Dynamization of the Trapezoid Method for Planar Point Location in Monotone Subdivisions

Yi-Jen Chiang and Roberto Tamassia

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-92-06
January 1992

Dynamization of the Trapezoid Method for Planar Point Location in Monotone Subdivisions *

Yi-Jen Chiang Roberto Tamassia

Department of Computer Science
Brown University
Providence, R. I. 02912–1910
yjc@cs.brown.edu rt@cs.brown.edu

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 (amortized for vertex insertion/deletion and worst-case for the others). 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.

*Research supported in part by the National Science Foundation under grant CCR-9007851, 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. A preliminary version of this paper was presented at the 7th ACM Symposium on Computational Geometry, North Conway, June, 1991.

1 Introduction

Planar point location is a fundamental geometric searching problem and has been extensively studied. Given a subdivision $\mathcal{S}$ of the plane into polygonal regions, we want to perform on-line queries that ask for the region of $\mathcal{S}$ containing a given query point. In the *static* case, where $\mathcal{S}$ is fixed, there are optimal techniques that achieve $O(\log n)$ query time using $O(n \log n)$ preprocessing time and $O(n)$ space [4, 9, 21], where n is the size of $\mathcal{S}$.

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. Also, time complexity bounds are worst-case unless otherwise stated.

The technique of Overmars [15], based on the segment-tree, deals with subdivisions whose regions have a bounded number of edges, such as triangulations. The update operations supported are inserting and deleting edges and isolated vertices. This method uses $O(n \log n)$ space and has $O(\log^2 n)$ query and update times (amortized for vertex insertion/deletion). Fries, Mehlhorn, and Naeher [5, 6] present a data structure for general subdivisions with $O(n)$ space, $O(\log^2 n)$ query time, and $O(\log^4 n)$ amortized update time, using an approach based on the static chain-method [10]. The update operations supported are inserting and deleting edges and isolated vertices. If only insertions are performed, the update time is reduced to $O(\log^2 n)$ amortized.

Preparata and Tamassia give two dynamic techniques for monotone and convex subdivisions, respectively [18, 19]. The data structure for monotone subdivisions [18] supports inserting vertices on edges, inserting monotone chains of edges, and their inverses. The space requirement is $O(n)$. The query time is $O(\log^2 n)$. Vertices can be inserted or deleted in time $O(\log n)$, and a monotone chain with k edges can be inserted or deleted in time $O(\log^2 n + k)$ (thus inserting or deleting a single edge takes $O(\log^2 n)$ time) [18]. All bounds are worst-case. The data structure for convex subdivisions [19] is based on the trapezoid method [16]. It further assumes that the vertices lie on a fixed set of N horizontal lines and achieves space $O(N + n \log N)$, query time $O(\log n + \log N)$, and update time $O(\log n \log N)$ [19].

Tamassia [22] presents a technique for point location in triangulations which allows a trade-off between query and update time. It is based on the dynamization methods for decomposable search problems [14]. The update operations are inserting a "star" (vertex and three edges) inside a region and swapping the diagonal of a convex quadrilateral formed by adjacent regions. It is shown that for any smooth nondecreasing integer function $b(n)$ with $2 \leq b(n) \leq \sqrt{n}$, there exists a dynamic point location data structure with $O(n)$ space, $O((\log^2 n)/\log b(n))$ query time, and $O((\log^2 n)b(n)/\log b(n))$ update time [22]. By setting $b(n) = 2$, we obtain $O(\log^2 n)$ query and update times. By setting $b(n) = \log n$, we obtain $O((\log^2 n)/\log\log n)$ query time and $O((\log^3 n)/\log\log n)$ update time. Finally, by setting $b(n) = n^\epsilon$, we obtain $O(\log n)$ query time and $O(n^\epsilon \log n)$ update time [22].

Cheng and Janardan [2] present two methods for dynamic planar point-location. Their first method achieves $O(\log^2 n)$ 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. Their second method achieves $O(\log n \log\log n + k)$ time for inserting/deleting monotone chains, at the expense of increasing vertex insertion/deletion time to $O(\log n \log\log n)$ and increasing the query time to $O(\log^2 n \log\log n)$. Both methods use $O(n)$ space and are based on a search strategy derived from priority search trees [11]. As emphasized by the authors, such methods are mainly of theoretical interest, because they involve rather complex manipulations of data structures.

Very recently, Goodrich and Tamassia [8] have shown how to dynamically maintain a monotone

subdivision so as to achieve $O(n)$ space, $O(\log^2 n)$ query time, $O(\log n)$ time for vertex insertion/deletion, and $O(\log n + k)$ time for the insertion/deletion of a monotone chain of k edges. Their method is based on a new optimal static point-location data structure that uses two interlaced spanning trees, one for the subdivision and one for its graph-theoretic dual, to answer queries. The link-cut trees of Sleator and Tarjan [20] are used to dynamize the method, which is relatively easy to implement. 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 [16]. The update 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 (amortized for vertex insertion/deletion and worst-case for the others). 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 [8] 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 [19] 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 [22] achieves the same query time but is limited to triangulations and has a substantially higher ($O(n^\epsilon \log n)$) update time.
- The semi-dynamic data structure of Goodrich-Tamassia [8] 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 [19]. 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 contains preliminary definitions and results. In Section 3 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 4, 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 vertices of r. A *planar subdivision* $\mathcal{S}$ is a partition of the plane into polygons, called the *regions* of $\mathcal{S}$. We allow unbounded region and vertices at infinity. A *monotone subdivision* $\mathcal{S}$ 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 [17].

We define the following update operations on a monotone subdivision $\mathcal{S}$:

INSERTVERTEX$(v, e; e_1, e_2)$: Split edge $e = (u, w)$ into 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 of 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 $e = (u, w)$.

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, with r_1 on the left of e and r_2 on the right of e.

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 left and right of e into a new region r. This operation is allowed only if the resulting subdivision is monotone.

MOVEVERTEX$(v; x)$: Translate a degree-two vertex v horizontally so that its x-coordinate becomes x. This operation is allowed only if it yields a topologically equivalent subdivision.

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

Theorem 1 *An arbitrary monotone subdivision $\mathcal{S}$ with n vertices can be assembled from the empty subdivision, and disassembled to obtain the empty subdivision, by a sequence of $O(n)$* INSERTVERTEX, REMOVEVERTEX, INSERTEDGE, REMOVEEDGE, *and* MOVEVERTEX *operations. Also, such a sequence can be computed from $\mathcal{S}$ in $O(n)$ time.*

Proof: Let $\mathcal{S}'$ be a triangulation of $\mathcal{S}$, which can be computed in $O(n)$ time [7]. As shown in [22], a triangulation with n vertices can be assembled/disassembled using a sequence of the following operations:

INSERTSTAR$(v, r; r_1, r_2, r_3)$: Add vertex v inside region r and edges between v and the three vertices of r, which is decomposed into new regions r_1, r_2, and r_3.

REMOVESTAR$(v, r_1, r_2, r_3; r)$: Remove degree-3 vertex v and its incident edges, which merges the three regions formerly containing v into a new region r.

SWAPDIAGONAL$(e; r_1, r_2)$: Let e be an edge such that the union of the two regions on the two sides of e is a convex quadrilateral. Remove e and reinsert it as the other diagonal of the quadrilateral, thus creating two new regions r_1 and r_2.

To assemble subdivision $\mathcal{S}$, we first construct $\mathcal{S}'$ by simulating each of the above operations with $O(1)$ operations of our repertory: for INSERTSTAR, assuming that a, b, c are the vertices of the triangular region (see Fig. 1(a)), we insert vertex v on (a, c), move vertex v, insert edges (a, c) and (b, v); REMOVESTAR is the inverse operation of INSERTSTAR; for SWAPDIAGONAL, let the two triangular regions be $\triangle abd$ and $\triangle bdc$, which share a common side (b, d) (see Fig. 1(b)). We simulate by removing edge (b, d) and then inserting edge (a, c). Next, we remove the triangulation edges of $\mathcal{S}'$ with a sequence of $O(n)$ REMOVEEDGE operations. □

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 [1].

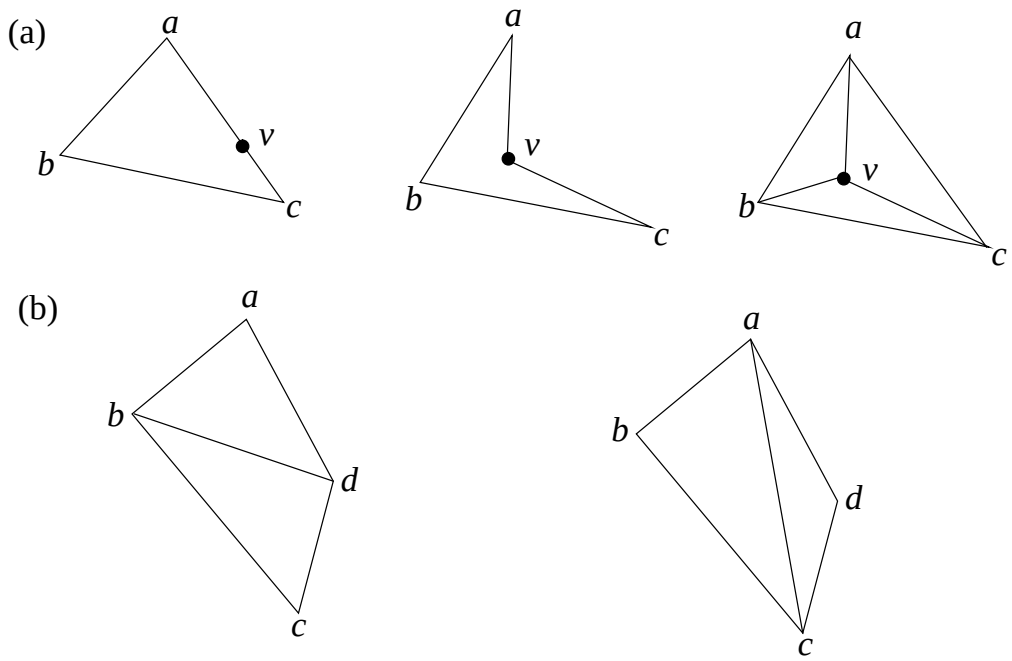

Figure 1: (a) Simulation of INSERTSTAR and REMOVESTAR; (b) simulation of SWAPDIAGONAL.

3 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 $\mathcal{S}$ lie on a fixed set $\mathcal{L}$ of N horizontal lines. Let $\mathcal{L} = \{l_1, l_2, \cdots, l_N\}$ be a set of horizontal lines, in this order from bottom to top, with l_1 defined by $y = -\infty$ and l_N defined by $y = +\infty$. The lines of $\mathcal{L}$ partition the plane into horizontal strips, called *elementary slabs*; the top most and bottom most elementary slabs are actually half planes. A *slab* is either an elementary slab or the union of two contiguous slabs.

3.1 Decomposing a Subdivision into Trapezoids

Our point location method is based on a recursive decomposition of the subdivision $\mathcal{S}$ into trapezoids, which is an extention of [16]. A *trapezoid* τ is a quadrilateral with two horizontal sides that lie on the lines of $\mathcal{L}$. 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 $\mathcal{S}$ is the trapezoid with the left and right sides at infinity and BOT($\mathcal{S}$) $= l_1$, TOP($\mathcal{S}$) $= l_N$ (thus all four sides are 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 $\mathcal{S}$ that spans τ is called a *spanning edge* of τ. Spanning edge e partitions τ into two trapezoids τ_L and τ_R, with $e =$ RIGHT(τ_L) $=$ LEFT(τ_R).

A *spanning region* of a trapezoid τ is a region r of $\mathcal{S}$ 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 a spanning region of τ and 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. 2(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. 2(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. 2(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).

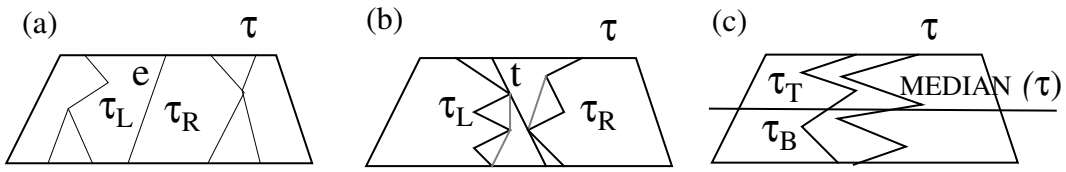

Figure 2: (a) vertical cut by a spanning edge; (b) vertical cut by a spanning tangent; (c) horizontal cut by the median line.

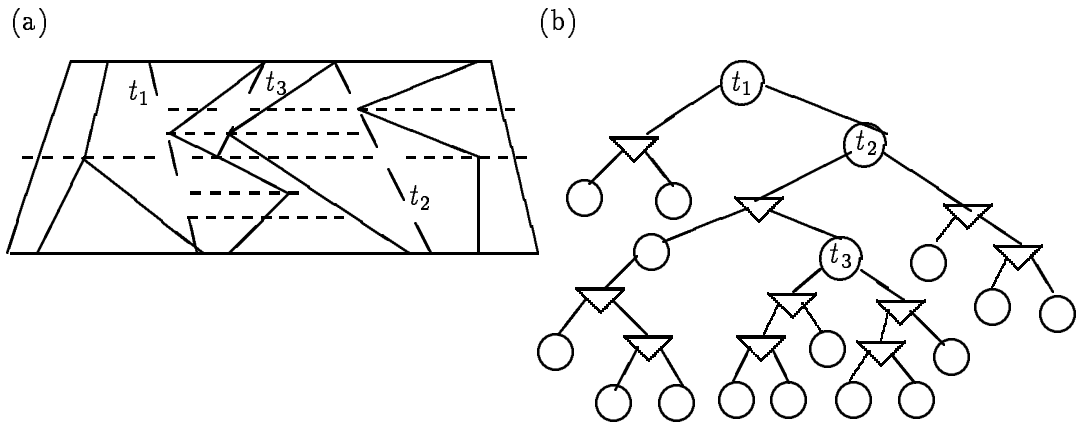

Figure 3: (a) Recursive decomposition of a trapezoid τ. (b) Trapezoid tree for τ with ε-nodes omitted.

The *canonical decomposition* of a trapezoid τ with spanning edges and/or 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. 4).

3.2 Trapezoid Tree

The main component of the point location data structure for $\mathcal{S}$ is called the *trapezoid tree*, which embeds many secondary structures called the *hull trees*. In addition, a dictionary stores the names

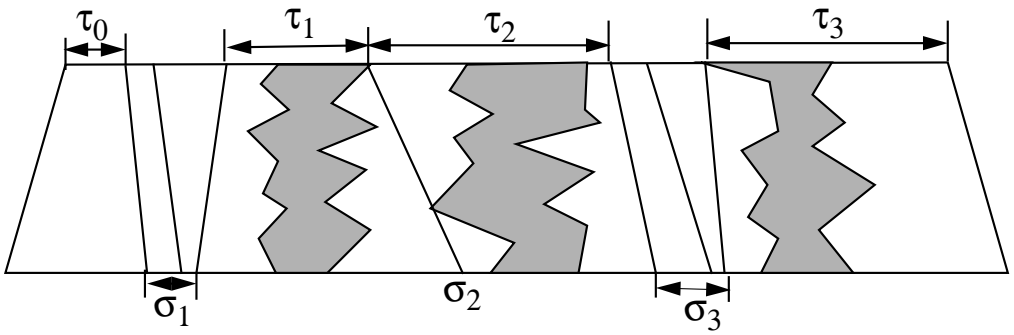

Figure 4: Canonical decomposition of a trapezoid

of vertices, edges, and regions, so that their associated substructures in the trapezoid tree can be efficiently accessed. We denote with n the current number of vertices of $\mathcal{S}$.

The *trapezoid tree*, denoted $\mathcal{T}(\mathcal{S})$, is a binary tree that represents the recursive decomposition of the subdivision $\mathcal{S}$ 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 excluding those on the boundary. There are four types of nodes: an ε-node corresponds to an empty trapezoid, and is always 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 terminology and symbols follow [16]).

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 $\mathcal{T}(\tau)$ 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 two pointers lc and rc (to nodes of hull trees, to be specified later); 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 (in tree inorder node sequence) 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, and if σ_i is a spanning tangent, node μ_i has two pointers lp and rp to the ∇-nodes of τ_{i-1} and τ_i, respectively. 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 (see Fig. 5). Note the difference between the *trapezoid tree* $\mathcal{T}(\tau)$ and the *decomposition tree* $T(\tau)$.

3.2.1 Embedding Hull Trees into the Trapezoid Tree

The horizontal cuts in the decomposition of subdivision $\mathcal{S}$ induce a decomposition of each region into several subregions, each having spanning edges or spanning tangents. The set of left and right

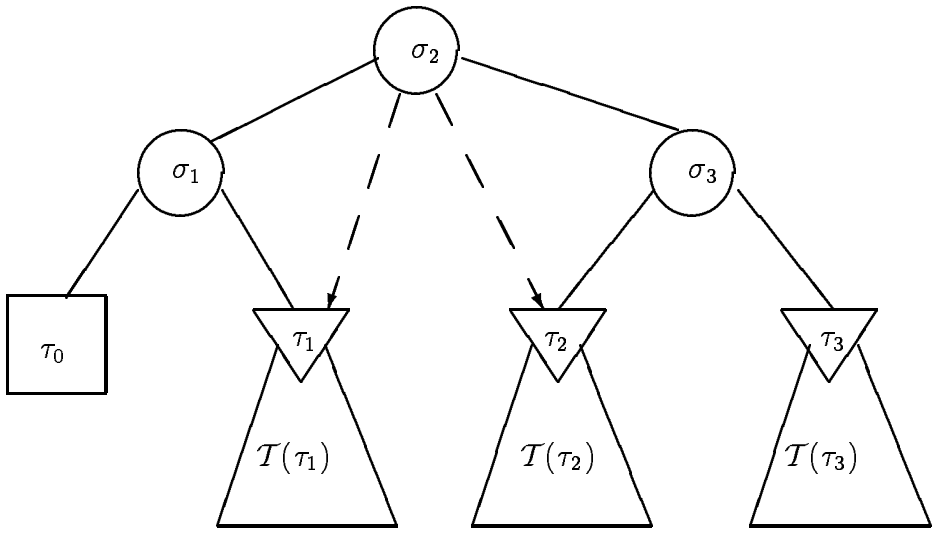

Figure 5: Trapezoid tree for the trapezoid of Fig. 4.

chains of the subregions of $\mathcal{S}$ are called the *elementary chains* of $\mathcal{S}$. We dynamically maintain the convex hull of the elementary chains of $\mathcal{S}$ using data structures called *hull trees*, which are a variation of the data structure of Overmars and van Leeuwen [13].

The hull trees are embedded as secondary structures in the trapezoid tree. Recall that each ∇-node μ of the trapezoid tree has two pointers lc and rc, and that μ corresponds to some trapezoid τ. Let $lc(\tau)$ and $rc(\tau)$ denote the left-most and right-most spanning chains of τ inside τ (they might have common edges or be coincident). We use two secondary nodes $lc(\mu)$ and $rc(\mu)$ to represent the chains $lc(\tau)$ and $rc(\tau)$, respectively. We describe $lc(\mu)$; $rc(\mu)$ is symmetrically defined. Node $lc(\mu)$ is a hull tree node, where the structure of the hull tree is defined according to the trapezoid tree. Let μ' and μ'' be the first ∇-node along the left-most root-to-leaf path in the left and right subtrees of μ, respectively; the left and right children of $lc(\mu)$ are respectively $lc(\mu')$ and $lc(\mu'')$. If there is no such μ' (resp. μ''), then the left (resp. right) child χ of μ is an O-node with an ε-node left child. We let χ be the left (resp. right) child of $lc(\mu)$ and a leaf of the hull tree. Each node $lc(\mu)$ of the hull tree also has a secondary structure (a balanced tree) that stores the points of the left convex hull of the left spanning chain $lc(\tau)$ of trapezoid τ (for leaf χ (an O-node) of the hull tree, the left spanning chain is the left-most spanning segment stored in χ), excluding those points that have already been stored in the secondary structures of the ancestors of $lc(\mu)$ in the hull tree.

Recall that each R-node ν also has two pointers lp and rp to some ∇-nodes ν' and ν'', respectively, and that ν corresponds to a gap g. Note that if t is the spanning tangent of g stored in ν and τ_1 and τ_2 are the trapezoides respectively corresponding to ∇-nodes ν' and ν'', then $\textsc{right}(\tau_1) = \textsc{left}(\tau_2) = t$. Thus the left boundary of gap g is the right spanning chain $rc(\tau_1)$ of τ_1 and the right boundary of g is the left spanning chain $lc(\tau_2)$ of τ_2. Given the hull trees as described, it is easy to see that $rc(\nu')$ is the root of one hull tree and its secondary structure stroes the complete right convex hull of the left boundary of g, and $lc(\nu'')$ is the root of another hull tree and its secondary structure stores the complete left convex hull of the right boundary of g.

For the purpose of update operations, we also store the *extreme-points* (a, b, c, d) in each node μ of the trapezoid tree (see Fig. 6):

(a) μ is a ∇-node associated with trapezoid τ.

a and b are the top and bottom points of the left-most spanning chain $lc(\tau)$ of τ inside τ, and c and d are the top and bottom points of the right-most spanning chain $rc(\tau)$, respectively.

(b) μ is an R-node corresponding to a gap g.

a and b are the upper-left and lower-left points of g, and c and d are the upper-right and lower-right points of g, respectively.

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

Let τ_1 and τ_2 be the trapezoids with $\text{RIGHT}(\tau_1) = s_1$ and $\text{LEFT}(\tau_2) = s_p$, respectively. If τ_1 is empty, then $a = b = \varepsilon$; otherwise a and b are the top and bottom points of the right-most spanning chain $rc(\tau_1)$ of τ_1. Similarly, if τ_2 is empty, then $c = d = \varepsilon$; else c and d are the top and bottom points of the left-most spaning chain $lc(\tau_2)$ of τ_2, respectively.

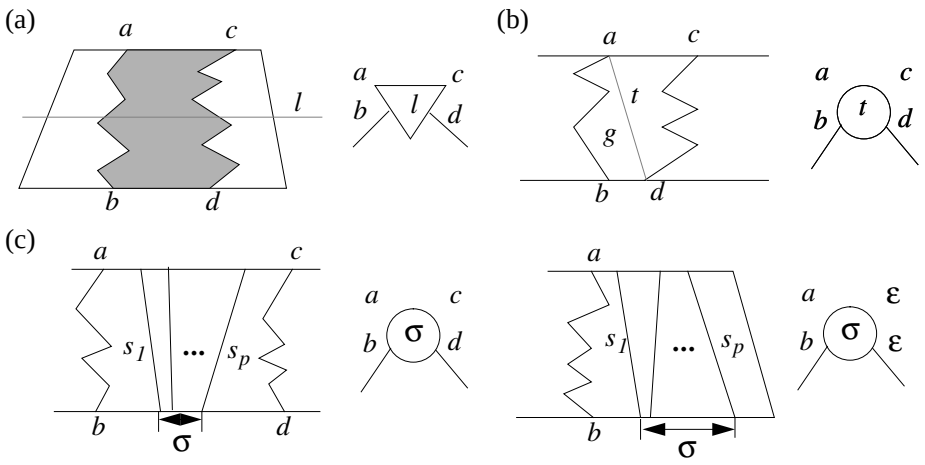

Figure 6: Examples of extreme-points (a, b, c, d): (a) for a ∇-node; (b) for an R-node; (c) for an O-node.

3.2.2 Properties of the Trapezoid Tree

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

Proof: Each R-node is associated with a gap; since a gap contains at least one vertex on its right chain and no gap is contained in the others, we can charge gaps to vertices. Hence there are $O(n)$ gaps and thus $O(n)$ R-nodes. For O-nodes, by the segment tree scheme, each edge is decomposed into $O(\log N)$ segments, so $O(n \log N)$ space suffices for the O-nodes and their sequences of spanning edges. For ∇-nodes, let the *count* of a ∇-node μ be the number of ∇-nodes from μ to the root, inclusive. By the median-line decomposition scheme, each ∇-node has count $O(\log N)$. Let λ be the first encountered ∇-node when walking up from a leaf, then λ corresponds to a trapezoid containing exactly one vertex. It follows that there are $O(n)$ such ∇-nodes, each with count $O(\log N)$, hence the total number of ∇-nodes is $O(n \log N)$. As for ε-nodes, each O-node contributes at most two ε-nodes as its children. Now consider the hull trees: each ∇-node is associated with two hull tree nodes, so there are $O(n \log N)$ hull tree nodes; for the secondary structures of the hull trees, we know that each edge of $\mathcal{S}$ is decomposed into $O(\log N)$ segments, so there are $O(n \log N)$ endpoints

of these segments. Each such endpoint appears twice in the secondary structures of overall hull trees, once as a left convex (sub-)hull point and once as a right convex (sub-)hull point. Therefore, the space complexity of the trapezoid tree $\mathcal{T}(\mathcal{S})$ is $O(n \log N)$. □

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

Proof: For simplicity, assume that N is a power of 2. Let $height(\tau)$ be the number of elementary slabs spanned by a trapezoid τ; obviously $height(\tau) \leq N$.

(1) For any ∇-node of $\mathcal{T}(\mathcal{S})$ corresponding to a trapezoid τ, we claim that the depth of the trapezoid tree $\mathcal{T}(\tau)$ is at most $\log weight(\tau) + 3 \log height(\tau) + 1$ (let $\log 0 = 1$ for notational simplicity).
We prove the claim by induction on $height(\tau)$. For the base case where $height(\tau) = 2$, the claim is trivially true since the left and right subtrees of $\mathcal{T}(\tau)$ consist each of an O-node with two children ε-nodes. For the induction step, observe that $\mathcal{T}(\tau)$ consists of a root and two subtrees $\mathcal{T}(\tau_B)$ and $\mathcal{T}(\tau_T)$; consider the deeper of the two, say $\mathcal{T}(\tau_B)$. The upper most part of the trapezoid tree $\mathcal{T}(\tau_B)$ is the decomposition tree $T(\tau_B)$; let τ_i be a leaf of $T(\tau_B)$, and let $w = weight(\tau)$, $w_i = weight(\tau_i)$, $h = height(\tau)$, and $h_i = height(\tau_i)$. Note that leaf τ_i is either a ∇-node or an ε-node. Since the decomposition tree $T(\tau_B)$ is a biased binary tree, the depth of the leaf τ_i in $T(\tau_B)$ is at most $\log(w/w_i) + 2$. By the induction hypothesis, the depth of the trapezoid tree $\mathcal{T}(\tau_i)$ is bounded by $\log w_i + 3 \log h_i + 1$, thus the depth of the trapezoid tree $\mathcal{T}(\tau)$ is at most $1 + (\log(w/w_i) + 2) + (\log w_i + 3 \log h_i + 1)$, which is $\log w + 3 \log h + 1$ since $h_i = h/2$. This completes the proof of the claim.

(2) For the trapezoid tree $\mathcal{T}(\mathcal{S})$, we have $weight(\mathcal{S}) = n$ and $height(\mathcal{S}) = N$. Hence, if the root of $\mathcal{T}(\mathcal{S})$ is a ∇-node, then we are done. Otherwise, $\mathcal{T}(\mathcal{S})$ consists of a decomposition tree $T(\mathcal{S})$ whose leaves are ∇-nodes or ε-nodes. Note that the height of the decomposition tree $T(\mathcal{S})$ is $O(\log n)$. Repeating the above argument to the subtrees of $\mathcal{T}(\mathcal{S})$ rooted at the leaves of $T(\mathcal{S})$, we have that the depth of $\mathcal{T}(\mathcal{S})$ is $O(\log n + \log N)$.

□

Lemma 1 *The hull trees allow to compute in $O(\log n)$ time the spanning tangents of a trapezoid τ induced by a gap of τ.*

Proof: By the segment tree scheme, each edge of $\mathcal{S}$ is cut into $O(\log N)$ segments, so there are $O(n \log N)$ points maintained in all hull trees and hence the secondary structures of the hull trees have each height $O(\log n)$. Let μ be an R-node of $\mathcal{T}(\mathcal{S})$ associated with gap g, and let μ' and μ'' be the ∇-nodes pointed by lp and rp of μ. As already described, the secondary structures of $rc(\mu')$ and $lc(\mu'')$ store the complete right convex hull of the left boundary of g and the complete left convex hull of the right boundary of g. Hence, a spanning tangent of a gap g is computed from such secondary structures in $O(\log n)$ time (see, e.g. [17]). □

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 [13], each bridging/de-bridging can be performed in $O(\log n)$ time.

3.3 Query

Point location for a query point q is performed by tracing down a path in $\mathcal{T}(\mathcal{S})$. 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 boundary trees for regions 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* $O(\log n + \log N)$.

Proof: Traversing a root-to-leaf path, we spend $O(1)$ time at each intermediate node, $O(\log n)$ time in the last node, and $O(\log N)$ time to compute region r from s_L or s_R. Hence, by Theorem 3, we have that a query takes $O(\log n + \log N)$ time. □

3.4 Insertion and Deletion of Edges

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

The edge-insertion algorithm consists of two phases: the first phase performs all necessary updates on the primal structure of the trapezoid tree $\mathcal{T}(\mathcal{S})$ and some work on the hull trees, and the second phase completes the updates on the hull trees.

The first phase is a recursive procedure. Starting from the root of $\mathcal{T}(\mathcal{S})$, 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 and extreme-points (a, b, c, d).

   If e spans g (namely $y(v_1) = y(b)$, $y(v_2) = y(a)$, $x(v_1) \in [x(b), x(d)]$, and $x(v_2) \in [x(a), x(c)]$), 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.

   Let ν' and ν'' be the ∇-nodes pointed by pointers lp and rp of μ, respectively. We construct the other spanning tangent $\bar{t}$ of g using the hull trees rooted at $rc(\nu')$ and $lc(\nu'')$. 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. 7, where each of the σ_m's denotes an O-node or R-node. The new spanning tangents t' and t'' for the lower and upper portions (with respect to l) of g, respectively, are obtained by first performing a de-bridging (and also a splitting of the hull trees) on both $rc(\nu')$ and $lc(\nu'')$, then computing the spanning tangents by the resulting hull trees. The remaining operation then essentially corresponds to replacing node μ, ν' and ν'' with a new ∇-node ν, where $lc(\nu)$ and $rc(\nu)$ are the original $lc(\nu')$ and $rc(\nu'')$, respectively. Special cases in the construction of subtree T_1 are schematically illustrated in Fig. 8 (and similarly for T_2). Note that the decomposition trees (biased search trees) T, T_1, and T_2 may need to be rebalanced according to the biased tree scheme. After the horizontal cut is performed, we recursively call the algorithm on the new ∇-node ν.

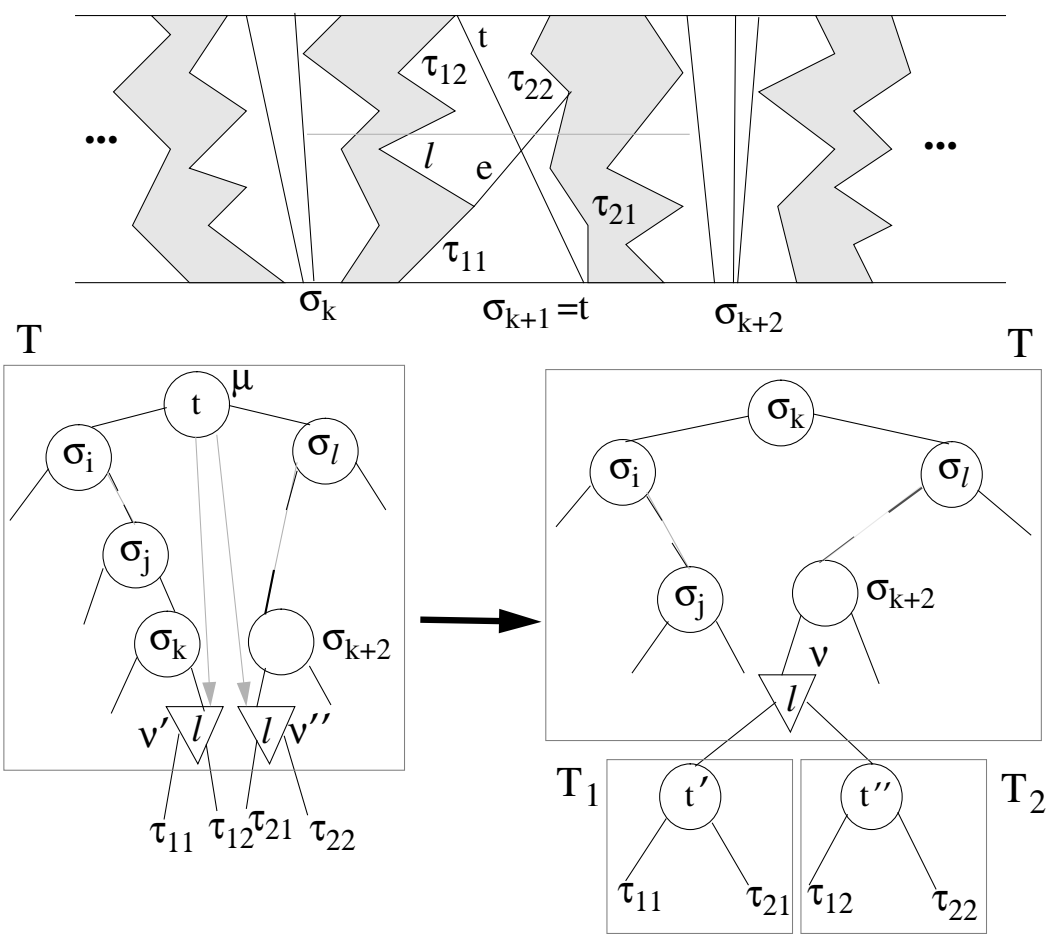

Figure 7: Operation of horizontal cut caused by inserting edge e

2. μ is an O-node with sequence of spanning edges $\sigma = s_1 ... s_p$ and extreme-points (a, b, c, d). We distinguish three subcases:

   (a) e is to the left of s_1.

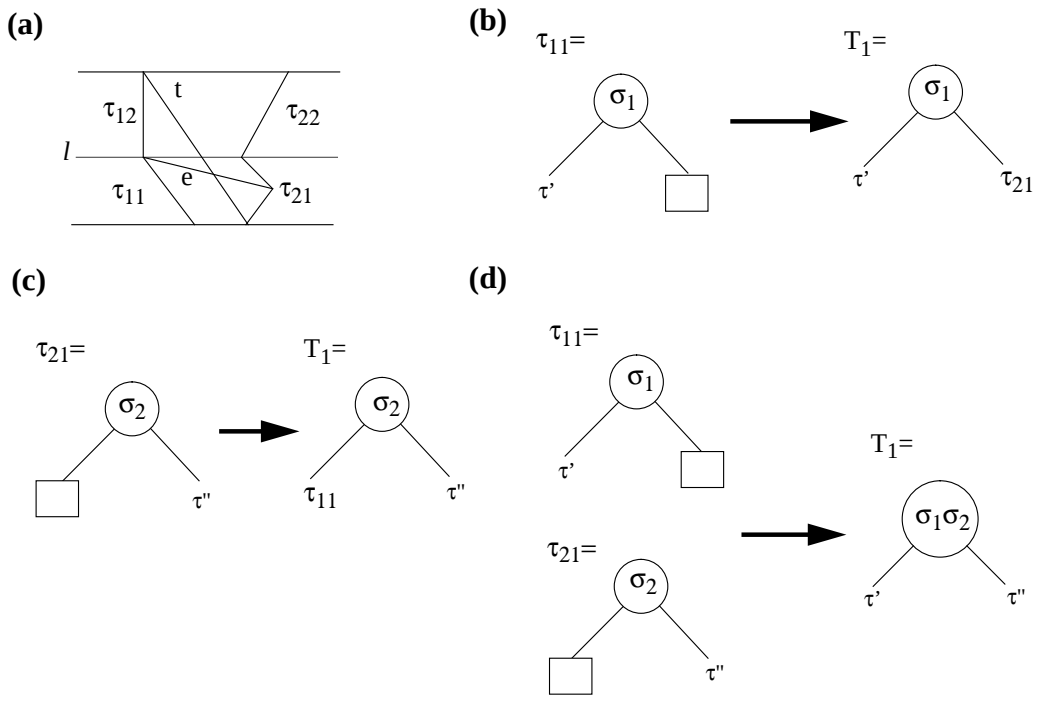

Figure 8: Special cases in the construction of T_1: (a) an example of the case in (b); (b) the root of τ_{11} is an O-node (thus its right subtree is an ε-node) while that of τ_{21} is not; (c) the case symmetric to (b); (d) both roots of τ_{11} and τ_{21} are O-nodes.

If e spans τ and r (the region where e is inserted) is the region formerly to the left of s_1(detected by either (i) $a = b = \varepsilon$, or (ii) $y(v_1) = y(b)$, $y(v_2) = y(a)$, $x(v_1) \geq x(b)$, and $x(v_2) \geq x(a)$), then insert e into the balanced tree associated with σ in the left-most 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 and extreme points (a, b, c, d).

First of all, mark μ as visited. If e spans τ (determined by $y(b) = y(v_1)$ and $y(a) = y(v_2)$), then τ must contain only a single spanning chain, i.e., $lc(\tau) = rc(\tau)$. (Or otherwise if τ has $lc(\tau) \neq rc(\tau)$ and e spans τ, then τ would have a spanning tangent/edge and thus μ would not be a ∇-node.) We create an R-node χ storing e, place χ at where μ was and make μ the left or right child of χ according to whether the single spanning chain $lc(\tau)$ is to the left or right of e (checked by the x-values of extreme points a and b). Also, if e is to the left (resp. right) of $lc(\tau)$, we break the link between $lc(\mu)$ (resp. $rc(\mu)$) and its parent λ in the hull tree, make χ a child of λ and thus a leaf of the hull tree.

Otherwise, if e does not span τ, we first consider the updates for the hull trees. The hull trees are changed only when e has an endpoint on either the top or bottom side of τ and

outside the region bounded by $lc(\tau)$ and $rc(\tau)$ (the other endpoint of e must be some point on $lc(\tau)$ or $rc(\tau)$). Specifically, if e touches the top or bottom side and is to the left of $lc(\tau)$ (i.e., either $y(v_2) = y(a)$ and $x(v_2) < x(a)$, or $y(v_1) = y(b)$ and $x(v_1) < x(b)$), we mark $lc(\mu)$, perform a de-bridging on $lc(\mu)$ *without* splitting the primal structure of the hull tree; similarly, if e touches the top or bottom side and is to the right of $rc(\tau)$ (i.e., either $y(v_2) = y(c)$ and $x(v_2) > x(c)$, or $y(v_1) = y(d)$ and $x(v_1) > x(d)$), we mark $rc(\mu)$, perform a de-bridging on $rc(\mu)$ without splitting the primal structure of the hull tree. After this hull tree update, we procede to update the primal structure of the trapezoid tree. 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 second phase procedes by walking up in the trapezoid tree: we traverse the path created from the first phase in reverse order (since we marked the ∇-nodes visited in the first phase, we know the entry points). For each ∇-node μ in the path, if $lc(\mu)$ is marked, we perform a bridging from its two children of the hull tree; if $rc(\mu)$ is marked, we perform the same operation.

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.*

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 may need to be horizontally cut (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 $\mathcal{T}(\mathcal{S})$ in consequence of operation* INSERTEDGE $(e, v_1, v_2, r; r_1, r_2)$ *takes $O(\log n \log N)$ time.*

Proof: We claim that each phase takes $O(\log n \log N)$ time. In the first phase, 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 $r_1, ..., r_m$, 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 INSERTEDGE. 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 perform de-bridgings, compute new spanning tangents t', t'', and rebalance the decomposition trees T, T_1, and T_2. Therefore the first phase takes $O(\log n \log N)$ time. In the second phase, there are $O(\log N)$ ∇-nodes traversed, each

takes $O(\log n)$ time for at most two bridging operations on the hull trees. So the second phase also takes $O(\log n \log N)$ time. □

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

Theorem 6 *Operation* REMOVEEDGE *takes* $O(\log n \log N)$ *time.*

3.5 Other Update Operations

The algorithm for operation INSERTVERTEX$(v, e; e_1, e_2)$ 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 v is a ∇-node containing l, and whose subtrees consist each of an O-node with two children ε-nodes, where the left O-node v' contains e' and the right O-node v'' contains e''. Also, both $lc(v)$ and $rc(v)$ have v' and v'' as their left and right children in the hull trees. 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 $\mathcal{T}_v$, as shown in Fig. 9. Also, if $e = s_1$ and $lc(\chi')$ was previously the parent of μ in some hull tree, we replace μ with $lc(v)$ as a child of $lc(\chi')$; similarly, if $e = s_p$ and $rc(\chi'')$ is the parent of μ in some hull tree, we replace μ with $rc(v)$ as a child of $rc(\chi'')$. Other details are given below.

3. If v lies on l then perform up-propagating rebalancings caused by the weight change, and 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.

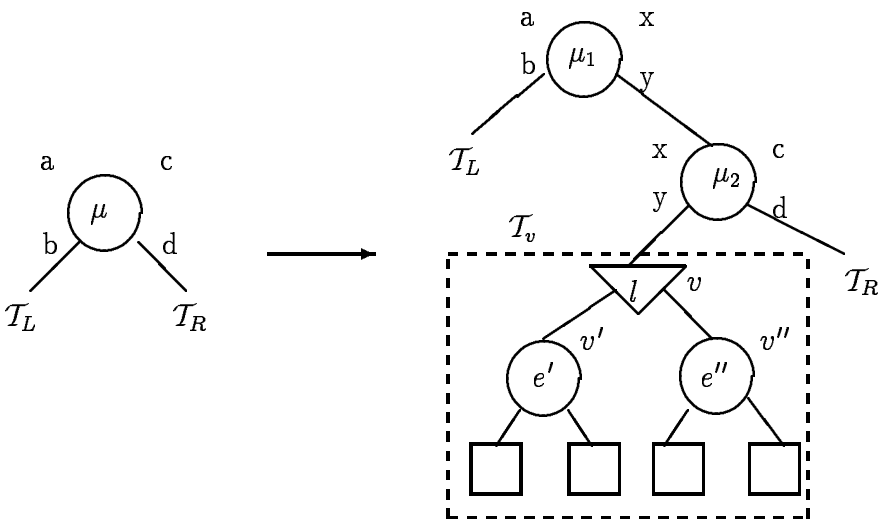

Figure 9: Restructuring of the trapezoid tree in operation INSERTVERTEX.

Now, we give the details of step 2. Let (a, b, c, d) be the extreme-points of μ, $e = (y, x)$, and $\sigma = \sigma_1 e \sigma_2$ ($\sigma_1 = \epsilon$ if $e = s_1$ and similarly for σ_2). Essentially, μ_1 corresponds to σ_1 and μ_2 to σ_2. We specify μ_1 as follows (μ_2 is defined symmetrically).

(a) $e = s_i$ with $i > 1$ (thus $\sigma_1 \neq \epsilon$).

μ_1 stores σ_1 and the extreme-points (a, b, x, y).

(b) $e = s_1$ (thus $\sigma_1 = \epsilon$).

If $a = b = \varepsilon$, then the subregion to the left of e is empty; we remove μ_1 and $\mathcal{T}_L$ altogether. Else, the subregion to the left of e now becomes a gap with extreme-points (a, b, x, y) and a spanning tangent $t = (y, a)$; we store t and (a, b, x, y) in μ_1.

Operation REMOVEVERTEX is the inverse of INSERTVERTEX, and is performed by a similar algorithm.

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

Proof: We analyze the time complexity of INSERTVERTEX: Step 1 takes $O(\log n + \log N)$ time; step 2 takes $O(\log n)$ time to split σ into σ_1 and σ_2, and to rebalance the decomposition tree containing μ_1 and μ_2. In step 3, the number of recursive calls is $O(\log N)$ since edge e can only be horizontally cut into $O(\log N)$ segments, and each recursion takes $O(1)$ time since the O-node μ now contains only one spanning edge. Finally, the propagaion of rebalancings at step 3 takes $O(\log n + \log N)$ time. Therefore, the total time for INSERTVERTEX is $O(\log n + \log N)$. The analysis of REMOVEVERTEX is analogous. □

Finally, operation MOVEVERTEX$(v; x)$ 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 8 *Operation* MOVEVERTEX$(v; x)$ *can be performed in time* $O(\log n \log N)$.

4 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 $\mathcal{S}$ 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 $\mathcal{S}$ 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 $\mathcal{S}$ is $O(n)$, the total number of vertices after the expansion is still $O(n)$. Every update operation in the original subdivision $\mathcal{S}$ can be simulated with $O(1)$ operations in the modified subdivision with bounded-degree vertices.

4.1 Y-Tree

We make use of another type of binary search tree, called $BB[\alpha]$-tree [12]. 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* $Y(\mathcal{S})$, for storing the vertices of $\mathcal{S}$ sorted by y-coordinate. Each node v of $Y(\mathcal{S})$ 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 of $\mathcal{S}$ is then constructed using the tree $Y(\mathcal{S})$ to determine the horizontal cutting scheme. Indeed, each ∇-node μ of the trapezoid tree is associated with node v of $Y(\mathcal{S})$ 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 $Y(\mathcal{S})$, respectively. Note that many ∇-nodes of the trapezoid tree may correspond to the same node v of $Y(\mathcal{S})$. We let v have bidirectional pointers to such ∇-nodes, so that the space requirement for $Y(\mathcal{S})$ is $O(n \log n)$.

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

4.2 Update Operations

Operations INSERTEDGE, REMOVEEDGE and MOVEVERTEX 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 $Y(\mathcal{S})$.

Regarding operation INSERTVERTEX$(v, e; e_1, e_2)$, we insert v into $Y(\mathcal{S})$, 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 $Y(\mathcal{S})$ is performed by means of rotations. Suppose that a rotation of $Y(\mathcal{S})$ 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 $Y(\mathcal{S})$. For each τ_i, we completely rebuild the trapezoid tree $\mathcal{T}(\tau_i)$ 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 $Y(\mathcal{S})$ 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 $O(k \log k)$.*

Proof: For each τ_i, consider the first encountered ∇-nodes of $\mathcal{T}(\tau_i)$ when walking up from leaves. Each such node λ corresponds to a trapezoid containing exactly one vertex v through which the median line goes, thus λ corresponds to node v of T_u. Since T_u has at most $2k - 1$ nodes, the total number of such v's from all $\mathcal{T}(\tau_i)$'s is at most $2k - 1$ and thus the union of $\tau_1, ..., \tau_m$ contains $O(k)$ vertices. Let k_i be the number of vertices in τ_i (thus the number of λ's in $\mathcal{T}(\tau_i)$). Define *count* of λ to be the number of ∇-nodes in the path from λ to the root of $\mathcal{T}(\tau_i)$, inclusive. The count of λ is $O(\log k)$ since T_u is a $BB[\alpha]$-tree with this height. So the total number of ∇-nodes in $\mathcal{T}(\tau_i)$ is $O(k_i \log k)$. For the O-nodes, note that τ_i has no spanning edges (otherwise the associated node of τ_i would not be a ∇-node), thus each edge in τ_i must be incident on some vertex inside τ_i. By the argument in the proof of Theorem 2, the total size of the O-nodes in $\mathcal{T}(\tau_i)$ is $O(k_i \log k)$. Again, by the argument in the proof of Theorem 2, the numbers of R-nodes and ε-nodes in $\mathcal{T}(\tau_i)$ are $O(k_i)$

and $O(k_i \log k)$, respectively, and the size of the overall hull subtrees in $\mathcal{T}(\tau_i)$ is $O(k_i \log k)$. Finally, recall that $\sum_{i=1}^{m} k_i = O(k)$, so the lemma follows. □

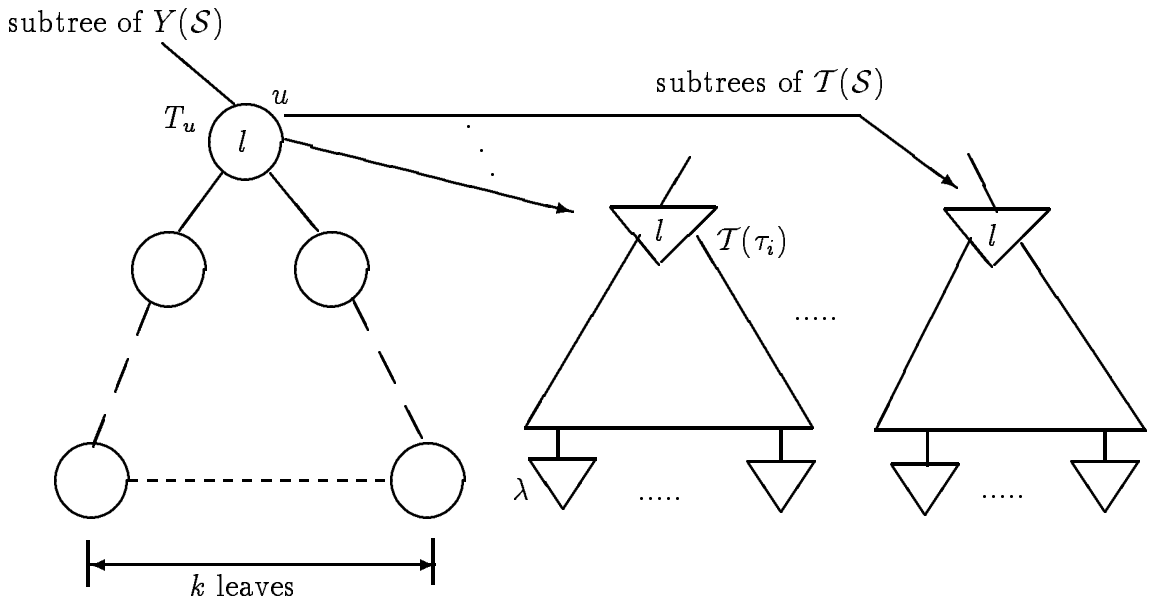

Figure 10: Proof of Lemma 3.

We specify the task to be carried out when a single rotation in $Y(\mathcal{S})$ occurs. Let τ_i and k_i be as defined in the above proof. For each τ_i, we rebuild the trapezoid tree $\mathcal{T}(\tau_i)$ as follows:

> After each horizontal cut, we must try to find all possible gaps so that the decomposition can proceed. Assume that originally τ_i was cut by l_u (going through vertex u) into τ_A and $\tau_B \cup \tau_C$, and the latter was cut by l_v (going through v) into τ_B and τ_C, where τ_A, τ_B, τ_C are in top-to-bottom order. After rotation between nodes u and v of $Y(\mathcal{S})$, τ_i is first cut by l_v into τ_C and $\tau_A \cup \tau_B$. At this stage, we must try to find a spanning tangent for *every* subregion in $\tau_A \cup \tau_B$ and in τ_C, since these were never encountered before. Then for the subregions in $\tau_A \cup \tau_B$ without spanning tangents, we cut them by l_u. Hence at most $4k_i$ subregions need to be tested; for each such subregion, we perform $O(1)$ de-bridgings and bridgings on the corresponding hull trees so that the two convex hulls of the left and right boundaries are available, then try to find the spanning tangent using the hull trees. If the subregion does not have a spanning tangent, then a horizontal cut on the subregion causes two de-bridgings on the two hull trees. This takes $O(\log k_i)$ time per subregion and thus total time $O(k_i \log k_i)$. The rest part is to rebuild $\mathcal{T}(\tau_i)$. Since the size is $O(k_i \log k)$ and all the information still exists, we spend the same amount of time to re-arrange the nodes.

Lemma 4 *If node u of $Y(\mathcal{S})$ has k leaves, then a single/double rotation at u takes $O(k \log k)$ time to update the corresponding trapezoid subtrees.*

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

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

Theorem 9 *There exists a fully dynamic point location data structure for a monotone subdivision whose current number of vertices is n such that the space complexity is $O(n \log n)$, point location queries take time $O(\log n)$ and operations* INSERTEDGE, REMOVEEDGE, MOVEVERTEX, INSERTVERTEX *and* REMOVEVERTEX *take time $O(\log^2 n)$, where the bounds are amortized for* INSERTVERTEX *and* REMOVEVERTEX, *and worst-case for the other operations.*

Acknowledgment

We would like to thank Franco Preparata for useful discussions.

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 CS 91-24, 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. Garey, D.S. Johnson, F.P. Preparata, and R. E. Tarjan, "Triangulating a Simple Polygon," *Information Processing Letters*, Vol. 7, No. 4, 175–180, 1978.

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

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

[10] 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.

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

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

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

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

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

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

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

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

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

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

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

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