Dynamic Trees and Dynamic Point Location

Michael T. Goodrich*
Dept. of Computer Science
The Johns Hopkins University
Baltimore, MD 21218
Email: goodrich@cs.jhu.edu

Roberto Tamassia†
Dept. of Computer Science
Brown University
Providence, RI 02912
Email: rt@cs.brown.edu

Summary of Results

We give new methods for maintaining a point-location data structure for a dynamically-changing monotone subdivision $\mathcal{S}$. Our approach is based on the maintenance of two interlaced spanning trees, one for $\mathcal{S}$ and one for the graph-theoretic dual of $\mathcal{S}$. Queries are answered by using a centroid decomposition of the dual tree to drive searches in the primal tree. We maintain these trees via the link-cut trees structure of Sleator and Tarjan, leading to a scheme that achieves vertex insertion/deletion in $O(\log n)$ time, insertion/deletion of k-edge monotone chains in $O(\log n + k)$ time, and answers queries in $O(\log^2 n)$ time, with $O(n)$ space. Our techniques also allow for the dual operations *expand* and *contract* to be implemented in $O(\log n)$ time, leading to an improved method for spatial point-location in a 3-dimensional subdivision. In addition, we apply our approach to on-line point-location (where one builds $\mathcal{S}$ incrementally), improving our query bound to $O(\log n \log\log n)$ time and our update bounds to $O(1)$ amortized time in this case. We believe this is the first on-line method to achieve a polylogarithmic query time and constant update time.

Keywords: Computational geometry, point location, centroid decomposition, dynamic data structures, on-line algorithms

1 Introduction

An exciting trend in algorithmic research has been to show how one can efficiently maintain various properties of a combinatoric or geometric structure while updating that structure in a dynamic fashion (e.g., see [10]). A problem with tremendous potential for dynamization is planar point location, a classic problem in computational geometry (e.g., see [1, 13, 24, 29, 33]). Given a subdivision S of the plane into "cells", described by a total of n line segments, the problem is to preprocess S to allow for efficiently naming the cell containing a query point p. An important special case of the point-location problem occurs when each face in the planar subdivision is a monotone polygon with respect to the y-axis, that is,

*This research was supported in part by the National Science Foundation under Grants CCR-8810568 and CCR-9003299, and by the NSF and DARPA under Grant CCR-8908092.

†This research was supported in part by the National Science Foundation under grant CCR-9007851, and by the U.S. Army Research Office under grant DAAL03-91-G-0035.

the boundary of each face is intersected at most twice by any horizontal line. Given such a subdivision, Kirkpatrick [22] shows that one can construct an $O(n)$ space structure in $O(n)$ time that allows $O(\log n)$-time point-location queries. Edelsbrunner, Guibas, and Stolfi [14] show that one can achieve these same bounds by applying the fractional cascading paradigm of Chazelle and Guibas [5, 6] to the chain-method of Lee and Preparata [23]. One can also achieve these bounds after $O(n \log n)$ preprocessing, as shown by Sarnak and Tarjan [37], by applying the persistence paradigm of Driscoll, Sarnak, Sleator, and Tarjan [12] to a simple plane-sweeping procedure (as an example of a static$\rightarrow$dynamic$\rightarrow$static conversion). (See [13, 24, 33] for other references on this important problem.)

We are interested in maintaining a monotone subdivision dynamically, subject to edge insertion and deletion, vertex insertion and deletion, as well as the insertion or deletion of a monotone chain of k edges. In addition, we are also interested in operations that are duals to edge insertion and deletion, as in the framework of Guibas and Stolfi [21], where we allow for vertex expansion and contraction: an *expansion* splits a vertex v into two new vertices connected by an edge, and a *contraction* merges two adjacent vertices into a new vertex. These operations are useful in applying a dynamic point-location to spatial point location in 3-dimensional subdivisions [36] via persistence [12].

1.1 Previous Work

Early work on dynamic point location includes a method by Overmars [31], which is based on a segment-tree [2] approach to planar-point location, and achieves an $O(\log^2 n)$ query and update time with $O(n \log n)$ space. Fries, Mehlhorn, and Naeher [16, 17] present a data structure with $O(n)$ space, $O(\log^2 n)$ query time, and $O(\log^4 n)$ amortized update time (for edge insertion/deletion only), using an approach based on the static chain-method of Lee and Preparata [23]. Neither of these methods seem to extend to the dual update operations of expand and contract, however.

More recently, Preparata and Tamassia [34] have given techniques for maintaining monotone subdivisions that is also based on this chain-method, but improves the bounds of Fries et al. by representing the chains topologically rather than geometrically. In their scheme, inserting/deleting vertices on edges requires $O(\log n)$ time, and inserting/deleting monotone chains of edges requires $O(\log^2 n + k)$ time. Moreover, as shown in [36], their scheme can be extended to the dual update operations, which leads, via persistence [12], to a data structure for spatial point location that uses $O(N \log^2 N)$ space, requires $O(N \log^2 N)$ processing time, and allows for queries to be answered in $O(\log^2 N)$ time, where N is the size of the 3-dimensional subdivision.

Cheng and Janardan [8] present two methods for dynamic planar point-location that improve the time of edge updates. In their Scheme I they achieve $O(\log^2 n)$ 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, and in their Scheme II they achieve $O(\log n \log \log n + k)$ 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 of their methods are based on a search strategy derived from the priority search tree data structure of McCreight [26]. They dynamize this approach with the $BB(\alpha)$ tree data structure (e.g., see [28]), using the approach of Willard and Lueker [39] to spread local updates over future operations, and the method of Overmars [30] to perform global rebuilding (at the same time) before the "current" data structure becomes too unbalanced. Their methods do not seem to extend to the dual update operations, however, nor does it seem possible to improve their bounds for the on-line case.

Very recently, Chiang and Tamassia [11] have presented a dynamic data structure for monotone subdivisions, which is based on the static trapezoid method of Preparata [32] and extends previous work by Preparata and Tamassia [35] on dynamic point location in convex subdivisions with vertices on a fixed set of lines. The operations supported are insertion and deletion of vertices and edges, and horizontal translation of vertices. They show how to achieve queries in $O(\log n)$ time, while requiring $O(\log^2 n)$ time for updates. The space requirement for their method is $O(n \log n)$. In related work, Chiang, Preparata, and Tamassia [9] have shown that one can achieve an $O(\log n)$ query time in a dynamic environment that allows for ray-shooting queries and subdivision updates in $O(\log^3 n)$ time, and Goodrich and Tamassia [19] show how to maintain a similar environment so as to achieve $O(\log^2 n)$ time for all updates and queries (using a method built upon the scheme of the present paper).

1.2 Our Results

In this paper we show how to dynamically maintain a monotone subdivision so as to achieve an $O(\log^2 n)$ query time, an $O(\log n)$ time bound for vertex insertion/deletion, and $O(\log n + k)$ time for the insertion/deletion of a monotone chain of k edges. Our methods are based on the maintenance of two interlaced spanning trees, one for the subdivision and one for its graph-theoretic dual, to answer queries. Queries are performed by using a centroid decomposition of the dual tree to drive searches in the primal tree. We dynamize this approach using the edge-ordered dynamic tree data structure of Eppstein *et al.* [15], which is an extension of the link-cut trees data structure of Sleator and Tarjan [15, 38]. We use the "built-in" operations of *link*, *cut*, *split*, and *merge* to implement both our updates and queries. Our methods improve the previous bounds for dynamically maintaining monotone subdivisions.

We also show how to extend our approach to implement the dual operations of *expand* and *contract*, which, in turn, leads to an improved data structure for spatial point location via the persistence paradigm of Driscoll, Sarnak, Sleator, and Tarjan [12], where one dynamizes the problem to a 3-dimensional space sweep that uses our data structure to maintain the current "slice". This leads to an $O(N \log N)$ space data structure that requires only $O(N \log N)$ preprocessing time while achieving an $O(\log^2 N)$ query time, for a 3-dimensional subdivision of size N, which improves the space and preprocessing of the previous method [36] by a

$\log N$ factor.

Finally, we show how to apply our approach to on-line planar point location, where one builds a monotone subdivision incrementally. In this case we show how to maintain the centroid decomposition of the dual tree explicitly (in a $BB(\alpha)$ tree [28]), and apply a simple version of the fractional cascading paradigm of Chazelle and Guibas [5, 6] to improve the query time to $O(\log n \log\log n)$ while also improving the complexity of updates to $O(1)$ amortized time. We believe this is the first on-line point location method to achieve a polylogarithmic query time and constant update time.

2 Preliminaries

2.1 Monotone subdivisions

A *(planar) subdivision* $\mathcal{S}$ is a partition of the plane into polygons, called the *regions* of $\mathcal{S}$. We assume that $\mathcal{S}$ has one unbounded region, called the external region. A subdivision $\mathcal{S}$ is generated by a planar graph embedded in the plane such that the edges are straight-line segments. We assume a standard representation for the subdivision $\mathcal{S}$, such as doubly-connected edge lists [33].

A *monotone chain* is a polygonal chain such that each horizontal line intersects it in at most one point. A polygon is *monotone* if its boundary is partitionable into two monotone chains. A *monotone subdivision* is such that all its regions are monotone polygons. A *triangulation* is a subdivision such that the boundary of each region has 3 edges.

By orienting each edge of a monotone subdivision $\mathcal{S}$ by decreasing ordinate (i.e., so that it "points down"), we obtain a planar *st*-graph, i.e., a planar acyclic digraph with exactly one source (vertex without incoming edges) and one sink (vertex without outgoing edges). The source s and sink t of $\mathcal{S}$ are the highest and lowest vertices of $\mathcal{S}$, respectively. The *left chain* of a region r of $\mathcal{S}$ is the monotone chain on the boundary of r such that r in on the left side of it when traversed from top to bottom. The *right chain* is similarly defined. Note that according to this definition the left (resp., right) chain of the external region appears on the right (resp., left) boundary of the subdivision.

2.2 Centroid decomposition

Let T be free tree with n vertices of degree at most 3. A *centroid edge* of T is an edge e whose removal partitions T into two trees of size at most $1 + 2n/3$ each. It is well-known that if $n > 1$, such an edge exists and can be found in time $O(n)$ (e.g., see [3, 27]).

A *centroid decomposition tree* for T is a rooted binary tree B recursively defined as follows: If T has a single vertex v, then B consists of a single leaf node that stores vertex v. Otherwise, let e be a centroid edge of T, and let T' and T'' be the trees that result when removing e form T. The root of B stores edge e, and the left and right subtrees of B are centroid trees for T' and T'', respectively. The centroid decomposition tree B has $O(\log n)$ height, and can be constructed in $O(n)$ time (e.g., see [7, 20]).

2.3 Dynamic trees

Dynamic trees [38] are a versatile dynamic data structure for maintaining a forest of rooted trees. We shall use an extension of dynamic trees, called *edge-ordered dynamic trees* [15].

An edge-ordered tree is a rooted tree in which a cyclic order is imposed on the edges incident on each node (including the edge to the parent). The circular sequence of edges incident to node μ is called the *edge-ring* of μ. For example, in our application the trees are drawn in the plane and we use the counterclockwise ordering of the edges around each vertex given by the embedding. Edge-ordered dynamic trees support the following repertory of update operations [15]:

link(μ', μ'', e', e''): this operation assumes that μ' is the root of a tree T', μ'' is a node of another tree T'', and e'' is an edge incident on μ''. Add a new edge e' from node μ' to node μ'', thus making T' a subtree of T''. The new edge e' is inserted after edge e'' in the edge-ring of μ''.

cut(μ, e): this operation assumes that node μ is not the root of a tree and e is the edge from μ to its parent. Remove edge e, thus separating the subtree rooted at μ.

split$(\mu, \mu', \mu'', e, e_1, e_2)$: split node μ into two nodes μ' and μ'' connected by a new edge e. If $(\alpha e_1 \beta e_2 \gamma)$ is the the edge-ring of μ, then $\alpha e \gamma$ and $e e_1 \beta e_2$ are the edge-rings of μ' and μ'', respectively.

merge(μ', μ'', e): merge adjacent nodes μ' and μ'' connected by edge e into a single node μ. If αe is the edge-ring of of μ' and βe is the edge-ring of μ'', then $\alpha\beta$ is the edge-ring of μ.

Let T be a dynamic tree, subject to the above operations. Sleator and Tarjan [38] present two schemes for efficiently performing the link and cut operations on T, and these schemes carry over naturally to edge-ordered dynamic trees [15]. In this paper we assume the scheme that uses *partitioning by size*. In this scheme the edges of T are considered to be directed form the child to the parent, and an edge e from μ to ν is said to be *solid* if the subtree rooted at μ has more than half of the edges of the subtree rooted at ν. Otherwise, edge e is said to be *dashed*. There is at most one solid edge entering any node (from its children). Therefore, every node is in exactly on one path of solid edges (of length 0 or more). We refer to these paths as *solid paths*. (See Fig. 1.a.) A solid path π is represented as a balanced binary tree P_π, so that T is then stored as a collection of these *path trees*. For more details on partitioning by size, and how it can be exploited to efficiently perform dynamic tree updates and queries, see [15, 38].

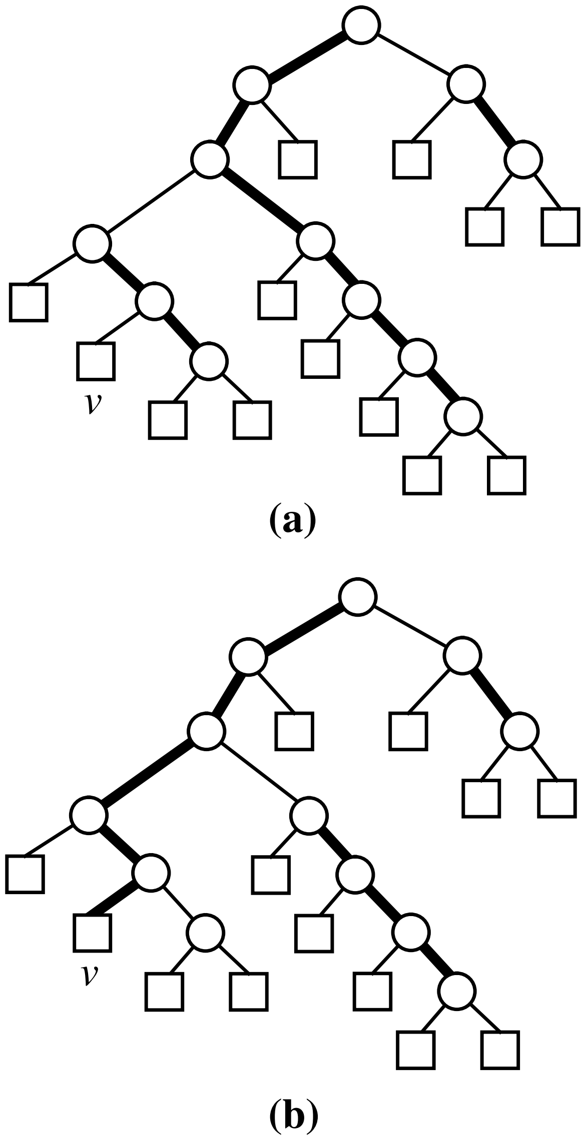


Figure 1: (a) A link-cut tree with edges partitioned into solid and dashed. (b) Effect of operation $expose(v)$ on the tree of part (a).

While link-cut trees support a variety of query operations, such as finding the least-common ancestor of two nodes, we shall use only the following operation that is part of the standard repertory of dynamic trees [15, 38]:

expose(μ): create a solid path π from node μ to the root by converting to solid all the dashed edges of π, and converting to dashed all the the solid edges that enter a node of π but are not on π. (See Fig. 1.b.) This operation may violate the definition of solid edges, so it is always followed by a procedure that undoes its effects.

Edge-ordered dynamic trees use linear space and support each of the above operations in $O(\log n)$ time, where n is the size of the tree(s) involved in the operation [15, 38].

3 Our Approach

In this section we address the problem of performing point location in a triangulation $\mathcal{S}$ with n vertices. Without loss of generality, we assume that $\mathcal{S}$ does not have horizontal edges. General subdivisions can be handled via a preliminary triangulation step, which takes $O(n)$ time if the subdivision is connected [4], and $O(n \log n)$ time otherwise [18].

We describe here a static method that uses $O(n \log n)$ space and preprocessing time, and supports point-location queries in $O(\log^2 n)$ time. We will show in the subsequent section how to dynamize this approach so as to achieve $O(n)$ space, the same query time, and an update time that is $O(\log n)$.

3.1 Building the Structure

A *monotone spanning tree* T of $\mathcal{S}$ is a rooted spanning tree such that any root-to-leaf path of T is monotone with respect to the y-axis. The root, s of T, then is the vertex in $\mathcal{S}$ with smallest y-coordinate. Such a monotone spanning tree T is obtained by choosing, for each vertex v in $\mathcal{S}$, some edge emanating from v, assuming all edges are directed downward. (See Fig. 2.) As we show in the next lemma, such a tree has a nice property that can be exploited for point location.

Lemma 3.1: *For any non-tree edge e of $\mathcal{S}$, the fundamental cycle $F(e)$ determined by e and T is a monotone polygon.*

Proof: Let $e = (v', v'')$. Since the spanning tree T is monotone, the paths π' and π'' of T from s to v' and v" are monotone chains. Let v be the least-common ancestor of v' and v''. The cycle $F(e)$ that results when adding e to T is the monotone polygon formed by the following monotone chains: (i) the subpath of π' from v to v' plus edge e, and (ii) the subpath of π'' from v to v''. $\Box$

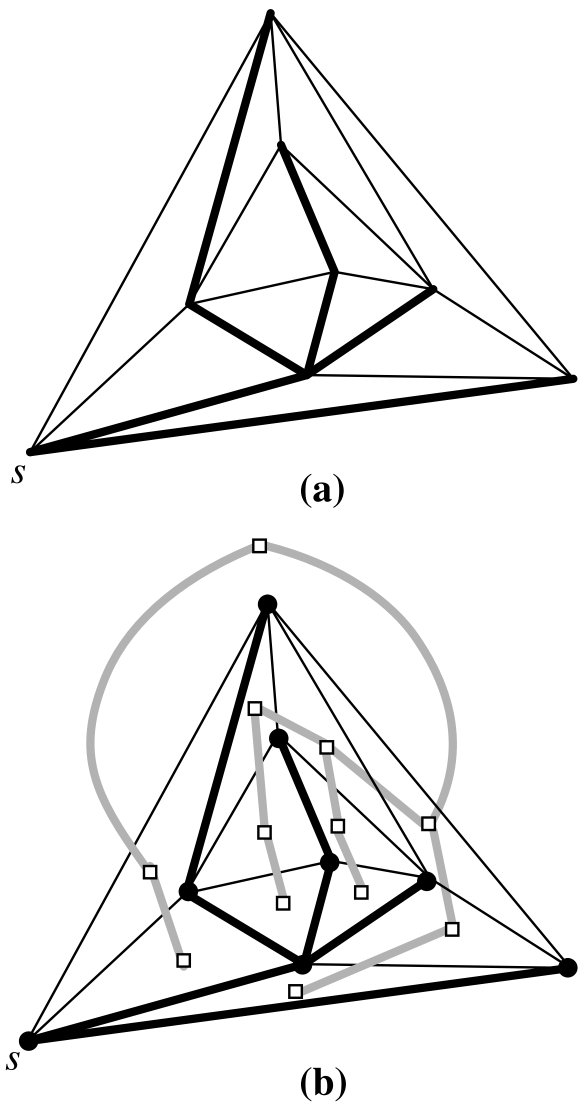


Figure 2: (a) A monotone spanning tree. (b) Its dual spanning tree.

This motivates the construction of our point-location structure, which is performed in the following four steps:

1. Construct a monotone spanning tree T for $\mathcal{S}$, and represent T as an edge-ordered tree rooted at the source vertex s, where the ordering of the edges incident on a vertex is given by the planar embedding.

2. Construct the graph-theoretic dual of $\mathcal{S}$, but exclude any edges dual to edges in T. This defines a spanning tree D on the dual graph [15], called the *dual spanning tree* of T (see Fig. 2). Each node of D is a region r of $\mathcal{S}$, and each edge e^* of D corresponds to a non-tree edge e in $\mathcal{S}$, which in turn determines a unique cycle $F(e)$ in $\mathcal{S}$ (when e is added to T). We represent D as an edge-ordered tree rooted at the external region, where the ordering of the edges incident on a node (region) is given by the planar embedding.

3. Form a centroid decomposition of D [3, 7, 20]. Recall that a centroid edge in D divides D into two subtrees whose sizes are in the interval $[|D|/3, 2|D|/3]$; hence, a centroid decomposition defines a binary tree B, where each internal node μ in B corresponds to a centroid edge e^*_μ of D, with the left child of μ being the portion of D "below" e_μ (i.e., inside $F(e_\mu)$), and the right child being the portion "above" e_μ (i.e., outside $F(e_\mu)$). (See Fig. 3.)

4. With each node μ in B we store the left and right chains of monotone polygon $F(e_\mu)$ in two sorted arrays $L(\mu)$ and $R(\mu)$, respectively. Note: to avoid confusion in the L and R lists we consider each edge to have two sides, a left side and a right side, which are distinct edges for the sake of this definition. (See Fig. 3.)

Lemma 3.2: *The above method runs in $O(n \log n)$ time and uses $O(n \log n)$ space.*

Proof: Steps 1 and 2 can be easily implemented in $O(n)$ time. Step 3 takes $O(n)$ time using the method of [7, 20]. Step 4 is the bottleneck step, in that it requires $O(n \log n)$ time and space to copy and store all the L and R lists for the nodes in B (since B has depth $O(\log n)$). □

Having presented our structure, let us describe how it can be used to answer a point location query.

3.2 Querying the Structure

Suppose we are given a query point p, and we wish to locate the cell in $\mathcal{S}$ containing p. Our method for performing this point location query is actually quite simple. We perform a search down B, where at each node μ we use the lists $L(\mu)$ and $R(\mu)$ to determine if p is inside or outside the polygon $F(e_\mu)$. Since since $L(\mu)$ and $R(\mu)$ are stored as arrays, we can perform two binary searches to determine if p is inside $F(e_\mu)$ in $O(\log n)$ time. If p is inside $F(e_\mu)$, then we visit μ's left child next; otherwise, we visit μ's right child next. This procedure continues until we reach the leaf of B corresponding to a single region—the cell of $\mathcal{S}$ containing p. Therefore, we have the following lemma:

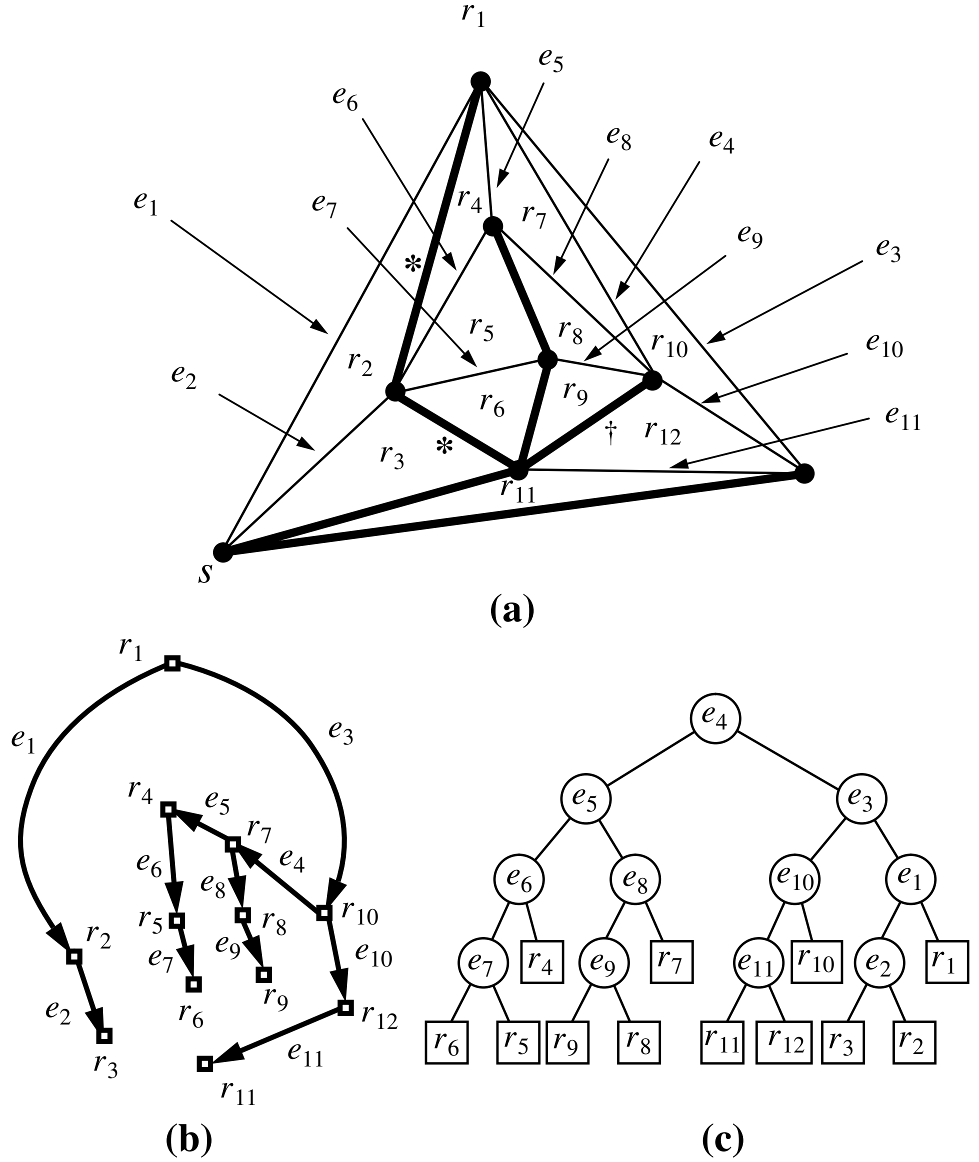


Figure 3: A centroid decomposition of the dual spanning tree of the previous figure. In this example, the edges in $L(\mu_{e_4})$ are marked with a "$*$" and the edge in $R(\mu_{e_4})$ is marked with a "$\dagger$".

Lemma 3.3: *Our point-location data structure supports point location queries in $O(\log^2 n)$ time.*

Incidentally, one can improve the query time to $O(\log n)$, while increasing the preprocessing time and space by at most a constant factor, via the *fractional cascading* technique of Chazelle and Guibas [5, 6]. One can also reduce the space to $O(n)$ using the same space-saving technique used by Edelsbrunner, Guibas, Stolfi [14], where one stores an edge only at the first place it appears in a root-to-leaf path in the decomposition tree. Thus, one can modify the above approach to match the query and space bounds of the previous best point location methods [14, 22, 37]. Our motivation for designing this new method was not to simply match the performance of previous methods, however, but to design a scheme that leads to an efficient dynamic point location method. So, let us leave the details of this static data structure to the interested reader, and concentrate instead on how this approach can be dynamized.

4 Dynamic Planar Point Location

In this section we show how to implement our point-location method dynamically using dynamic trees. Our dynamic environment supports the following repertory of update operations on a monotone subdivision $\mathcal{S}$:

InsertEdge$(e, r, v, w; r_1, r_2)$: Insert edge e between vertices v and w inside region r, which is then decomposed into regions r_1 and r_2 to the left and right of e, respectively.

DeleteEdge$(e, v, w, r_1, r_2; r)$: Remove edge e between vertices v and w and merge into region r the two regions r_1 and r_2 formerly to the left and right of e, respectively.

Expand$(e, v, r_1, r_2; v_1, v_2)$: Expand vertex v into vertices v_1 and v_2 connected by edge e, which has regions r_1 and r_2 to its left and right, respectively.

Contract$(e, r_1, r_2, v_1, v_2; v)$: Contract edge e between vertices v_1 and v_2 into vertex v. Regions r_1 and r_2 are those formerly to the left and right of e, respectively.

InsertChain$(\gamma, r, v, w; r_1, r_2)$: Insert a monotone chain γ between vertices v and w inside region r, which is then decomposed into regions r_1 and r_2 to the left and right of e, respectively.

DeleteChain$(\gamma, v, w, r_1, r_2; r)$: Let γ be a monotone chain between vertices v and w, whose internal vertices have degree 2. Remove γ and merge into region r the two regions r_1 and r_2 formerly to the left and right of γ, respectively.

Several problems arise in the dynamization of the static structure of Section 3. The first is that the static data structure assumes a preliminary triangulation step to ensure that the dual spanning tree has bounded degree and therefore admits a centroid decomposition. But it appears difficult to dynamically maintain a triangulation, since a newly inserted edge could intersect many triangulation edges.

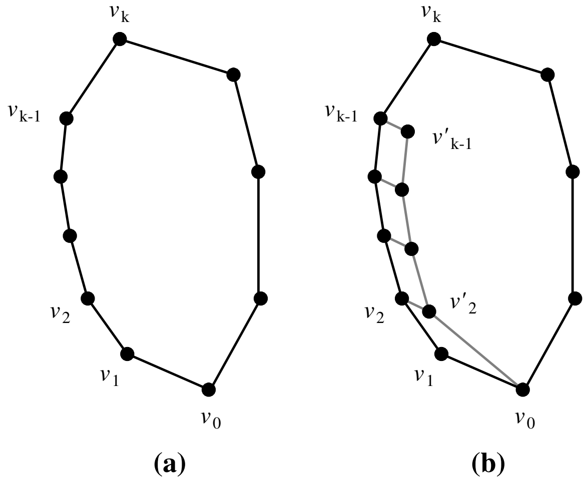


Figure 4: Example of refinement of a region.

4.1 A Virtual Triangulation of the Regions in $\mathcal{S}$

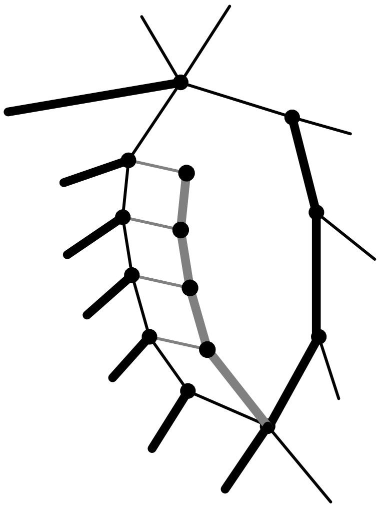
Let $\mathcal{S}$ be a monotone subdivision. Our method for overcoming this difficulty is to refine $\mathcal{S}$ into a new subdivision $\mathcal{R}$, as follows. For each region r of $\mathcal{S}$, let $v_0, v_1, \cdots, v_k$ be the left chain of r. If $k \geq 3$, we add inside r a "comb" consisting of $k-2$ new vertices, $v'_2, v'_3, \cdots, v'_{k-1}$, and $2(k-2)$ edges, (v_0, v'_2), (v'_i, v'_{i+1}), $(i = 2, \cdots, k-2)$, and (v'_i, v_i) $(i = 2, \cdots, k-1)$. See an example in Fig. 4. We assume that each new vertex v'_i is placed below and to the right of v_i, and infinitesimally close to it. Hence, the above refinement affects only the topology of the subdivision, so that a point location query has the same answer in $\mathcal{S}$ and $\mathcal{R}$. Also, it is immediate to verify that the refined subdivision has $O(n)$ vertices.

The *leftist spanning tree* of a monotone subdivision is defined as the monotone spanning tree obtained by selecting the leftmost incoming edge of every vertex, except the source (see Fig. 5.a). In addition to the above refinement of $\mathcal{S}$ into $\mathcal{R}$, we also maintain T as a leftist spanning tree of $\mathcal{R}$, with D being its graph-theoretic planar dual. As we show in the following lemma, this is sufficient to achieve the desired result.

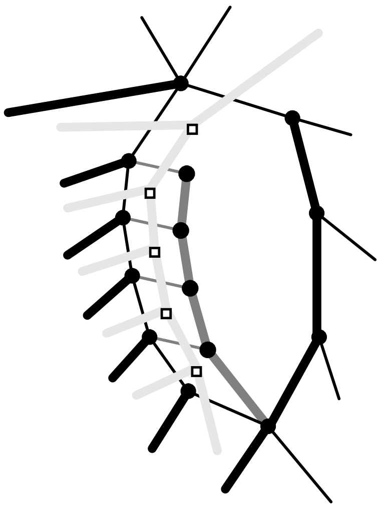
Lemma 4.1: *The planar dual of the leftist spanning tree in $\mathcal{R}$ has degree at most 3.*

Proof: Let T be the leftist spanning tree, and D its dual (see Fig. 5.b). We observe that tree D consists exactly of the dual edges of the topmost edges of the right chain of each region. Also, all the remaining edges of the right chain of each region are in tree T. Hence, the degree of a node r of D is at most one plus the number of nontree edges on the left chain of region r.

To prove the lemma, we show that every region r of the refined subdivision $\mathcal{R}$ has at most two non-tree edges in its left chain. Namely, if region r is to the left of a comb, then it has exactly two edges and thus no more than two non-tree edges in its left chain. Else (r is to the right of a comb), only the two topmost edges of the left chain of r may not be in T,



(a)



(b)

Figure 5: (a) Leftist spanning tree of a refined subdivision $\mathcal{R}$. (b) Dual of the leftist spanning tree of part (a). Tree edges are drawn thick, with comb edges being dark gray. The edges of the dual tree are drawn as thick light gray lines, and their dual edges in the comb are drawn as dotted lines. (We use this same convention in all the figures that follow, as well.)

since each of the remaining edges (which form the "spine" of the comb) is the only incoming edge of its end vertex and hence is in T. $\square$

Our dynamic data structure for point location in $\mathcal{S}$ simply consists of the leftist spanning tree T of $\mathcal{R}$, and of its dual spanning tree D, each represented as an edge-ordered dynamic tree [15, 38]. Tree T is rooted at the node associated with the (bottom-most) source vertex s, and tree D is rooted at the node associated with the external region. In both trees, the ordering of the edges incident on each node is given by the planar embedding. The overall space requirement of the data structure is $O(n)$.

4.2 Finding Fundamental Cycles

In order to perform queries efficiently we must be able to construct searchable representations of fundamental cycles in T, the monotone spanning tree for $\mathcal{R}$. More significantly, like our triangulation, our representations must be virtual, since an update operation may cause substantial restructurings in the centroid tree and edge-lists. Our approach for overcoming this difficulty consists of representing T as an edge-ordered dynamic tree [15] (see also [38]). As we show in the following lemma, this is sufficient for us to be able to quickly perform a point-cycle query in T.

Lemma 4.2: *Let T be a monotone spanning tree of $\mathcal{S}$, with root s. By representing T as an edge-ordered link-cut tree, one can determine in time $O(\log n)$ whether a query point p is on, inside, or outside the fundamental cycle $F(e)$ induced by a non-tree edge e of $\mathcal{S}$.*

Proof: In a link-cut tree [15, 38] representing T the operation *expose*(v) returns a balanced binary tree P_π that represents the path π of T from the root s to vertex v (i.e., the external and internal nodes of P_π store the vertices and edges of π, respectively, such that the in-order visit of P_π yields π). Hence, a balanced-tree representation of the left and right chains of $F(e)$, minus edge e, can be obtained by issuing *expose*(v') and *expose*(v''), where v' and v'' are the the endpoints of edge e. Testing whether a point p is on, inside, or outside cycle $F(e)$ can be reduced to searching such trees. All of the above steps take $O(\log n)$ time. $\square$

Constructing fundamental cycles in $\mathcal{S}$ is important, but not sufficient, for, in order to achieve an $O(\log^2 n)$ query time, we must also be able to find a centroid edge in the dual tree, D.

4.3 Locating a Centroid Edge in the Dual Tree

Fortunately, by also representing D as a link-cut tree, we can find this edge quickly, as we show in the following lemma:

Lemma 4.3: *Let D be a tree of degree 3, represented by a link-cut tree with partitioning by size. Then a centroid edge in D can be located in $O(\log n)$ time.*

Proof: As mentioned above, one of the main ideas of the link-cut tree data structure is to partition the tree D into "solid" paths and "dashed" edges [15, 38], and represent each solid path with a binary search tree. Let π be the solid path containing the root of D. We claim that the set of edges that are either in π or incident on the first node of π contains a centroid edge.

> (*Proof of claim:*) Let $\pi = (\mu_1, \cdots, \mu_k)$. We denote with S_i the subset of nodes of D consisting of node μ_i and the nodes in the (at most 3) subtrees connected to μ_i by dashed edges (see Fig. 6.a). Let w_i be the size of S_i, called the *weight* of node μ_i. We have that $\sum_{i=1}^{k} w_i = n$, where n is the number of nodes of D. From the definition of dashed edges [15, 38], we have that $w_i < n/2$ for $i = 2, \cdots, k$. We distinguish two cases. If $w_1 \leq 1 + 2n/3$, then there exists some j such that $n/3 - 1 \leq \sum_{i=1}^{j} w_i \leq 1 + 2n/3$. In this case the solid edge from μ_j to μ_{j+1} is a centroid edge. Otherwise, the dashed edge connecting the largest subtree of μ_1 is a centroid edge, since the largest of the subtrees of μ_1 has at most $n/2$ nodes (because it is connected by a dashed edge) and at least $n/3 - 1$ nodes (because μ_1 has no more than 3 incident edges). $\square$ (of claim)

Therefore, we can find a centroid edge of D in $O(\log n)$ time by traversing a root-to leaf path in the binary tree of the solid path containing the root of D. (See Fig. 6.b.) $\square$

Thus, we have shown how to perform the two main components of our point location procedure.

4.4 Point Location Querying

The location of a query point p, therefore, is performed as follows:

1. If D consists of a single node, we return the corresponding region and stop.

2. We find a centroid edge e^* of D using the algorithm of Lemma 4.3.

3. We cut tree D at edge e^*, and let D' and D'' be the resulting trees, where D'' contains the former root of D. Note: since this is a query step, not an update, we also store the edge e^* on a stack in this step, so that, after the query is done, we can reconstruct the original D via a series of link operations.

4. We determine if p is on, inside, or outside cycle $F(e)$ by applying the algorithm of Lemma 4.2 to tree T.

5. If p is on cycle $F(e)$, return the edge or vertex of $F(e)$ that contains e. Else, if p is inside the cycle, recursively apply the algorithm using D'. Otherwise (p is outside the cycle), recursively apply the algorithm using D''.

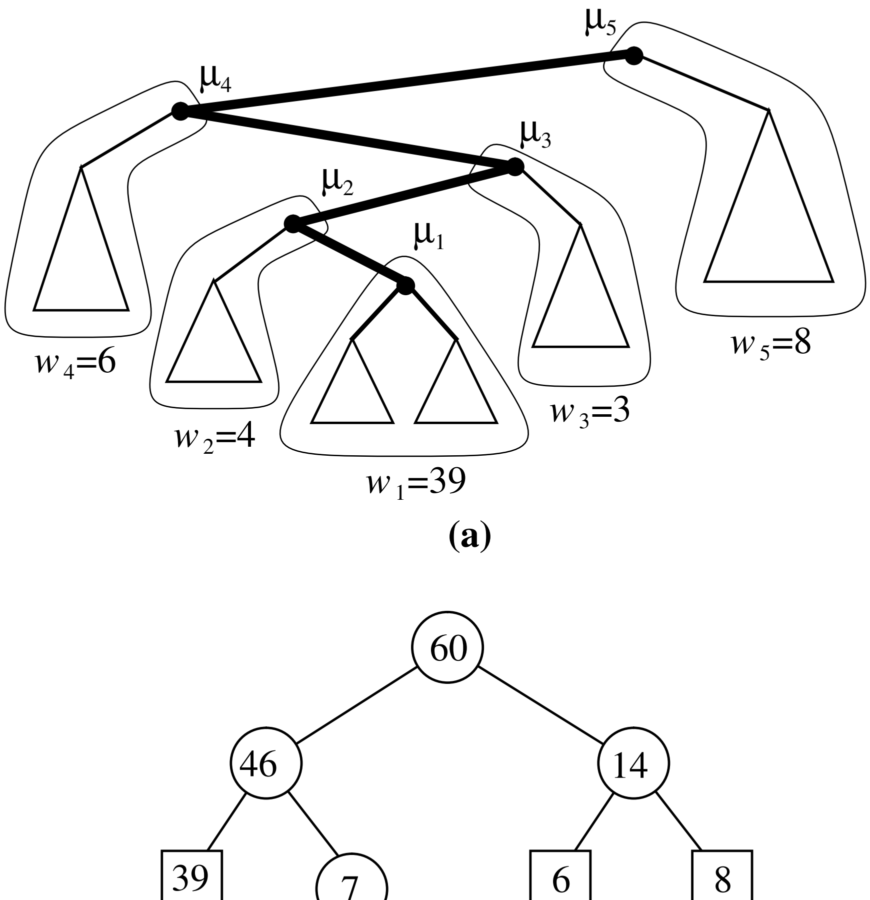


Figure 6: (a) Solid path containing the root of a dynamic tree. (b) Balanced tree associated with the solid path of part (a).

Our query operation is completed by reconstructing tree D by means of a sequence of $O(\log n)$ link operations that undo the cuts (by a series of pop operations on the stack used in Step 3).

Thus, we have the following theorem:

Theorem 4.4: *The above dynamic point location data structure supports point location queries in time $O(\log^2 n)$ and uses only $O(n)$ space.*

Proof: The query time bound should be immediately apparent given the above description. For the space bound note that data structure is no more than $\mathcal{R}$, represented using any standard plane graph representation, and T and D represented as link-cut trees (plus cross pointers, so, for example, each non-tree edge in $\mathcal{R}$ has a pointer to its dual in D). □

Having given our method for performing queries, let us next address our methods for updating $\mathcal{R}$. We begin with the *Contract* and *Expand* operations.

4.5 Edge Contraction and Expansion

Recall that in the *Expand*(v, v_1, v_2, e) operation we expand vertex v into vertices v_1 and v_2 connected by edge e with regions r_1 and r_2 being to the left and right of e, respectively (see Figure 7). There are two cases. In the first case the relative positions of v_1 and v_2 require that e become an edge in the leftist spanning tree T (as illustrated in Figure 7.a). In this case we perform the obvious *split* in T at v and update the corresponding pointer structures in $\mathcal{R}$ and D. We may also have to add an edge to the combs of pseudo-edges in r_1 and r_2 so as to maintain our refinement invariant. If this occurs we need also update the dual tree D (using $O(1)$ *split* and *link* operations) so that it remains a planar dual to $\mathcal{R}$. This can all be done in $O(\log n)$ time.

In the second case the positions of v_1 and v_2 require that e become a non-tree edge (see Figure 7.b). In this case we perform the obvious *split* in T at v, forming v_1 and v_1, *cut* T along e, and then link v_2 to its leftmost adjacent node, w in $\mathcal{R}$. We also perform any edge additions to combs in r_1 and r_2, if necessary, as in the first case. Of course, our modifications of T require that we perform changes to D. Specifically, we must cut D at the edge dual to (v_2, w) and perform a link to create a node dual to e. Since this can all be done in $O(\log n)$ time, it implies that we can perform the *Contract* operation in $O(\log n)$ time.

We implement the *Expand* operation by "reversing" the above steps in the obvious manner. Thus, we may also perform the *Expand* operation in $O(\log n)$ time.

4.6 Edge Insertions and Deletions

The next update operation we consider is edge insertion. Recall that in the operation *InsertEdge*$(e, r, v, w; r_1, r_2)$ we insert edge e between vertices v and w inside region r, which

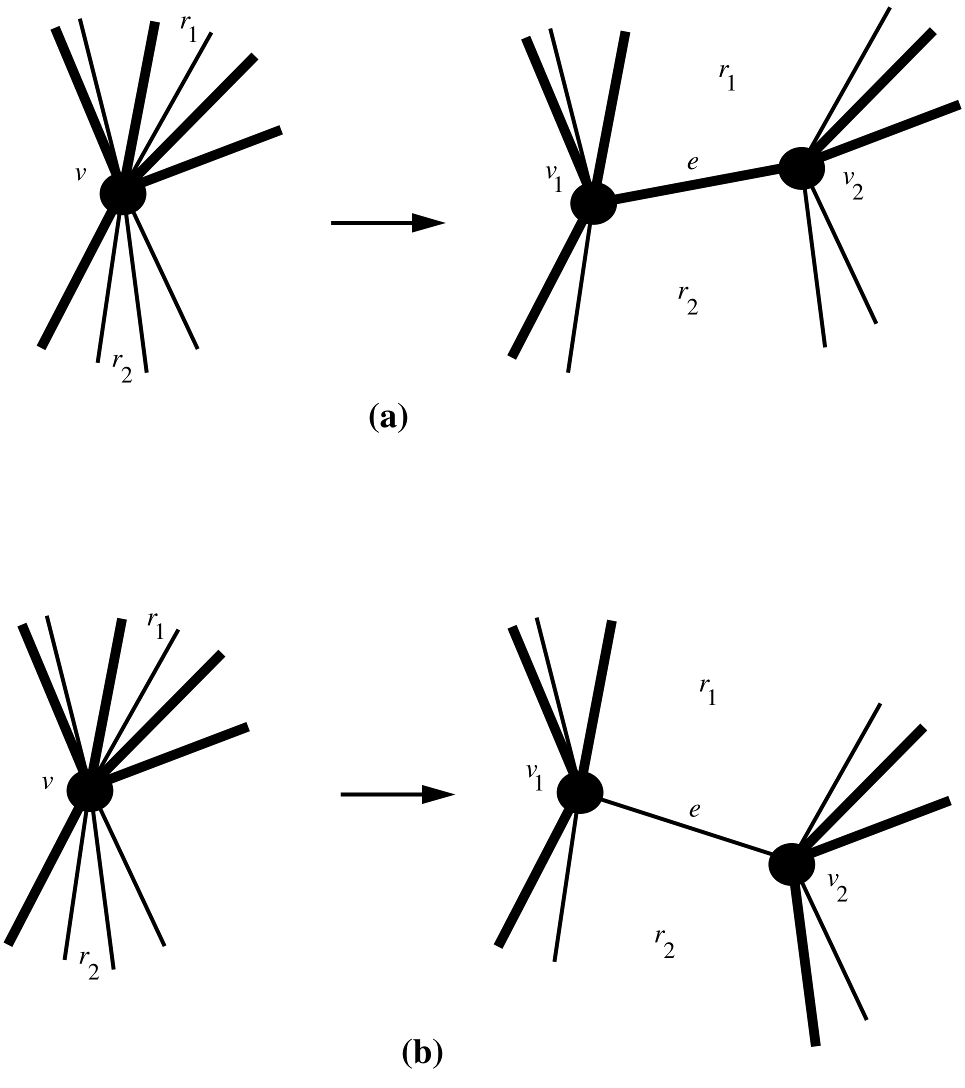


Figure 7: (a) An *Expand* operation in which e becomes an edge in the leftist spanning tree T, and (b) an *Expand* in which e becomes a non-tree edge.

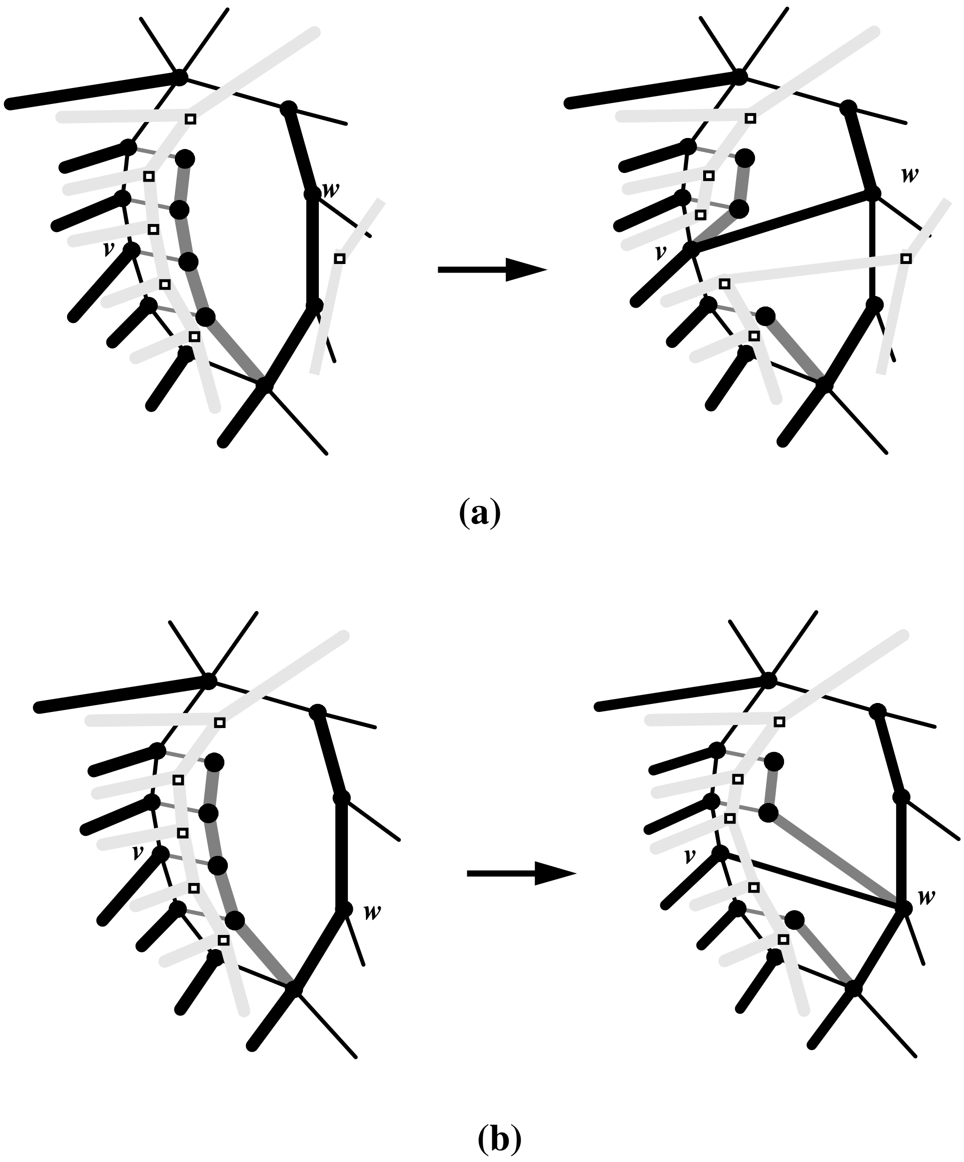


Figure 8: Insertions where v and w are on opposite sides of the region r. We illustrate in (a) an *InsertEdge* operation in which e becomes an edge in the leftist spanning tree T, and in (b) an *InsertEdge* in which e becomes a non-tree edge.

is then decomposed into regions r_1 and r_2 to the left and right of e, respectively. There are two cases.

In the first case v and w are on opposite sides of r, i.e., without loss of generality, v is on the left chain of r and w is on the right chain of r (see Figure 8). We distinguish two subcases:

1.1. Suppose $e = (v, w)$ must become an edge in the leftist spanning tree T (see Figure 8.a). In this case we perform a *cut* in T along the edge going out of w and then link the resulting subtree rooted at w to v. This may also require that we cut the comb in r at the vertex associated with v, deleting its incident edges, and begin a new comb at v (which contains any previous comb edges above v). Likewise, each *cut* in T has a corresponding *link* in D, and each *link* in T has a corresponding *cut* in D. Since the number of needed *cut* and *link* operations is $O(1)$, the total time for this case is $O(\log n)$.

1.2. Suppose $e = (v, w)$ must become a non-tree edge (see Figure 8.b). In this case we need only change the comb in r by cutting it at the vertex associated with v and beginning a new comb at w. This can be done with $O(1)$ *cut* and *link* operations in T, but it does not change the topology of D. Thus, this case also takes only $O(\log n)$ time.

In the second case for edge insertion vertices v and w are on the same side of r. There are two obvious subcases:

2.1. Suppose v and w are both on the left side of r (see Figure 9.a), where, without loss of generality, v has larger y-coordinate than w. In this case we must cut the comb in r at the vertex associated with v and the vertex associated with w, and add the edge $e = (v, w)$ as a non-tree edge in $\mathcal{R}$. We begin a new comb at w in r_1 that retains the edges of the old comb between the vertices associated with v and w, and we concatenate the portion of the old comb below the vertex associated with w with the portion of the old comb above the vertex associated with v, to form the new comb for r_2. This can all be done using $O(1)$ *link* and *cut* operations in T. Likewise, cutting the comb in r also requires that we perform associated cuts in D, and the comb concatenations have associated links in D. Again, there are only $O(1)$ such operations, however, so this case can be implemented in $O(\log n)$ time.

2.2. Suppose v and w are both on the right side of r (see Figure 9.b), where, without loss of generality, v has larger y-coordinate than w. In this case we simply cut T at the edge f going out of v and perform a link of the subtree rooted at v along the edge $e = (v, w)$. This also requires that we create a new (leaf) node in D and link it along the edge dual to f. We need not change the comb in r (which is now the comb for r_1), and the comb in r_2 is the null comb, so this completes the construction. Clearly, this case requires $O(\log n)$ time.

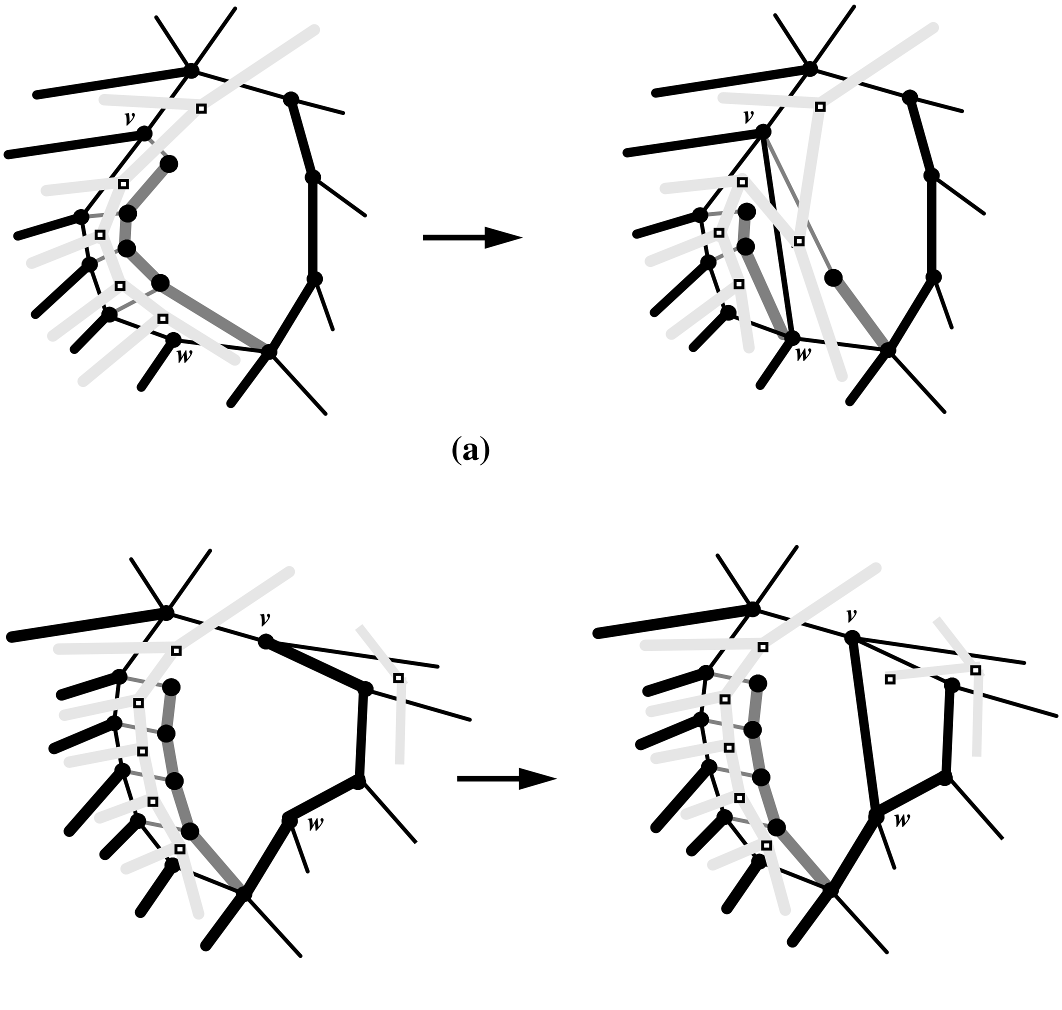


Figure 9: Insertions where v and w are on the same side of the region r. We illustrate in (a) the case when v and w are on the left side and in (b) the case when v and w are on the right side.

Thus, we can perform the *InsertEdge* operation in $O(\log n)$ time. Since the *DeleteEdge* operation is the "reverse" on an *InsertEdge*, this also implies that we can perform the *DeleteEdge* operation in $O(\log n)$ time.

4.7 Chain Updates

The only update operations that remain to be described are the chain update operations. Recall that in the operation $InsertChain(\gamma, r, v, w; r_1, r_2)$ we insert a monotone chain γ between vertices v and w inside region r, which is then decomposed into regions r_1 and r_2 to the left and right of e, respectively. Note that this is essentially the same as in the case of the *InsertEdge* operation, except that instead of adding a single edge (v, w) we are now inserting a monotone chain. It should not be surprising, then, that our method for performing the *InsertChain* operation is the same as that for the *InsertEdge* operation, except that where we previously performed a single link in T from v to w, and added a trivial chain in r_2 from v to w, we must now link in an entire chain in T, as well as its corresponding comb. If we perform these link operations in series, then we will require $O(k \log n)$ time, where k is the length of the chain. So, instead, we first build a link-cut tree representation of the chain and its corresponding comb, which takes $O(k)$ time [15, 38], and then perform the $O(1)$ link operation required to link these chains into T. Likewise, we must build a chain of size $O(k)$ dual to the inserted chain and its comb, and link this into D, but again a link-cut tree representation of the chain can be built in $O(k)$ time, and then this can be linked into D with $O(1)$ link operations. Thus, the entire time needed for the *InsertChain* operation is $O(\log n + k)$. Since the *DeleteChain* operation amounts to the reversal of this procedure, this also implies that the *DeleteChain* operation can be implemented in $O(\log n + k)$ time (it is actually easier, since we replace the building of link-cut tree representations of $O(k)$-length chains with the garbage collection of the space used by such representations). Therefore, we have the following.

Theorem 4.5: *Let $\mathcal{S}$ be a monotone subdivision of current size n that is subject to a sequence of on-line updates. Point location in $\mathcal{S}$ can be done with a fully dynamic data structure that uses $O(n)$ space and supports queries in time $O(\log^2 n)$ and update operations* InsertEdge, DeleteEdge, Expand, *and* Contract *in time $O(\log n)$. Also, update operations* InsertChain *and* DeleteChain *take time $O(\log n + k)$, where k is the size of the monotone chain being inserted or deleted. All the time bounds are worst-case.*

5 Spatial Point Location

We can extend our method further to derive an efficient algorithm for performing point location in 3-dimensional cell complexes whose cells are convex polytopes. Let $\mathcal{C}$ be such a convex cell-complex with n vertices and N facets. Note that both n and the number of edges of $\mathcal{C}$ are $O(N)$. Following the same general approach of Preparata and Tamassia [36], we obtain a spatial point location data structure by combining the persistence-addition technique of Driscoll, Sarnak, Sleator, and Tarjan [12] and our dynamic structure for planar point location.

A conventional dynamic data structure is called *ephemeral* since its instantiation preceding an update is not recoverable after the execution of the update. A *fully persistent* structure supports both accesses and updates to any of its past versions; a *partially persistent* structure supports accesses to any of its past versions, but updates only to its most current version. The general technique of Driscoll et al. [12] can be used to add persistence to an ephemeral linked data structure whose records are pointed to by a bounded number of pointers. The resulting persistent data structure uses additional $O(1)$ amortized space per update operation, and has the same asymptotic query time (worst-case for partial persistence, and amortized for full persistence). Thus, the following lemma is vital to our construction.

Lemma 5.1: *The dynamic point-location data structure of Theorem 4.5 can be implemented with a linked representation such that each record is pointed to by a bounded number of pointers.*

Proof: As mentioned above, our data structure is essentially just a link-cut tree representation of a leftist monotone spanning tree T and its graph-theoretic planar dual D. Moreover, we maintain D as a degree-3 tree, which implies that the underlying link-cut tree representation satisfies the bounded number of pointers condition (see [15, 38] for details). The tree T need not have bounded degree, however. Nevertheless, by using the implementation of edge-ordered dynamic trees trees given by Eppstein *et al.* [15], we represent T so as to satisfy the bounded-degree condition (see [15] for details). □

Thus, we can create a persistent version of our point location data structure. But for this to be useful it must also be meaningful. We find this meaning in the following lemma.

Lemma 5.2: *Let $\mathcal{S}_1$ and $\mathcal{S}_2$ be monotone subdivisions whose associated planar st-graphs are isomorphic. A dynamic point location data structure for $\mathcal{S}_1$ (as discussed in Theorem 4.5) can be used for dynamic point location in $\mathcal{S}_2$ after changing only the values of the vertex coordinates.*

Proof: Since the planar st-graphs associated with $\mathcal{S}_1$ and $\mathcal{S}_2$ are isomorphic, the leftist spanning tree for $\mathcal{S}_1$ and the leftist spanning tree for $\mathcal{S}_2$ are isomorphic (as are their respective graph-theoretic planar duals). Thus, applying our construction to $\mathcal{S}_1$ yields a data structure that is topologically identical to that for $\mathcal{S}_2$. Moreover, at no place in our point location method do we ever explicitly need the coordinates of the subdivision endpoints. We only needed to be able to perform a comparison-based binary search for a y-coordinate in a monotone chain, and then be able to determine to which side of a line L a query point lies. Thus, by replacing the comparison tests of $\mathcal{S}_1$ with the isomorphic tests in $\mathcal{S}_2$, we obtain a dynamic point location structure for $\mathcal{S}_2$. □

We reduce the 3-d point location problem to an application of persistence to a dynamic 2-d point location where we "sweep" space by a plane $\pi(z)$ parallel to the x and y axes and at height z, for $z = -\infty$ to $z = +\infty$. Let $\mathcal{C}(z)$ be the intersection of $\mathcal{C}$ with the plane $\pi(z)$. It is easy to verify that $\mathcal{C}(z)$ is a convex (and hence monotone) subdivision. We view the z-axis as a measure of "time" and consider the process of making plane $\pi(z)$ sweep the cell complex $\mathcal{C}$. The location of a query point $q = (x, y, z)$ in the cell complex $\mathcal{C}$ can be reduced to the location of point (x, y) in the monotone subdivision $\mathcal{C}(z)$. Hence, spatial point location can be performed using a partially persistent planar point location data structure.

While the geometry of $\mathcal{C}(z)$ continuously evolves in time, its topology changes only when plane $\pi(z)$ goes through a vertex v of $\mathcal{C}$, i.e., for z', z'' such that $z_i < z' < z'' < z_{i+1}$ the planar st-graphs associated with $\mathcal{C}(z')$ and $\mathcal{C}(z'')$ are isomorphic. Hence, the space-sweep process goes through $2n + 1$ topologically different subdivisions. Also, when the plane $\pi(z)$ goes through a vertex v_i, the resulting modification of the subdivision $\mathcal{C}(z)$ can be performed by a sequence of f_i update operations, each an *Expand* or a *Contract*, where f_i is the number of facets whose top or bottom vertex is v_i [36]. (Note that $\sum_{i=1}^{n} f_i = O(N)$.)

By Lemma 5.2, the same planar point-location data structure can be used for all query points whose z-coordinate is in the range (z_i, z_{i+1}), provided the x and y coordinates of the vertices are expressed as (linear) functions of z. Thus, our data structure for spatial point location consists of a partially persistent version of the dynamic planar point location data structure of Theorem 4.5. By Lemma 5.1, such structure satisfies the hypothesis for applying the persistence-addition technique of [12].

It is important to observe that, although our query algorithm modifies the ephemeral data structure (see section 4.2), such changes are only temporary and need not be remembered by the persistent data structure. Hence, at the expense of increasing the storage space by a constant factor we can use duplicate copies for the pointers and data fields that are modified by a query operation. The duplicate fields are disregarded during the updates.

We summarize the performance of our spatial point location data structure in the following theorem:

Theorem 5.3: *Let $\mathcal{C}$ be a convex 3-dimensional cell complex with N facets. There exists a data structure for point location in $\mathcal{C}$ that uses $O(N \log N)$ space and supports queries in $O(\log^2 N)$ time worst-case.*

6 On-Line Point Location

Many applications involve constructing an object incrementally while requiring that all the properties of the structure be maintained *on-line*. In the context of this paper, we desire a scheme to incrementally construct a planar subdivision while maintaining an efficient point-location data structure for it. This can also be viewed as an instance of dynamic point location when only insertions are allowed. In this section we show how to implement our centroid-decomposition approach to planar point location on-line using $BB(\alpha)$ trees and some dynamic data structuring techniques of Overmars to achieve $O(1)$ amortized time per update and $O(\log n \log\log n)$ time (worst-case) for answering queries.

We support the following operations:

InsertVertex$(v, e, r_1, r_2; e_1, e_2)$: Insert a vertex v on edge e, which has regions r_1 and r_2 to its left and right, respectively, expanding e into e_1 and e_2.

InsertEdge$(e, r, v_1, v_2; r_1, r_2)$: Insert edge e between vertices v_1 and v_2 inside region r, which is decomposed into regions r_1 and r_2 to the left and right of e, respectively.

InsertChain$(\gamma, r, v_1, v_2; r_1, r_2)$: Insert a monotone chain γ between vertices v_1 and v_2 inside region r, which is decomposed into regions r_1 and r_2 to the left and right of e, respectively.

6.1 A Simple On-Line Point Location Structure

We explicitly maintain the monotone tree T and the dual tree D for the refined subdivision $\mathcal{R}$. We also explicitly maintain B, the balanced decomposition tree of D, in a $BB(\alpha)$ tree. Each node μ in B corresponds to a subtree D_μ of D, which in turn corresponds to a subpolygon P_μ of P. Thus, each leaf in B corresponds to a node of D, which in turn corresponds to a region in $\mathcal{R}$. For each node μ in B we explicitly store the chains $L(\mu)$ and $R(\mu)$. In addition, in the spirit of *fractional cascading* [5, 6], for each node μ, we maintain auxiliary lists $AL(\mu)$ and $AR(\mu)$, which are defined recursively as follows:

$$AL(\mu) = \begin{cases} L(\mu) & \mu \text{ is a leaf} \\ L(\mu) \cup AL(\lambda) \cup AL(\nu) & \text{otherwise} \end{cases}$$

$$AR(\mu) = \begin{cases} R(\mu) & \mu \text{ is a leaf} \\ R(\mu) \cup AR(\lambda) \cup AR(\nu) & \text{otherwise} \end{cases}$$

where λ and ν are the children of μ should μ be an internal node. By keeping pointers from each element x in $AL(\mu)$ to its predecessors in $L(\mu)$, $AL(\lambda)$, and $AL(\nu)$ (and similar pointers for each x in $AR(\mu)$) we can answer queries in $O(\log n)$ time. This is because after an $O(\log n)$ time binary search in the AL and AR lists for the root of B, the search at each other node in B requires only $O(1)$ time, and there are $O(\log n)$ such other nodes. The update operations are easily implemented as follows:

InsertVertex$(v, e, r_1, r_2; e_1, e_2)$: We first make the obvious update to the planar graph representation of $\mathcal{R}$, inserting v on e and splitting e into e_1 and e_2. We then use the pointer to e to locate the records in $L(\mu)$ and $R(\mu')$ for the endpoint, w, of e that is nearer to the root of T. We then add a record for v next to w's record in $L(\mu)$ and $R(\mu')$, respectively. This can all easily be done in $O(1)$ time. We must also update the auxiliary lists, however, by adding a record for v to the AL list for each node from μ to the root of B and a record for v to the AR list for each node from μ' to the root. This can be implemented in $O(\log n)$

time by performing a query for v from the root to μ and μ', respectively, and adding v to each list we search in along the way.

InsertEdge$(e, r, v_1, v_2; r_1, r_2)$: We first make the obvious update to the planar graph representation of $\mathcal{R}$, inserting e into r and splitting r into r_1 and r_2. This also necessitates that we modify the node λ in D corresponding to r. This modification will be in the form of the division of λ into two nodes λ_1 and λ_2, with some of λ's adjacencies becoming λ_1's adjacencies and the other adjacencies of λ becoming λ_2's adjacencies. Of course, in performing this modification of D we must also update B to reflect this modification. We do this by visiting the leaf μ in B associated with λ, creating two new nodes μ_1 and μ_2, which are associated with λ_1 and λ_2, respectively, and making these nodes be the children of μ. This addition, in turn, requires that we update the balance information stored in B, and in some cases, this requires that we perform node rotations in B to maintain the weight-balance requirements of this $BB(\alpha)$ tree. Performing a rotation at a node μ in B requires more than just updating balance information and changing some pointers at the nodes around μ—it also requires that we change the L and R lists (and their associated auxiliary lists) for μ and ν, the child of μ that is now becoming the parent of ν. Note, however, that we must change the pointer fields of the records in the auxiliary lists at μ's old parent, ρ, but we do not need to add or delete any records from these lists. This is because the set of descendents of ρ do not change. Thus, the time required to perform such a rotation is proportional to the size of the L, R, AL, and AR lists at μ and ν, which is proportional to n_μ, the number of descendents of μ.

InsertChain(γ, v_1, v_2): This operation can be implemented by combining the methods for *InsertVertex* and *InsertEdge*. We leave the details to the interested reader.

From the above descriptions it should be clear that queries can be answered in $O(\log n)$ worst-case time, as can *InsertVertex* operations. Also, the worst-case time for an *InsertEdge* operation is $O(n)$. Nevertheless, since we are represented B as a $BB(\alpha)$ tree, we can derive an efficient amortized running time for the *InsertEdge* operation. In particular, we observe that performing a rotation at μ requires $O(n_\mu)$ time, where n_μ is the number of leaf descendents of μ. This is because the size of $AL(\mu)$ and $AR(\mu)$ is bounded by the number of vertices in $P(\mu)$, the subpolygon associated with μ, which is $O(n_\mu)$. We can, therefore, take advantage of the following lemma:

Lemma 6.1 [28]: *Let $\alpha \in (1/4, 1-\sqrt{2}/2)$ and let f be a non-decreasing function such that the cost of performing a rotation in a $BB(\alpha)$ tree at a node μ is $f(n_\mu)$. Then the total cost of the rebalancing operations in a sequence of m insertions and deletions into an initially empty tree is*

$$O(m \sum_{i=1}^{c \log m} (1-\alpha)^i f((1-\alpha)^{-i-1})),$$

where $c = 1/\log(1-\alpha)$.

This immediately implies that the cost of performing a sequence of n *InsertEdge* operations, starting with an initially empty subdivision is $O(n \log n)$; hence, the amortized cost of each *InsertEdge* operation is $O(\log n)$. This gives us the following theorem:

Theorem 6.2: *One can maintain a monotone subdivision on-line with $O(\log n)$ query time for point locations and $O(\log n)$ amortized time for vertex and edge insertions (inserting a chain of k vertices requires $O(k \log n)$ amortized time). The space for this data structure is $O(n \log n)$.*

One can improve the space of the above method to $O(n)$ at the expense of making the query time an amortized bound, using the methods of Fries et al. [16, 17]. As we mentioned earlier, however, our interest is in performing updates in $O(1)$ amortized time ($O(k)$ time for chain insertion). In the next section we show how to modify our approach to achieve this goal. Our modification reduces the space to $O(n)$ and increases the query time by only a $\log \log n$ factor.

6.2 Improving the Implementation

The main idea of our improvement is to apply a "bucketing" technique [30] at two different places in our structure. The first application is for the L and R lists at the nodes of B. For simplicity of expression, let us concentrate our attention on the L lists; the modifications for the R lists are similar. For each node μ we add a list $L'(\mu)$, which we maintain to be a subsequence of $L(\mu)$ so that between any two consecutive elements (e, f) in $L'(\mu)$ there are at most $2N$ elements of $L(\mu)$, where N is $O(\log n)$. Moreover, for each pair of consecutive elements (e, f) in $L'(\mu)$ we store the elements of $L(\mu)$ that fall between e and f in a data structure that allows $O(1)$ insertion time, given an element's position, and $O(\log n_e)$ query time, where $n_e = O(N)$ is the number of elements between e and f [25]. The elements of $L(\mu)$ between e and f can intuitively be viewed as belonging to a "bucket" for the pair (e, f).

We modify our definition of the AL and AR lists to take advantage of the sublists L' and R'. In particular we now define $AL(\mu)$ and $AR(\mu)$ as follows:

$$AL(\mu) = \begin{cases} L'(\mu) & \mu \text{ is a leaf} \\ L'(\mu) \cup AL(\lambda) \cup AL(\nu) & \text{otherwise} \end{cases}$$

$$AR(\mu) = \begin{cases} R'(\mu) & \mu \text{ is a leaf} \\ R'(\mu) \cup AR(\lambda) \cup AR(\nu) & \text{otherwise} \end{cases}$$

where λ and ν are the children of μ should μ be an internal node. This immediately implies that the query time increases to $O(\log n \log N) = O(\log n \log \log n)$, since, given an query value x, determining the predecessor of x in $L(\mu)$ requires $O(\log N) = O(\log \log n)$ time given the position of x in $AL(\mu)$. Nevertheless, this modification has a worthwhile consequence—it reduces the time for *InsertVertex* operations to $O(1)$ (amortized). This bound follows from the fact that an *InsertVertex* requires more than $O(1)$ time only if that *InsertVertex* operation causes a bucket size to grow larger than $2N$. In such a case we simply split this bucket into two buckets with sizes N and $N + 1$, respectively. Of course, splitting a bucket in, say, $L(\mu)$

requires that we add a new element to $L'(\mu)$ and, hence, to the AL list at each node from μ to the root of B. Since all of these updates can be implemented in $O(\log n + N) = O(N)$ time, we can charge this cost to the N operations that previously inserted elements to this bucket (to make it grow to size $2N$). Thus, the amortized cost of an *InsertVertex* is $O(1)$.

We can also reduce the space of this method to $O(n + n \log n/N) = O(n)$ by maintaining that N be $\Omega(\log n)$ and by insuring that any time two consecutive buckets have size smaller than $\lfloor N/2 \rfloor$ we concatenate these buckets into a single bucket.

These modifications do not reduce the cost for *InsertEdge* operations, however. To reduce their cost we apply the bucketing idea a second time, this time to the tree B itself. In particular, we modify our maintenance of B so that, instead of associating a single node of D (corresponding to a single region of $\mathcal{R}$) with each leaf of B, we associate a subtree of D with at most $2N$ nodes. Any time an *InsertEdge* operation causes a leaf subtree D_μ to grow to more than $2N$ nodes, we locate the centroid edge in D_μ and split D_μ into two trees D_{μ_1} and D_{μ_2}, creating two new nodes μ_1 and μ_2, which become the children of μ. This of course necessitates that we build new lists (L, R, L', R', AL, and AR) for μ_1 and μ_2, and update the AL and AR lists from μ to the root to reflect the addition of any new values (needed to maintain the recursive definitions of the AL and AR lists). Nevertheless, this can all be done in $O(\log n + N) = O(N)$ time, which can be charged to the N previous *InsertEdge* operations at μ (which each were implemented in $O(1)$ time). Thus, each *InsertEdge* will run in $O(1)$ amortized time.

The method for implementing *InsertChain* operations is basically a combination of the methods for *InsertVertex* and *InsertEdge*, that details of which we leave to the interested reader. Thus, we have the following theorem:

Theorem 6.3: *One can maintain a monotone subdivision on-line with $O(\log n \log\log n)$ query time for point locations and $O(1)$ amortized time for vertex and edge insertions (inserting a chain of k vertices requires $O(k)$ amortized time). The space needed for this data structure is $O(n)$.*

We believe this theorem provides the first on-line point-location data structure with an $O(1)$ amortized update time and polylogarithmic query time.

7 Conclusion

We have given a new approach to planar point location and showed how it can be used to derive new, improved bounds for dynamic point location, spatial point location, and on-line point location. We leave as an open problem the existence of a fully dynamic method for dynamic point-location that runs in $O(\log n)$ time per query and $O(1)$ amortized time per update.

Acknowledgements

We would like to thank Bernard Chazelle for suggesting the pursuit of an on-line point-location method with $O(1)$ amortized update time. We would also like to thank Siu-Wing Cheng, Ravi Janardan, and S. Rao Kosaraju for several stimulating conversations related to topics addressed in this paper.

References

[1] M. J. Atallah, *Parallel techniques for computational geometry*, Proc. IEEE, 80 (1992), pp. 1435–1448.

[2] J. L. Bentley and D. Wood, *An optimal worst case algorithm for reporting intersections of rectangles*, IEEE Trans. Comput., C-29 (1980), pp. 571–577.

[3] B. Chazelle, *A theorem on polygon cutting with applications*, in Proc. 23rd Annu. IEEE Sympos. Found. Comput. Sci., 1982, pp. 339–349.

[4] ——, *Triangulating a simple polygon in linear time*, Discrete Comput. Geom., 6 (1991), pp. 485–524.

[5] B. Chazelle and L. J. Guibas, *Fractional cascading: I. A data structuring technique*, Algorithmica, 1 (1986), pp. 133–162.

[6] ——, *Fractional cascading: II. Applications*, Algorithmica, 1 (1986), pp. 163–191.

[7] ——, *Visibility and intersection problems in plane geometry*, Discrete Comput. Geom., 4 (1989), pp. 551–581.

[8] S. W. Cheng and R. Janardan, *New results on dynamic planar point location*, SIAM J. Comput., 21 (1992), pp. 972–999.

[9] Y.-J. Chiang, F. P. Preparata, and R. Tamassia, *A unified approach to dynamic point location, ray shooting, and shortest paths in planar maps*, in Proc. 4th ACM-SIAM Sympos. Discrete Algorithms, 1993, pp. 44–53.

[10] Y.-J. Chiang and R. Tamassia, *Dynamic algorithms in computational geometry*, Proc. IEEE, 80 (1992), pp. 1412–1434.

[11] ——, *Dynamization of the trapezoid method for planar point location in monotone subdivision*, Internat. J. Comput. Geom. Appl., 2 (1992), pp. 311–333.

[12] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, *Making data structures persistent*, J. Comput. Syst. Sci., 38 (1989), pp. 86–124.

[13] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*, Springer-Verlag, 1987.

[14] H. Edelsbrunner, L. J. Guibas, and J. Stolfi, *Optimal point location in a monotone subdivision*, SIAM J. Comput., 15 (1986), pp. 317–340.

[15] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung, *Maintenance of a minimum spanning forest in a dynamic planar graph*, J. Algorithms, 13 (1992), pp. 33–54.

[16] O. Fries, *Zerlegung einer planaren Unterteilung der Ebene und ihre Anwendungen*, m.S. thesis, Inst. Angew. Math. Inform., Univ. Saarlandes, Saarbrücken, West Germany, 1985.

[17] O. Fries, K. Mehlhorn, and S. Näher, *Dynamization of geometric data structures*, in Proc. 1st Annu. ACM Sympos. Comput. Geom., 1985, pp. 168–176.

[18] M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan, *Triangulating a simple polygon*, Inform. Process. Lett., 7 (1978), pp. 175–179.

[19] M. Goodrich and R. Tamassia, *Dynamic ray shooting and shortest paths via balanced geodesic triangulations*, in Proc. 9th Annu. ACM Sympos. Comput. Geom., 1993, pp. 318–327.

[20] L. J. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. E. Tarjan, *Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons*, Algorithmica, 2 (1987), pp. 209–233.

[21] L. J. Guibas and J. Stolfi, *Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams*, ACM Trans. Graph., 4 (1985), pp. 74–123.

[22] D. G. Kirkpatrick, *Optimal search in planar subdivisions*, SIAM J. Comput., 12 (1983), pp. 28–35.

[23] D. T. Lee and F. P. Preparata, *Location of a point in a planar subdivision and its applications*, SIAM J. Comput., 6 (1977), pp. 594–606.

[24] ——, *Computational geometry: a survey*, IEEE Trans. Comput., C-33 (1984), pp. 1072–1101.

[25] C. Levcopoulos and M. H. Overmars, *A balanced search tree with $o(1)$ worst-case update time*, Acta Informatica, 26 (1988), pp. 269–277.

[26] E. M. McCreight, *Priority search trees*, SIAM J. Comput., 14 (1985), pp. 257–276.

[27] N. Megiddo, *Linear-time algorithms for linear programming in R^3 and related problems*, SIAM J. Comput., 12 (1983), pp. 759–776.

[28] K. Mehlhorn, *Sorting and Searching*, vol. 1 of Data Structures and Algorithms, Springer-Verlag, Heidelberg, West Germany, 1984.

[29] J. O'Rourke, *Computational geometry*, Annu. Rev. Comput. Sci., 3 (1988), pp. 389–411.

[30] M. H. Overmars, *The design of dynamic data structures*, vol. 156 of Lecture Notes in Computer Science, Springer-Verlag, 1983.

[31] ——, *Range searching in a set of line segments*, in Proc. 1st Annu. ACM Sympos. Comput. Geom., 1985, pp. 177–185.

[32] F. P. Preparata, *A new approach to planar point location*, SIAM J. Comput., 10 (1981), pp. 473–482.

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

[34] F. P. Preparata and R. Tamassia, *Fully dynamic point location in a monotone subdivision*, SIAM J. Comput., 18 (1989), pp. 811–830.

[35] ——, *Dynamic planar point location with optimal query time*, Theoret. Comput. Sci., 74 (1990), pp. 95–114.

[36] ——, *Efficient point location in a convex spatial cell-complex*, SIAM J. Comput., 21 (1992), pp. 267–280.

[37] N. Sarnak and R. E. Tarjan, *Planar point location using persistent search trees*, Commun. ACM, 29 (1986), pp. 669–679.

[38] D. D. Sleator and R. E. Tarjan, *A data structure for dynamic trees*, J. Computer and Systems Sciences, 26 (1983), pp. 362–391.

[39] D. E. Willard and G. S. Lueker, *Adding range restriction capability to dynamic data structures*, J. ACM, 32 (1985), pp. 597–617.