International Journal of Computational Geometry & Applications, Vol. 2, No. 3 (1992) 311-333

DYNAMIZATION OF THE TRAPEZOID METHOD FOR PLANAR POINT LOCATION IN MONOTONE SUBDIVISIONS*

YI-JEN CHIANG
Department of Computer Science, Brown University
Providence, R.I. 02912–1910, U.S.A.
yjc@cs.brown.edu

and

ROBERTO TAMASSIA
Department of Computer Science, Brown University
Providence, R.I. 02912–1910, U.S.A.
rt@cs.brown.edu

Received 2 January 1992
Revised 15 August 1992

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 other updates). 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.

Keywords: Computational geometry, planar point location, dynamic data structure, monotone subdivision, analysis of algorithms.

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 [5,10,22], where n is the size of $\mathcal{S}$.

*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.

Research on dynamic algorithms for geometric problems has received increasing attention in the last years. The authors have recently written a survey on the subject [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 [16], 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. Fries, Mehlhorn, and Näher [6,7] 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 [11]. 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 [19,20] give two dynamic techniques for monotone and convex subdivisions, respectively. The data structure for monotone subdivisions [19] 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. The data structure for convex subdivisions [20] is based on the trapezoid method [17]. 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 $O(\log n \log N)$.

Tamassia [23] presents a technique for point location in triangulations which allows a tradeoff between query and update time. It is based on the dynamization methods for decomposable search problems [15]. The update operations are inserting a "star" (a 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 [23]. 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.

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 [12]. As emphasized by the authors, such methods are mainly of theoretical interest, because they involve rather complex manipulations of data structures.

Recently, Goodrich and Tamassia [9] 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 [21] 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 [17]. 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 other operations). 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:

- The fully dynamic data structures of Cheng-Janardan [2] and Goodrich-Tamassia [9] 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 [20] 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 [23] 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 [9] 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 [20] for convex subdivisions with vertices on a fixed set of N horizontal lines. 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 polygon r is said to be a *monotone polygon* if 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 regions 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 (e.g., on *segment-trees*), see the book by Preparata and Shamos [18].

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.

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.

As stated by the following theorem, 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 [8]. As shown by Tamassia [23], a triangulation with n vertices can be assembled/disassembled using a sequence of the following operations, such that each intermediate subdivision is also a triangulation:

INSERTSTAR($v, r; r_1, r_2, r_3$): Add vertex v inside region r, and add edges between v and the three vertices of r, decomposing r 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, merging the three regions formerly containing v (r_1, r_2, and r_3) 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. Next, we remove the triangulation edges of $\mathcal{S}'$ with a sequence of $O(n)$ REMOVEEDGE operations. The simulation is performed as follows. For INSERTSTAR($v, r; r_1, r_2, r_3$), let $r = abc$, where a is the bottommost vertex, and c is the topmost vertex (see Fig. 1(a)): we insert vertex v on edge (a, c), then move vertex v, and finally insert edges (a, c) and (b, v). REMOVESTAR is the inverse operation of INSERTSTAR. For operation SWAPDIAGONAL($e; r_1, r_2$), let $r_1 = abd$ and $r_2 = bdc$ (see Fig. 1(b)): we remove edge (b, d) and then insert edge (a, c). □

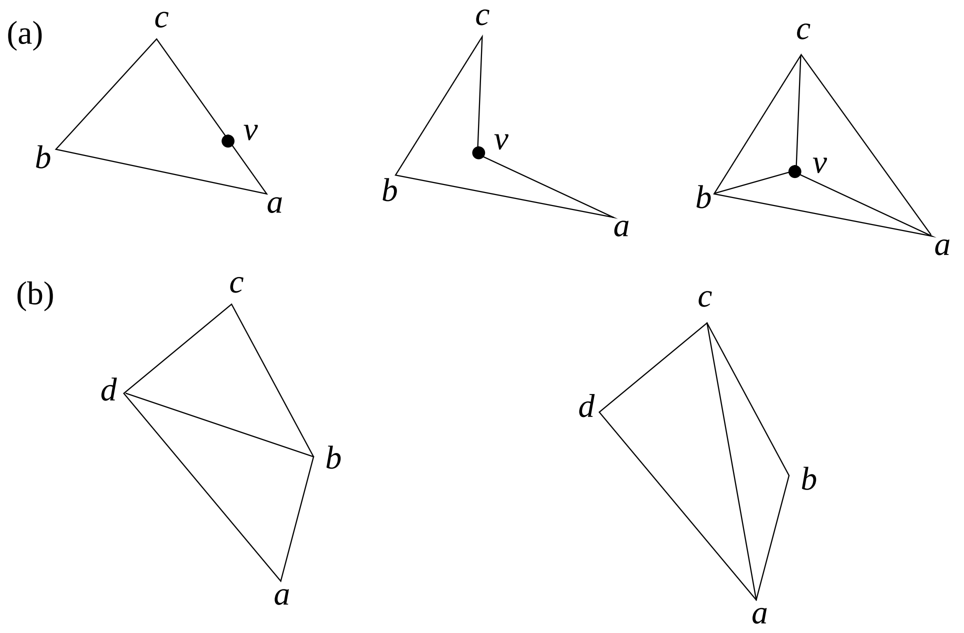

Fig. 1. (a) Simulation of INSERTSTAR and REMOVESTAR; (b) simulation of SWAPDIAGONAL.

The above repertory is minimal, i.e., removing any operation does not allow us to assemble or disassemble infinitely many subdivisions.

Regarding data structures, in the following we shall make use of a modified version of *biased binary trees*. Biased binary trees [1] are binary search trees whose leaves store items with positive weights. Let n be the number of items, and W be the total weight of the items stored in a biased binary tree T. We have:

- the space used by T is $O(n)$;
- the depth of a leaf with weight w is at most $\log(W/w) + 2$;
- the weight w of item i can be changed into $w + \delta$ in time $O(\log(W/w) + \log((W + \delta)/(w + \delta))$;
- a new item of weight w can be inserted in T in time $O(\log(W + w))$;
- splitting T into two biased binary trees, or splicing two biased binary trees to form T takes time $O(\log W)$.

In our modified biased binary trees the first and last item can have null weight. The introduction of such items increases the depth of the other items by at most 1, so that the depth of an item with weight w is at most $\log(W/w) + 3$, while the above space and time bounds remain unchanged.

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 of horizontal lines. Let $\{l_1, l_2, \cdots, l_N\}$ be a set of N horizontal lines, in this order from bottom to top, with l_1 defined by $y = -\infty$ and l_N defined by $y = +\infty$. These lines partition the plane into $N - 1$ horizontal strips, called *elementary slabs*; the topmost and bottommost 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 extension of [17]. A *trapezoid* τ is a quadrilateral with two horizontal sides that lie on $\{l_1, \cdots, l_N\}$. We use $\text{LEFT}(\tau)$, $\text{RIGHT}(\tau)$, $\text{BOT}(\tau)$, and $\text{TOP}(\tau)$ to denote the four sides of τ. Let $\text{BOT}(\tau)$ and $\text{TOP}(\tau)$ be on lines l_i and l_j, respectively, then the *median line* of τ, denoted by $\text{MEDIAN}(\tau)$, 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 $\text{BOT}(\mathcal{S}) = l_1$, $\text{TOP}(\mathcal{S}) = l_N$ (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 $\text{BOT}(\tau)$ and $\text{TOP}(\tau)$, 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 $e = \text{RIGHT}(\tau_L) = \text{LEFT}(\tau_R)$.

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) one can draw a segment inside r 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). 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. (This is one of the reasons for the efficiency of the trapezoid method: always cutting horizontally first would lead to the slab method [4], which has quadratic space requirement.) 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).

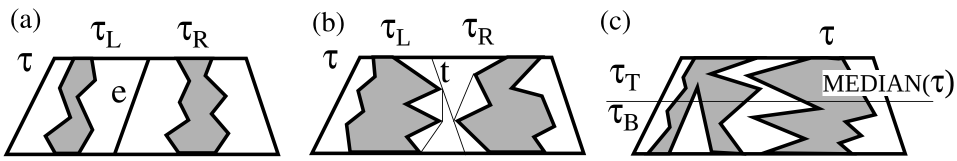

Fig. 2. (a) Vertical cut by a spanning edge. (b) Vertical cut by a spanning tangent. (c) Horizontal cut by the median line.

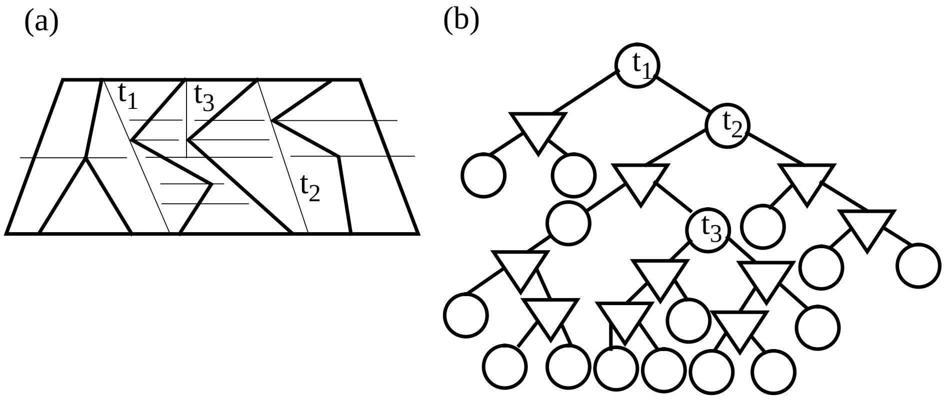

Fig. 3. (a) Recursive decomposition of a trapezoid τ. (b) Trapezoid tree for τ with ε-nodes omitted.

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).

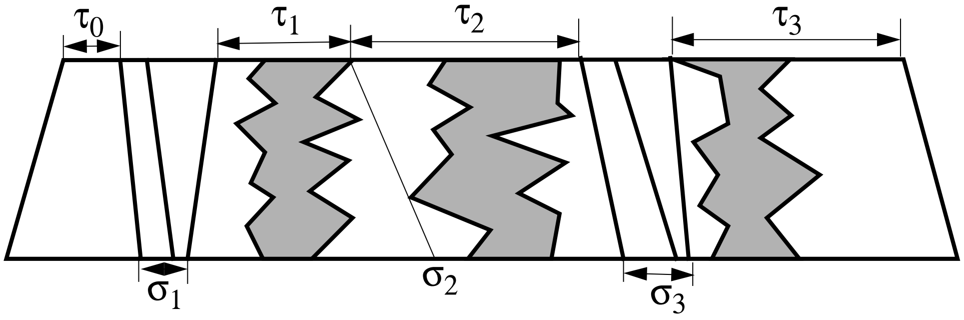

Fig. 4. Canonical decomposition of a trapezoid

3.2. Data Structure

We denote with n the current number of vertices of $\mathcal{S}$. The point location data structure consists of a main component, called *trapezoid tree*, and an auxiliary component, called *dictionary*.

3.2.1. Dictionary

The *dictionary* stores the names of vertices, edges and regions, so that their associated substructures in the trapezoid tree can be efficiently accessed. Also, as a part of the dictionary, a *boundary tree* (a balanced binary tree) is maintained for each region r to store the edges on its boundary, according to their circular sequence (starting at any edge). The root of the boundary tree stores the name of r. Let r_1 and r_2 be the regions on the left and right of edge e, respectively. Edge e has a *left representative* in the boundary tree of r_1, and a *right representative* in the boundary tree of r_2. We store with edge e pointers to these two representatives. Hence, given an edge e, we can determine the region immediately to the left (resp. right) of e by accessing the left (resp. right) representative of e and walking up to the root of its boundary tree. This takes $O(\log n)$ time since each region has $O(n)$ edges. As we shall see in Section 3.3, this scheme is used as a part of our query algorithm. Regarding updates on the boundary trees, adding or removing an edge corresponds to performing $O(1)$ split/splice operations on the boundary trees, which takes time $O(\log n)$; the same time bound holds for the updates on the boundary trees due to vertex updates. We conclude that all necessary operations on the dictionary can be performed in $O(\log n)$ time.

3.2.2. Trapezoid Tree

The *trapezoid tree* of $\mathcal{S}$, 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*, denoting the number of vertices inside τ, excluding those on the top and bottom sides of τ. 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 [17]).

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

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

- 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.

- 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 modified 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, then the i-th leaf of $T(\tau)$ is an ε-node with conventional zero weight. (Recall that our modified biased binary trees support such zero-weight items.) The i-th (in the inorder sequence) internal node μ_i of $T(\tau)$ represents σ_i. 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. If σ_i is a maximal sequence of spanning edges, then μ_i is an O-node, and σ_i is stored in a secondary structure as a balanced search tree. If σ_i is a spanning tangent t, then μ_i is an R-node, and $\sigma_i = t$ is stored at μ_i; also, μ_i has pointers *lp* and *rp* to the ∇-nodes of τ_{i-1} and τ_i, respectively. 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* $T(\tau)$.

3.2.3. Hull Trees

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 chains of the subregions of $\mathcal{S}$ are called the *elementary chains* of $\mathcal{S}$. We dynamically maintain the convex hull of the elementary

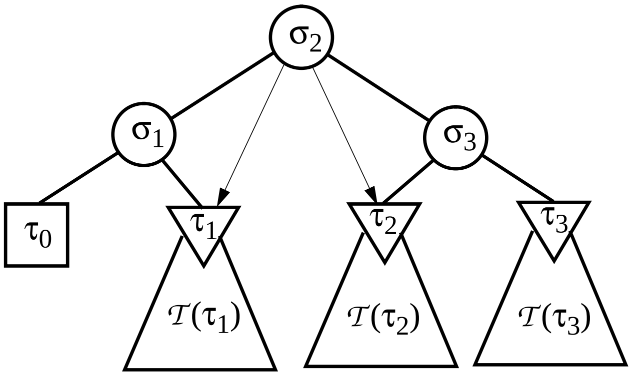

Fig. 5. Trapezoid tree for the trapezoid of Fig. 4. The lp and rp pointers of the root R-node are shown with arrows.

chains of $\mathcal{S}$ using data structures called *hull trees*, which are a variation of the data structure of Overmars and van Leeuwen [14].

The hull trees are embedded as secondary structures in the trapezoid tree (see Fig 6): the leaves of the hull trees are O-nodes; the internal nodes of the hull trees are pointed by the lc and rc pointers stored at the ∇-node. Let μ be a ∇-node, and τ be the trapezoid associated with μ. We denote with $lc(\tau)$ and $rc(\tau)$ the leftmost and rightmost spanning chains of τ (they might have common edges or be coincident). Hull-tree nodes $lc(\mu)$ and $rc(\mu)$ represent $lc(\tau)$ and $rc(\tau)$, respectively. We describe $lc(\mu)$; $rc(\mu)$ is symmetrically defined. Node $lc(\mu)$ has a secondary structure (a balanced tree) that stores the left convex hull of $lc(\tau)$, excluding the portions that have already been stored in the secondary structures of the ancestors of $lc(\mu)$. Let μ' and μ'' be the first ∇-nodes on the leftmost root-to-leaf paths in the left and right subtrees of μ, respectively. The left and right children of $lc(\mu)$ are $lc(\mu')$ and $lc(\mu'')$, respectively. If there is no such ∇-node μ', then the left child of $lc(\mu)$ is the last O-node on the leftmost root-to-leaf path in the left subtree of μ, and similarly if there is no such ∇-node μ''. Note that the left (right) child of $lc(\mu)$ is associated with the lower (upper) part of chain $lc(\tau)$.

The boundaries of gaps are associated with hull-trees as follows. Recall that each R-node ν corresponds to a trapezoid τ with a gap g, and stores a spanning tangent t of g and pointers lp and rp to two ∇-nodes. Let τ_1 and τ_2 be the trapezoids obtained by cutting τ along t, i.e., $\text{RIGHT}(\tau_1) = \text{LEFT}(\tau_2) = t$. The left boundary of gap g is chain $rc(\tau_1)$, and the right boundary of g is $lc(\tau_2)$. It is easy to see that node $rc(lp(\nu))$ is the root of a hull tree, and its secondary structure stores the complete right convex hull of the left boundary of g, and similarly for $lc(rp(\nu))$.

For the purpose of update operations, we also store at each node μ of the trapezoid tree the *extreme-points* (a, b, c, d) (see Fig. 7), defined as follows:

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

a and b are the top and bottom points of $lc(\tau)$, and c and d are the top and bottom points of $rc(\tau)$, respectively.

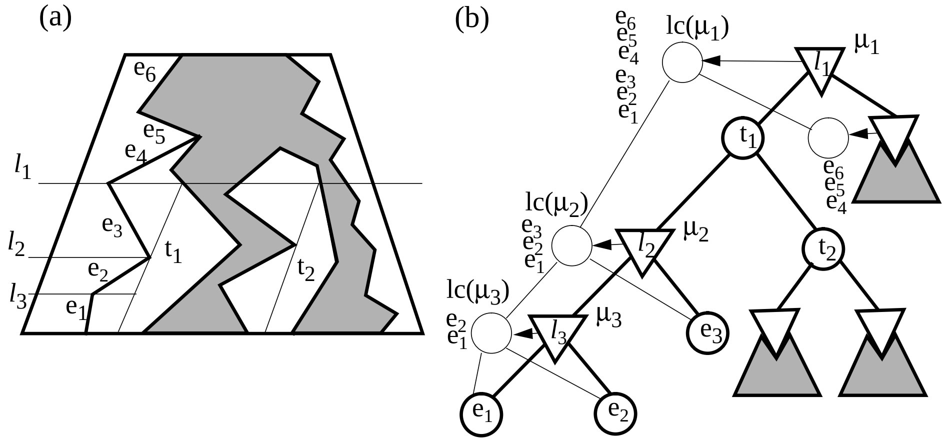

Fig. 6. Hull trees embedded in the trapezoid tree. (a) A trapezoid with leftmost spanning chain $\gamma = (e_1, e_2, e_3, e_4, e_5, e_6)$. (b) Portion of the hull tree for the left convex hull of γ.

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

a and b are the top-left and bottom-left points of g, and c and d are the top-right and bottom-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 $rc(\tau_1)$. Similarly, if τ_2 is empty, then $c = d = \varepsilon$; else c and d are the top and bottom points of $lc(\tau_2)$, respectively.

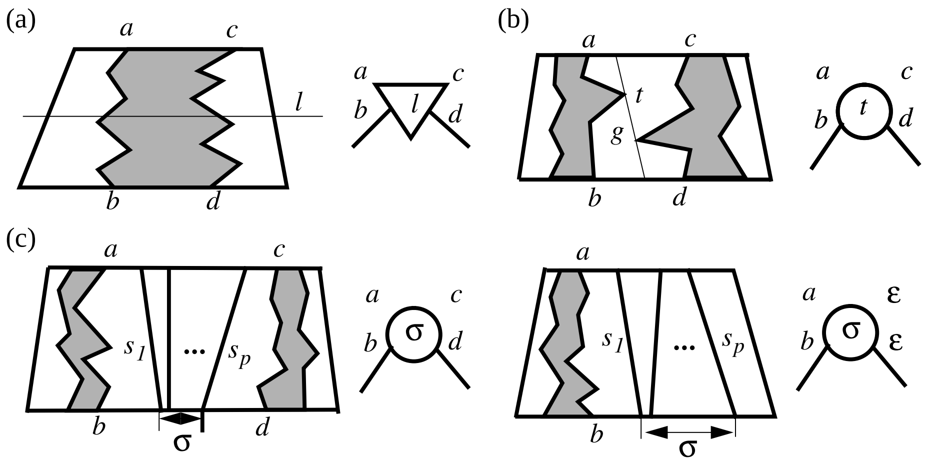

Fig. 7. Examples of extreme-points (a, b, c, d): (a) for a ∇-node; (b) for an R-node; (c) for an O-node.

3.2.4. Properties of the trapezoid tree

Lemma 1 *A path of the trapezoid tree $\mathcal{T}(\mathcal{S})$ has at most* $\log N$ ∇*-nodes.*

Proof. 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 $O(\log N)$, and the lemma follows. $\square$

Theorem 2 *The space complexity of the trapezoid tree and of the hull-trees for a monotone subdivision with n vertices on N fixed lines is $O(n \log N)$.*

Proof. Consider all the ∇-nodes with no descendant ∇-nodes. Such *basic ∇-nodes* correspond to trapezoids containing exactly one vertex, in their interior or on their left or right side (see, e.g., the rightmost ∇-node in Fig. 3). It follows that there are $O(n)$ basic ∇-nodes. Thus, by Lemma 1 the total number of ∇-nodes is $O(n \log N)$. Now, consider the ε-nodes, which are the leaves of the trapezoid tree. For each ∇-node μ there are at most four ε-nodes in the decomposition trees of the children of μ. Hence, the total number of ε-nodes is $O(n \log N)$. Since the trapezoid tree is a binary tree, we conclude that it has $O(n \log N)$ nodes. Each edge of the subdivision is decomposed into $O(\log N)$ fragments by the segment-tree scheme. Hence, the total size of the secondary structures stored at the O-nodes is $O(n \log N)$. Now consider the hull trees: each ∇-node is associated with two internal hull-tree nodes, so there are $O(n \log N)$ hull-tree nodes. Regarding the secondary structures of the hull trees, we know that each edge of the subdivision is decomposed into $O(\log N)$ fragments, so there are $O(n \log N)$ endpoints of these segments. Each such endpoint appears at most twice in the secondary structures of hull-tree nodes, once as a left convex hull point, and once as a right convex hull point. Therefore, the space complexity of the hull-trees is $O(n \log N)$. $\square$

Theorem 3 *The height of the trapezoid tree is $O(\log n + \log N)$.*

Proof. Let π be a longest leaf-to-root path in the trapezoid tree. Path π can be partitioned into subpaths $\pi_1, \cdots, \pi_k$, where π_1 consists of an ε-node followed by a maximal sequence of O-nodes or R-nodes, and π_i $(i = 2, \cdots, k)$ consists of a ∇-node followed by a maximal sequence of O-nodes or R-nodes. By Lemma 1, $k \leq \log N$. Also, since the decomposition tree rooted at the tail node μ_i of π_i is a modified biased binary tree, the number of nodes of subpath π_i is at most $\log(w_i/w_{i-1}) + 3$, where w_i denotes the weight of μ_i $(i = 1, \cdots, k)$, and $w_0 = 1$. Hence, the height of the trapezoid tree is given by:

$$\sum_{i=1}^{k} (\log(w_i/w_{i-1}) + 3) = O(\log w_k + \log N) = O(\log n + \log N).$$

$\square$

The hull trees allow us to efficiently compute the spanning tangents in the gap g of an R-node ν. Namely, since the balanced trees forming the secondary structures of hull-tree nodes $rc(lp(\nu))$ and $lc(rp(\nu))$ have $O(n)$ nodes, and hence $O(\log n)$ height, the two spanning tangents of g can be computed from such secondary structures in time $O(\log n)$ [18].

The dynamic maintenance of the hull trees is performed using the following primitive operations [14]: Let α be a hull-tree node, with children α' and α''. A *debridging operation* at node α consists of transferring to the secondary structures of α' and α'' the portions of convex hull stored at α. This consists of performing $O(1)$ split/splice operations and takes time $O(\log n)$. The inverse operation of debridging is called *bridging*, and also takes $O(\log n)$ time.

We summarize the above results as follows:

Lemma 2 *The operations of bridging, debridging, and computing the spanning tangents in a gap take each $O(\log n)$ time.*

3.3. Query

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

1. If μ is an ε-node then stop.

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

3. If μ is an R-node with spanning tangent t, discriminate q against t. If q is on t or to the left of t, set $s_R \leftarrow \emptyset$ and go to the left child of μ; else (q is to the right of t) 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, 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 right of edge s_L and/or to the left of edge s_R (one of these edges may not be defined). Region r can be determined in additional $O(\log n)$ time using the dictionary, by walking up to the root of the the boundary tree storing the right representative of s_L and/or the left representative of s_R. Note that the "masking" action in Step 3 (where s_L or s_R is set to $\emptyset$) ensures that we use the information obtained from the final point-edge discrimination, and thus 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. $\square$

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 that visits top-down a set of nodes. 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 trapezoid τ and gap g, with spanning tangent t and extreme-points (a, b, c, d).

   Consider the following subcases:

   (a) e spans g.
   This case arises when $y(v_1) = y(b)$, $y(v_2) = y(a)$, $x(v_1) \in [x(b), x(d)]$, and $x(v_2) \in [x(a), x(c)]$. Transform μ into an O-node with spanning edge e.

   (b) e does not span g and 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.

   (c) e does not span g and intersects t.
   Let $\nu' = lp(\mu)$ and $\nu'' = rp(\mu)$. Construct the other spanning tangent $\bar{t}$ of g using the secondary structures of $rc(\nu')$ and $lc(\nu'')$. We have two subcases:

      i. 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.

      ii. e intersects $\bar{t}$.
      Perform a *horizontal section*, which consists of replacing the vertical cut of τ with a horizontal cut (see Fig. 8). Nodes μ, ν' and ν'' are replaced by a new ∇-node ν, where $lc(\nu) = lc(\nu')$ and $rc(\nu) = rc(\nu'')$. If the rightmost node in the decomposition tree rooted at the left child of ν' and the leftmost node in the decomposition tree rooted at the left child of ν'' are both ∇-nodes, then the left child of ν is an R-node. Else, the left child of ν is an O-node (see Fig. 9). Analogous considerations hold for the right child of ν. Note that the horizontal section causes a deletion and a weight-change to be performed in the decomposition tree of node μ (tree T in Fig. 8), and two splice operations to be performed to construct the subtrees of ν (trees T_1 and T_2 in Fig. 8) from the subtrees of ν' and ν''. Regarding the hull trees, perform a debridging at nodes $rc(\nu')$ and $lc(\nu'')$, and remove such nodes, so that their children become new hull-tree roots, and their secondary structures are used to efficiently

compute the spanning tangents for the lower and upper portions of g (t' and t''in Fig. 8). After the horizontal section is performed, recursively call the algorithm on the new ∇-node ν.

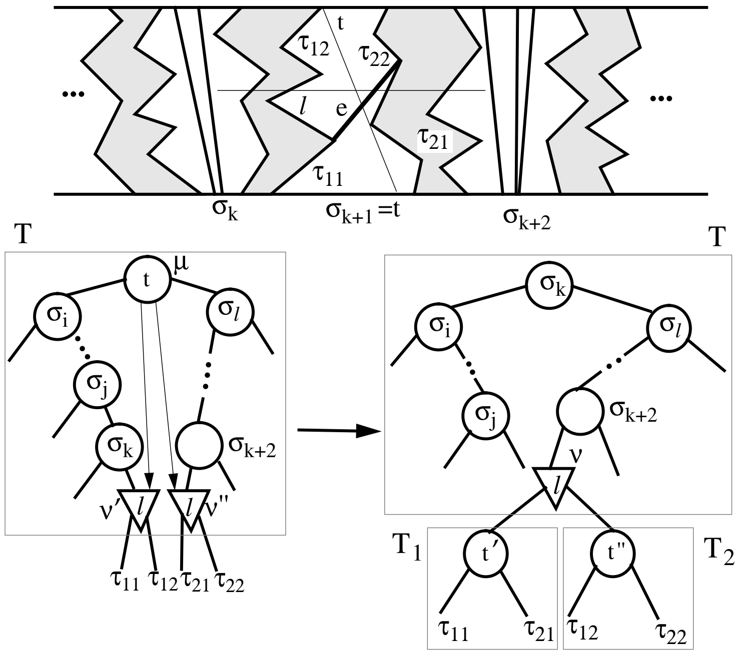

Fig. 8. Schematic illustration of a horizontal section.

2. μ is an O-node, corresponding to trapezoid τ, 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.

   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. This case is 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 $x(v_2) \geq x(a)$. 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 spans τ.)

   Insert e into the balanced tree of σ between s_i and s_{i+1}.

3. μ is a ∇-node, corresponding to trapezoid τ, with median line l and extreme points (a, b, c, d).

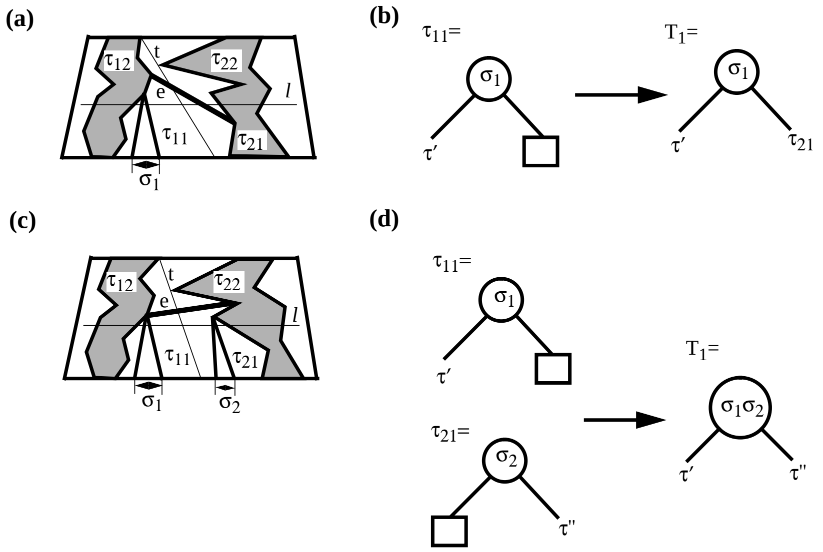

Fig. 9. Examples of horizontal sections where the left subtree T_1 of the newly created ∇-node ν is rooted at an O-node: (a–b) the rightmost node in the decomposition tree of τ_{11} is an ε-node, and the leftmost node in the decomposition tree of τ_{21} is a ∇-node; (c–d) the rightmost node in the decomposition tree of τ_{11} and the leftmost node in the decomposition tree of τ_{21} are ε-nodes.

We have two subcases:

(a) e spans τ.
In this case, determined by $y(b) = y(v_1)$ and $y(a) = y(v_2)$, e is either to the left of $lc(\tau)$ or to the right of $rc(\tau)$. (Otherwise e would be a spanning segment of τ, and μ would be an R-node or O-node.) Replace μ with a new O-node χ, storing e. If e is to the left of $lc(\tau)$, then make μ the right child of χ, create a new ε-node λ and make λ the left child of χ, remove the hull-tree edge between $lc(\mu)$ and its parent α, and make χ a child of α. Note that $lc(\mu)$ becomes a new hull-tree root. Else (e is to the right of $rc(\tau)$), perform symmetric operations on μ and $rc(\mu)$.

(b) e does not span τ.
If e is to the left of $lc(\tau)$ and has exactly one endpoint on $lc(\tau)$ (i.e., $y(v_2) = y(a)$ and $x(v_2) < x(a)$, or $y(v_1) = y(b)$ and $x(v_1) < x(b)$), mark $lc(\mu)$ and perform a debridging at $lc(\mu)$. Similarly, if e is to the right of $rc(\tau)$ and has exactly one endpoint on $rc(\tau)$, mark $rc(\mu)$ and perform a debridging at $rc(\mu)$. We distinguish three subcases:

i. e lies below l.
Recursively call the procedure on the left child of μ.

ii. e lies above l.

Recursively call the procedure on the right child of μ.

iii. 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 proceeds by visiting bottom-up the nodes visited in the first phase. For each ∇-node μ visited in the first phase, if $lc(\mu)$ is marked (Case 3b), we perform a bridging at $lc(\mu)$, and similarly for $rc(\mu)$.

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

Lemma 3 *A horizontal section at an R-node μ causes $O(\log N)$ additional horizontal sections in the subtree of μ.*

Proof. Let τ be the trapezoid associated with R-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 sectioned. Recall that a horizontal section occurs only when e intersects *both* spanning tangents (Case 1(c)ii). We conclude that the number of horizontal sections is bounded by the number of ∇-nodes on a root-to-leaf path, which is $O(\log N)$ by Lemma 1. □

Theorem 5 *Updating the trapezoid tree and the hull trees 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 cuts edge e into $O(\log N)$ fragments and stores them into the secondary structures of a set of $O(\log N)$ O-nodes, called the *allocation O-nodes* of e. The cutting takes place at $O(\log N)$ ∇-nodes (Case 3(b)iii), and the allocation of fragments to O-nodes takes place in Cases 1a, 2a, 2b, 2c, and 3a. For each allocation O-node μ, we spend $O(\log n)$ time to insert e in the secondary structure of μ, which yields $O(\log n \log N)$ total time spent at the allocation O-nodes. A total of $O(\log N)$ ∇-nodes are visited. At each such node $O(\log n)$ time is spent for debridging (Case 3b). We now discuss the time spent at R-nodes. A total of $O(\log n + \log N)$ R-nodes are visited. Handling Cases 1a and 1b takes $O(1)$ time. In Case 1c, the tangent computation takes $O(\log n)$ time (see Lemma 2). Also, in Case 1(c)ii, the horizontal section takes time $O(\log n)$ to restructure the affected decomposition trees, perform debridgings in the hull trees, and compute new spanning tangents. Hence, handling Case 1c takes $O(\log n)$ time. Regarding Case 1(c)i, it cannot arise more than once in the same decomposition tree because e can be in only one gap of the same canonical decomposition. Thus, Case 1(c)i occurs $O(\log N)$ times. To bound the number of occurrences of Case 1(c)ii, we observe that before the insertion of edge e, region r is horizontally partitioned into subregions $r_1, ..., r_m$, each having a vertical cut. 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 sections in r_k during operation INSERTEDGE. Thus, only the two trapezoids of r_i and r_j may get horizontally sectioned. By

Lemma 3, a total of $O(\log N)$ horizontal sections are performed. Hence, Case 1(c)ii occurs $O(\log N)$ time. We obtain that the total number of occurrences of Case 1c is $O(\log N)$. Therefore, we conclude that the first phase takes $O(\log n \log N)$ time.

In the second phase, there are $O(\log N)$ hull-tree nodes marked, and for each of them we spend $O(\log n)$ time for bridging. Thus, also the second phase 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 sections.

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. Let $\sigma = \sigma_1 e \sigma_2$.

2. Perform a local restructuring at μ (see Fig. 10), which corresponds to cutting e horizontally along the median line l of τ into e' and e'', respectively below and above l. First, create a tree consisting of a root ∇-node ν storing line l, and of subtrees formed each by an O-node with two children ε-nodes, where the left child ν' of ν stores e' and the right child ν'' stores e''. Nodes ν' and ν'' are also the left and right children of both $lc(\nu)$ and $rc(\nu)$ in the hull trees. Next, replace μ with a subtree consisting of two nodes, μ_1 and μ_2, associated respectively with σ_1 and σ_2, and of the tree rooted at ν, as shown in Fig. 10. If $i = 1$, then μ_1 is an R-node, else μ_1 is an O-node. If $i = p$, then μ_2 is an R-node, else μ_2 is an O-node.

3. If v lies on l then perform up-propagating restructurings in the decomposition trees ancestors of ν caused by the weight change, and stop. Otherwise, if v is below l, then recursively call the procedure on the left subtree of ν, else (v is above l) recursively call it on the right subtree.

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: By Theorem 4, step 1 takes $O(\log n + \log N)$ time. In step 2, $O(\log n)$ time is spent to split σ into σ_1 and σ_2, and to restructure the decomposition tree. The number of recursive calls is $O(\log N)$ since edges e_1 and e_2 are cut into $O(\log N)$ fragments. Processing each recursive call takes $O(1)$ time because the current node μ stores a single spanning edge and is in a decomposition tree of $O(1)$ size. The final restructuring due to the weight changes proceeds along a leaf-to-root path in the trapezoid tree. Let $\pi = \pi_1, \cdots, \pi_k$ be the path from node ν to the root, where each π_i consists of a

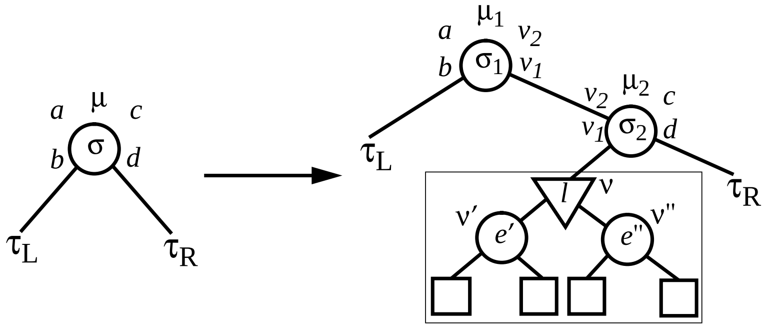

Fig. 10. Restructuring of the trapezoid tree in operation INSERTVERTEX. The bottom and top endpoints of edge e are v_1 and v_2, respectively.

∇-node followed by a maximal sequence of O-nodes or R-nodes. By Lemma 1, $k \leq \log N$. Denote with w_i the weight of the tail node of path π_i $(i = 1, \cdots, k)$, and let $w_0 = 1$. By the properties of biased binary trees, we have that the time for processing the weight changes is

$$O\left(\sum_{i=1}^{k} \log(w_i/w_{i-1}) + \log((w_i+1)/(w_{i-1}+1))\right) =$$

$$O(\log w_k + \log N) = O(\log n + \log N).$$

(Note the similarity with the proof of Theorem 3.) 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 sections 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 with degree greater than three 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 [13]. 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:

- A $BB[\alpha]$-tree with n nodes has height $O(\log n)$.
- A subtree of a $BB[\alpha]$-tree is itself a $BB[\alpha]$-tree.
- 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 time 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 $Y(\mathcal{S})$, called *Y-tree*, 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 $\mathcal{T}(\mathcal{S})$ is then constructed using the n horizontal lines through the vertices of $\mathcal{S}$ to delimit the elementary slabs (instead of the N fixed lines), and tree $Y(\mathcal{S})$ to determine the median line in horizontal cuts (instead of line l_m with $m = \lfloor (i+j)/2 \rfloor$). Indeed, each ∇-node μ of the trapezoid tree is associated with a 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 and the hull trees use overall space $O(n \log n)$ and have height $O(\log n)$, so that the query time is $O(\log n)$.

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 $Y(\mathcal{S})$. Their time complexity is $O(\log^2 n)$ (worst-case).

Regarding operation INSERTVERTEX$(v, e; e_1, e_2)$, we insert vertex v into $Y(\mathcal{S})$, and modify the algorithm of Section 3.5 so that at the last recursive call, 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 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 $\mu_1 \cdots \mu_m$ of these trapezoids are pointed by node u of $Y(\mathcal{S})$ (see Fig. 11). For each trapezoid τ_i, we completely rebuild the trapezoid tree $\mathcal{T}(\tau_i)$ rooted at μ_i and the associated hull subtrees according to the new decomposition sequence. This consists of performing a static pre-processing to construct each $\mathcal{T}(\tau_i)$.

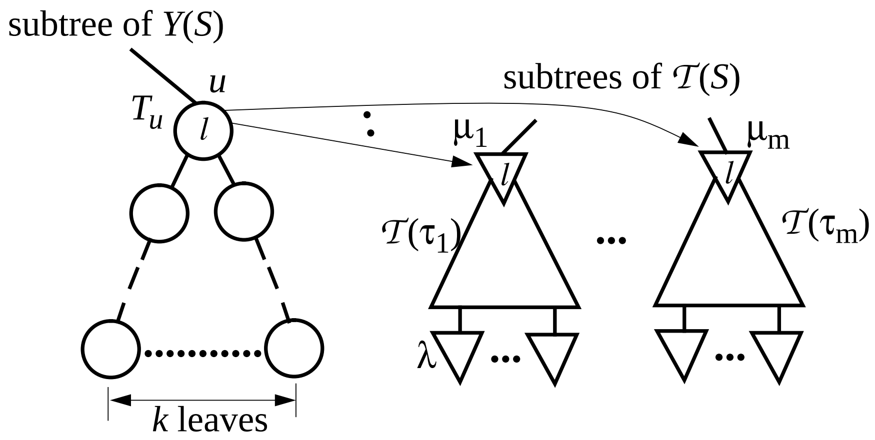

Fig. 11. Illustration of the proof of Lemma 4.

Lemma 4 *Let k be the number of leaves in the subtree T_u rooted at node u of $Y(\mathcal{S})$. The total space used by the trapezoid subtrees rooted at the ∇-nodes pointed by u and by their associated hull subtrees is $O(k \log k)$. Also, a single or double rotation at u takes $O(k \log k)$ time to rebuild such subtrees.*

Proof. We refer to Fig. 11 to visualize the proof, which is analogous to the one of Theorem 2. We consider the basic ∇-nodes in the subtrees rooted at $\mu_1, \cdots, \mu_m$, i.e., the ∇-nodes with no descendant ∇-nodes. Such basic ∇-nodes correspond to trapezoids containing exactly one vertex, in their interior or on their left or right side. Hence, each such basic ∇-node corresponds to a distinct vertex stored at a node of T_u. Since T_u has at most $2k - 1$ nodes, the total number of such basic ∇-nodes 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, which is equal to the number of basic ∇-nodes in $\mathcal{T}(\tau_i)$. Since T_u is a $BB[\alpha]$-tree with height $O(\log k)$, by a straightforward extension of Lemma 1 the total number of ∇-nodes in $\mathcal{T}(\tau_i)$ is $O(k_i \log k)$. Thus, by an argument similar to the one in the proof of Theorem 2, tree $\mathcal{T}(\tau_i)$ has $O(k_i \log k)$ nodes. Regarding the space requirement of the secondary structures of the O-nodes and hull-tree 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. Since each vertex has bounded degree, there are $O(k_i)$ edges in τ_i, and hence, by the segment-tree scheme, we have that $O(k_i \log k)$ total space is used for the secondary structures of the O-nodes in $\mathcal{T}(\tau_i)$, and of the hull-tree nodes associated with ∇-nodes of $\mathcal{T}(\tau_i)$. Finally, since $\sum_{i=1}^{m} k_i = O(k)$, we obtain the stated space bound.

The rebuilding of each trapezoid tree $\mathcal{T}(\tau_i)$ proceeds as follows. Let $\mathcal{S}_i$ be the portion of $\mathcal{S}$ inside τ_i, and let k_i be the number of vertices of $\mathcal{S}$ inside τ_i. Since μ_i is a ∇-node, all the edges of $\mathcal{S}_i$ have at least one endpoint vertex inside τ_i. Hence, because each vertex has bounded degree, the number of edges of $\mathcal{S}_i$ is $O(k_i)$. Subdivision $\mathcal{S}_i$ can be constructed in time $O(k_i)$ by a partial visit of $\mathcal{S}$ that starts at the vertices inside τ_i (such vertices are associated with the basic ∇-nodes in the subtree of μ_i). We construct the decomposition trees of the children of μ_i by traversing all the regions of $\mathcal{S}_i$. Since the chains bounding the regions of $\mathcal{S}_i$ are monotone, the convex-hull computations for building the hull-trees and determining the gaps take linear time. Hence, the decomposition trees of the children of μ_i use $O(k_i)$ space and are built in time $O(k_i)$. We complete the construction of $\mathcal{T}(\tau_i)$ by applying this procedure recursively on the ∇-nodes leaves of these decomposition trees. Hence, by charging $O(\log k)$ time to each vertex inside τ_i, we obtain that the time to rebuild $\mathcal{T}(\tau_i)$ is $O(k_i \log k)$. Since $\sum_{i=1}^{m} k_i = O(k)$, we obtain the stated result. $\Box$

Hence, by Theorem 7, Lemma 4, and the properties of $BB[\alpha]$-trees, we conclude that the amortized time complexity 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* 14, 545–568, 1985.
2. S.W. Cheng and R. Janardan, "New Results on Dynamic Planar Point Location," *Proc. 31st IEEE Symp. on Foundations of Computer Science*, 96–105, 1990.
3. Y.-J. Chiang and R. Tamassia, "Dynamic Algorithms in Computational Geometry," *Proceedings of the IEEE, Special Issue on Computational Geometry*, ed. G. Toussaint, 1992. Also available as Technical Report CS 91-24, Dept. of Computer Science, Brown Univ., 1991.
4. D. Dobkin and R. Lipton, "Multidimensional Searching Problems," *SIAM J. Computing* 5, 181–186, 1976.
5. H. Edelsbrunner, L.J. Guibas, and J. Stolfi, "Optimal Point Location in a Monotone Subdivision," *SIAM J. Computing* 15, 317–340, 1986.

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

7. O. Fries, K. Mehlhorn, and S. Näher, "Dynamization of Geometric Data Structures," *Proc. 1st ACM Symp. on Computational Geometry*, 168–176, 1985.

8. M. Garey, D.S. Johnson, F.P. Preparata, and R. E. Tarjan, "Triangulating a Simple Polygon," *Information Processing Letters* 7, 175–180, 1978.

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

10. D. Kirkpatrick, "Optimal Search in Planar Subdivision," *SIAM J. Computing* 12, 28–35, 1983.

11. D.T. Lee and F.P. Preparata, "Location of a Point in a Planar Subdivision and its Applications," *SIAM J. Computing* 6, 594–606, 1977.

12. E.M. McCreight, "Priority Search Trees," *SIAM J. Computing* 14, 257–276, 1985.

13. K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, 189-199, 1984.

14. M.H. Overmars and J. van Leeuwen, "Maintenance of Configurations in the Plane," *J. Computer Systems Sciences* 23, 166–204, 1981.

15. M.H. Overmars, *The Design of Dynamic Data Structures*, Lecture Notes in Computer Science 156, Springer-Verlag, 1983.

16. M..H. Overmars, "Range Searching in a Set of Line Segments," *Proc. 1st ACM Symp. on Computational Geometry*, 177–185, 1985.

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

18. F.P. Preparata and M.I. Shamos, *Computational Geometry*, Springer-Verlag, NY, 1985.

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

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

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

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

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