O(\log n)$ and use overall space $O(n \log n)$.

5.2 Update Operations

Operations INSERTSEGMENT, REMOVESEGMENT and MOVEPOINT are performed essentially as before, except that every time we create/destroy a ∇-node we have to update the pointers to and from the corresponding node in $Y(P)$.

Regarding operation INSERTPOINT$(v, e; e_1, e_2)$, we insert v into $Y(P)$, and modify the algorithm of the previous section so that at the last recursion where the trapezoid τ of μ is within an elementary slab, the horizontal line going through v is added and taken as the median line of τ.

The rebalancing of the tree $Y(P)$ is performed by means of rotations. Suppose that a rotation of $Y(P)$ occurs at nodes u and v, where the former parent u becomes the child of v. This means that in the trapezoid decomposition process the horizontal cuts through v now take priority over the cuts through u. Let the trapezoids horizontally cut along line $y = y(u)$ be $\tau_1, ..., \tau_m$. The ∇-nodes of these trapezoids are pointed by node u of $Y(P)$. For each τ_i, we completely rebuild the trapezoid tree $\mathcal{T}(\tau_i)$ and the corresponding hull subtrees, according to the new decomposition sequence. This consists of performing a static pre-processing to construct each $\mathcal{T}(\tau_i)$.

Lemma 3 *If the subtree T_u rooted at node u of $Y(P)$ has k leaves, then the total space used by the trapezoid subtrees $\mathcal{T}(\tau_1), \cdots, \mathcal{T}(\tau_m)$ rooted at the ∇-nodes $\mu_1 \cdots \mu_m$ pointed by u is $O(k \log k)$.*

Sketch of Proof: Let k_i be the number of vertices inside trapezoid τ_i. We have that $\sum_{i=1}^{m} k_i = O(k)$. Also, since each vertex has bounded degree, and each edge stored at an O-node of $\mathcal{T}(\tau_i)$ is incident on some vertex inside τ_i, we have that the O-nodes of $\mathcal{T}(\tau_i)$ store representatives of $O(k_i)$ edges. Arguing as in the proof of Theorem 2, we conclude that the space used by $\mathcal{T}(\tau_i)$ is $O(k_i \log k)$. □

Lemma 4 *If node μ of $Y(P)$ has k leaves, then a single/double rotation at μ takes $O(k \log k)$ time to update the corresponding trapezoid subtrees and hull subtrees.*

Hence, applying the properties of $BB[\alpha]$ trees we have that the amortized cost of operations INSERTPOINT and REMOVEPOINT is $O(\log^2 n)$.

By combining the results of this and the previous section, we obtain the central result of our paper:

Theorem 10 *There exists a fully dynamic point location data structure for a monotone subdivision whose current number of vertices is n such that point location queries take time $O(\log n)$ and operations* INSERTSEGMENT, REMOVESEGMENT, MOVEPOINT, INSERTPOINT *and* REMOVEPOINT *take time $O(\log^2 n)$, where the bounds are amortized for* INSERTPOINT *and* REMOVEPOINT, *and worst-case for the other operations.*

Acknowledgment

I would like to thank my advisor Roberto Tamassia for his kind and helpful advices.

References

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

[2] 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.)

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

[4] H. Edelsbrunner, L.J. Guibas, and J. Stolfi, "Optimal Point Location in a Monotone Subdivision," *SIAM J. Computing*, Vol. 15, 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, Saarbrcken, Germany, 1985.

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

[7] M.T. Goodrich and R. Tamassia, "Dynamic Trees and Dynamic Point Location," *Proc. 23rd ACM Symp. on Theory of Computing*, 1991 (to appear).

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

[9] 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.

[10] E.M. McCreight, "Priority Search Trees," *SIAM J. on Comput.*, Vol. 14, 257–276, 1985.

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

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

[13] M. Overmars, *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science, Springer-Verlag, 1983.

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

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

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

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

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

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

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