Dynamic Algorithms in Computational Geometry*

(Revised Version)

Yi-Jen Chiang   Roberto Tamassia
Department of Computer Science
Brown University
Providence, RI 02912–1910
(yjc@cs.brown.edu, rt@cs.brown.edu)

Abstract

Research on dynamic algorithms for geometric problems has received increasing attention in the last years, and is motivated by many important applications in circuit layout, computer graphics, and computer-aided design. In this paper we survey dynamic algorithms and data structures in the area of computational geometry. Our work has a twofold purpose: it introduces the area to the nonspecialist and reviews the state-of-the-art for the specialist.

(September 1992)

*Research 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.

1 Introduction

The development of dynamic algorithms has acquired increasing theoretical interest, motivated by many important applications in network optimization, VLSI layout, computer graphics, and distributed computing. In this paper we survey dynamic algorithms for computational geometry problems.

Dynamic (or incremental) computation considers updating the solution of a problem when the problem instance is modified. Many applications are incremental (or operation-by-operation) in nature. Considerable savings can be achieved if the new solution need not be generated "from scratch," especially when the problem is large-scale.

A typical framework is to process *on-line* a sequence of *queries* and *updates* on some structure that evolves over time. By "on-line," we mean that the sequence of operations is not known in advance, and each operation must be completed before the next one is processed. Another dynamic framework consists of *maintaining* the value of a function over a dynamically evolving set of objects; for example, maintaining the minimum distance of a point set that is updated by insertions and deletions.

Typical quality measures for a dynamic algorithm are the storage space and the query and update times. A dynamic algorithm or data structure is *semi-dynamic* if the repertory of update operations consists only of "insertions" or only of "deletions," while a *fully dynamic* algorithm supports an intermixed sequence of insertions and deletions.

Dynamic data structures are not only a goal on themselves, but also an important tool in solving higher dimensional static problems using the *space-sweep* paradigm. Examples include hidden surface removal, construction of visibility graphs, and point location using persistent data structures.

We survey dynamic algorithms and data structures in the area of computational geometry. Our work has a twofold purpose: it introduces the area to the nonspecialist and reviews the state-of-the-art for the specialist.

Section 2 reviews fundamental data structures such as balanced search trees. In Section 3 we review general techniques for dynamization. Sections 4–8 focus on range searching, intersections, point location, convex hull, and proximity. Problems that do not fall in these categories are discussed in Section 9. Section 10 concludes the paper with open problems.

Previous tutorial and survey work in the area is as follows. A recent overview of major topics in computational geometry is given by F. Yao [175]. Fundamental dynamic geometric techniques are described in the books by Edelsbrunner [56], Mehlhorn [109] and by Preparata and Shamos [133]. Iyengar et al. [84] survey dynamic data structures for multidimensional searching. Overmars' thesis [121] is an excellent reference for dynamic computational geometry results up to 1983.

Throughout the paper, n denotes the current input size of the problem being considered, and k denotes the output size of a query operation (e.g., the number of points reported in a range searching query).

2 Fundamental Data Structures

In this section we define some terminology and briefly describe some fundamental dynamic data structures that are used as building blocks by a variety of algorithms.

2.1 Asymptotic Notation

Let $f(n)$, $g(n)$ be functions denoting the time or space complexity of an algorithm or data structure. We assume that these functions are positive, nondecreasing and defined over the positive integers. The notation $\log n$ will be used as a shorthand for $\max(1, \log_2 n)$.

We make use of the following standard asymptotic notations:

- $f(n) = O(g(n))$ if there exist constants $c, n_0 > 0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$.
- $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$.
- $f(n) = o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

The concept of *amortized time complexity* [162] will be used in the analysis of data structure for which the worst-case time complexity of an operation is larger than the average time complexity over a sequence of operations. That is, if a sequence of m operations takes total time t, then we say that the *amortized time complexity* of an operation is t/m. A detailed discussion on amortized complexity and techniques for analyzing it can be found in [162].

2.2 Balanced Trees

Consider a set P of n points in one-dimensional space (i.e., a set of real numbers). A *balanced tree* supports the following operations on P:

- test if a query point q is in set P (membership query);
- *locate* a query point q in P, i.e., find the largest point p of set P that is less than or equal to q (p is called the *immediate predecessor* of q in P);
- insert a new point into set P;
- remove a point from set P.

Balanced trees are typically realized as binary search trees whose internal nodes store the points of P. A balanced tree has logarithmic height, so that queries can be efficiently executed. After an update (insertion or deletion), the tree is "rebalanced" by means of "rotations" to maintain the logarithmic height. We can classify balanced trees into two groups: *height balanced* and *weight balanced*. In the former, one balances the height of the subtrees of each node, while in the latter, one balances the number of nodes in the subtrees of each node. Examples of height balanced trees are AVL trees [1] and red-black trees [75]. Weight-balanced trees were first presented in [119] and include $BB[\alpha]$ trees [25]. All of these trees use $O(n)$ space and support each operation of the above repertory in time $O(\log n)$.

An important variation of balanced trees stores weighted items, where the weight of an item is usually associated with its access frequency in membership queries. Several schemes have been devised to support membership queries in time $O(\log w/w_i)$, where w_i is the weight of the item being accessed, and w is the total weight. Analogous time bounds can be achieved for update operations. See, e.g., the *biased search trees* of Bent, Sleator and Tarjan [12].

Several data structures for computational geometry are obtained by augmenting a balanced tree with secondary structures stored at its nodes, thus the cost to perform a rotation is no longer $O(1)$ since we also have to update the secondary structures. If the time to rebuild the secondary data structures is logarithmic, then it is convenient to use red-black trees, since they can be rebalanced with only $O(1)$ rotations after an insertion or deletion [79,80,104,161].

If updating the secondary structures takes more than logarithmic time, we can instead use weight-balanced trees. Let $f(\ell)$ denote the time to update the secondary structures after a rotation at a node whose subtree has ℓ leaves, and assume that we perform a sequence of n update operations, each an insertion or a deletion, into an initially empty $BB[\alpha]$-tree. If $f(\ell) = O(\ell \log^c \ell)$, with $c \geq 0$, then the amortized rebalancing time of an update operation is $O(\log^{c+1} n)$; if $f(\ell) = O(\ell^a)$, with $a < 1$, then the amortized rebalancing time of an update operation is $O(1)$ [110]. That is, even if a rotation may need considerable time to rebuild the secondary structures involved, the amortized cost per insertion/deletion is still fairly small, because expensive rebalancing actions are guaranteed not to occur too often. This property has been extensively used by a variety of geometric data structures.

Balanced trees also support the following *concatenable-queue* operations on a collection of one-dimensional point sets:

- split set P into sets P' and P'', where P' contains the points of P that are less than or equal to a given point q;
- splice sets P' and P'' into a new set P, where all the elements of P' are less than the elements of P''.

The time complexity of these operations is $O(\log n)$, where n is the size of P.

Details on balanced trees can be found in textbooks such as [46,110,160]. New techniques for rebuilding balanced tree structures are presented by Andersson [6,7].

Aragon and Seidel [8] present *randomized search trees*, whose balancing strategy is based on *randomization*. The performance bounds are *expected-case*, where the expectation is taken over all possible sequences of "coin flips" in the update algorithms, and does not rely on any assumptions about the input. Randomized search trees combine the features of other types of search trees, including red-black trees, biased search trees, and $BB[\alpha]$ trees. Namely, query and update times are $O(\log n)$, rebalancing is done with $O(1)$ rotations as in red-black trees, rebuilding of secondary structures can be done with performance analogous to the one of $BB[\alpha]$ trees, and operations on weighted items have time bounds similar to those of biased search trees.

2.3 Fractional Cascading

The problem of searching for an element in a collection of k sets of total size n frequently arises in computational geometry. The naive method, which consists of performing binary search separately in each set, uses space $O(n)$ and has query time $O(k \log n)$. More efficient techniques have been developed for specific instances of this repetitive search problem [57,83,101,163,167,168]. Chazelle and Guibas [33,34] generalize these solutions and provide a data structuring technique called *fractional cascading*, which achieves query time $O(\log n + k)$ using $O(n)$ space. This time bound is optimal, and is faster than the one of the naive method by a logarithmic factor.

Fractional cascading can be formalized as follows: Let U be an ordered universe, e.g., the real numbers, and let G be an undirected graph. Assume that for each vertex v of G, there is a set $C(v) \subseteq U$, called the *catalog* of v, and that for every edge e of G there is a given *range* $R(e) = [l(e), r(e)]$ (closed interval in U with endpoints $l(e)$ and $r(e)$). Let $n = \sum_{v \in G} |C(v)|$ denote the total size of the catalogs and assume that G has $O(n)$ vertices. We want to organize the catalogs in a data structure that supports the following operations efficiently:

- Let q be an arbitrary query element of U, and T a tree subgraph of G such that $q \in R(e)$ for all $e \in T$. Locate q in each catalog $C(v)$, with $v \in T$.
- Given an element $y \in U$ and (a pointer to) the immediate predecessor x of y in $C(v)$, insert y into $C(v)$.
- Given (a pointer to) element $x \in C(v)$, delete x from $C(v)$.

We assume that G has *locally bounded degree*, i.e., there is a constant d such that for all $v \in G$ and $x \in U$, there are at most d edges e incident upon v such that $x \in R(e)$. Clearly, if G has bounded degree, it also has locally bounded degree for any choice of the ranges associated with the edges of G.

Let k denote the size of the query tree T. The static fractional cascading technique of Chazelle and Guibas [33,34] achieves $O(\log n + k)$ query time. Regarding dynamic fractional cascading, Chazelle and Guibas [34] show that insertions and deletions of elements can be supported in $O(\log n)$ amortized time, such that the query time is $O(\log n + k \log\log n)$. Mehlhorn and Näher's dynamic fractional cascading data structure [111] improves the update time down to $O(\log\log n)$ amortized. Also, if only insertions or deletions are allowed, the $O(\log\log n)$ factor in the query and update time decreases to $O(1)$. Dietz and Raman [50] eliminate the amortization and show how to obtain $O(\log\log n)$ worst-case update time.

3 General Dynamization Methods

Several general techniques have been developed for constructing dynamic data structures from static ones. Overmars' thesis [121] describes in detail the results in this area until 1983.

Throughout this section, $P(n)$, $Q(n)$ and $M(n)$ indicate preprocessing time, query time and storage space, respectively.

Local rebuilding, or *balancing*, denotes the techniques applied to search trees so that they maintain logarithmic height during a sequence of updates. *Partial rebuilding* is an alternative technique that proceeds by reconstructing entire subtrees when they become out of balance. It is typically applied to weight-balanced trees.

Global rebuilding is a method that periodically reconstructs the entire data structure [121]. It is best used in conjunction with data structures that support "weak" forms of update, where an item is inserted or deleted quickly at the expense of slightly deteriorating the balance of the structure. An example of weak update is *lazy deletion*, which does not remove the deleted item from the structure but instead marks it. More formally, a data structure supports *weak* updates if there are constants b and c such that after $b \cdot n$ updates on a newly built data structure S of size n, the query time, update time, and space requirement are at most a factor c larger than the corresponding quantities in S. Global rebuilding consists of completely reconstructing the structure after $b \cdot n$ updates. This gives a dynamic data structure with query time $O(Q(n))$, space requirement $O(M(n))$, amortized insertion time $O(I(n) + P(n)/n)$ and amortized deletion time $O(D(n) + P(n)/n)$, where $I(n)$ and $D(n)$ are the weak insertion and weak deletion times, respectively. It is possible to turn the amortized bound into worst-case by spreading the reconstruction over a number of subsequent updates, while using the old structure for queries.

A search problem on a set S of objects is called *decomposable* if for any partition (S', S'') of S, the answer to a query on S can be obtained in constant time from the answers to queries in S' and S''. For example, the closest point problem (find the point of set S closest to a given query point) is decomposable. Dynamic data structures for a decomposable search problem can be obtained by maintaining a collection of static data structures that operate on blocks of items of S. The following two dynamization methods are developed for decomposable search problems:

- The *equal block method* partitions S into $f(n)$ subsets of about equal size, each represented by a static data structure. In terms of the parameters of the static data structure and the function $f(n)$, the equal block method has query time $O(f(n)Q(n/f(n)))$, update time $O(\log n + P(n)/n + P(n/f(n)))$, and space $O(f(n)M(n/f(n)))$. If $Q(n) = O(\log n)$ and $P(n) = O(n)$, the function $f(n)$ can be chosen as $f(n) = \sqrt{n/\log n}$ so that the dynamic query and update times are all $O(\sqrt{n \log n})$.

- The *logarithmic method* partitions S into $O(\log n)$ sets of sizes given by the powers of two in the binary representation of n. If weak deletion can be supported in $D(n)$ time, then combining the global rebuilding technique, it gives a dynamic data structure with query time $O(Q(n)\log n)$, insertion time $O((\log n)P(n)/n)$, deletion time $O(\log n + D(n) + P(n)/n)$ and space $O(M(n))$.

The equal block method is developed by Maurer and Ottmann [107], van Leeuwen and Wood [99] and van Leeuwen and Maurer [97]. The simple version stated here is based on van Leeuwen and Overmars [98]. The first version of the logarithmic method is given by Bentley [15]. It is extended by Bentley and Saxe [20] who describe some more general methods and by Overmars and van

Leeuwen [123,127] who show how to turn this insertion-only method to fully dynamic. Several variations of dynamization methods for decomposable problems are reported in [53,71,112,120,121, 125,126,140,152,169].

The dynamization technique of Rao Vaishnavi Iyengar [140] considers either semi-dynamic decomposable search problems where only insertions are performed, or deletion-decomposable search problems, where the answer of a query on a set S' can be computed in time proportional to computing the answers to queries on $S' \cup S''$ and S''. Membership, range searching, and range counting are examples of deletion-decomposable problems. If the static query time depends only on n, the dynamic query time and space requirement are the same as those of the static data structure, while the update time is guaranteed to be asymptotically smaller than the static preprocessing time. (Note that this is true for range counting but not for range searching.)

Van Kreveld and Overmars [93] show how to obtain concatenable data structures from existing data structures, based on a technique called the *ordered equal block method*. This can be applied to all data structures for decomposable search problems.

Dobkin and Suri [51] have recently presented a general technique for maintaining the maximum value of a symmetric function $f(x, y)$ over pairs of elements of a dynamic set S. Examples include maintaining the minimum (maximum) distance of a set of points or the minimum (maximum) separation among a set of axis-parallel rectangles. They consider a *semi-on-line* dynamic model, where the time when an element is deleted becomes known at the moment of its insertion. The semi-on-line model lies between the *offline* and *on-line* models, in the sense that it assumes no knowledge of the insertions and partial knowledge of the deletions.

The main result of [51] can be expressed as follows: If the function f admits a static data structure that can be constructed in time $P(n)$ and allows one to find the maximum of $f(x, y)$ over all $y \in S$ for a given $x \in S$ in time $Q(n)$, then the maximum value of f (over all pairs of elements) in a semi-on-line sequence of insertions and deletions can be maintained in amortized time $O((P(n) \log n)/n + Q(n) \log n)$ per update operation.

Thus, at the cost of $\log n$ factor, the data structure allows semi-on-line update capability. As an immediate application, the diameter, or the closest pair, of a planar point set can be maintained in time $O(\log^2 n)$ per semi-on-line update. Smid [152] gives an alternative description of the algorithm, and shows how the amortized update time can be made worst-case. Moreover, it is shown that the method is a generalization of the logarithmic method: if the semi-on-line sequence consists of only insertions, we get the logarithmic method.

Vaishnavi [164,165,166] proposes several multidimensional tree structures. A multidimensional generalization of balanced binary trees, called *d-dimensional balanced binary tree*, is given in [164]. It stores a set of n d-dimensional points in $O(d \cdot n)$ space and supports access, insertion and deletion operations in $O(\log n + d)$ time, where only $O(d)$ rotations are needed to rebalance the tree after an insertion or deletion. An amortized analysis of the insertions and deletions of this data structure is given in [165], which shows that a mixed sequence of m insertions and deletions in the structure storing n d-dimensional points takes $O(d(n + m))$ total rebalancing time. Based on the two-dimensional version of the data structure, [165] also gives another data structure called *self-organizing balanced binary tree*, which is *nearly-optimal*, meaning that the access probabilities of the items are not known *a priori*, and the tree is updated after each access so that the access time of the i-th item with searching frequency w_i is $O(\log w/w_i)$, where w is the sum of the frequencies of all items (see [11]). It is shown that the total rebalancing time for a mixed sequence of m accesses, insertions and (restricted) deletions in a self-organizing balanced binary tree initially storing n items is $O(n+m)$. A dynamic multidimensional file structure, called *multidimensional (a, b)-tree*, which is a multidimensional generalization of (a, b)-trees of [110] is presented in [166]. It is shown that a *weak* d-dimensional (a, b)-tree (with b $\geq$ 2a) of size n requires $O(d(m+n))$ time for a mixed sequence of m insertions and deletions. Based on the two-dimensional (a, b)-trees, a *self-organizing (a, b)-tree* is also given which is proved to be *nearly-optimal* as defined above, and is shown to require total of $O(m+n)$ rebalancing time for a mixed sequence of m accesses, insertions and (restricted) deletions, if initially storing n items.

4 Range Searching

Range searching is a fundamental multidimensional search problem that consists of reporting the points of a set P contained in an orthogonal query range (an interval in one dimension, a rectangle with sides parallel to the coordinate axes in two dimensions, etc.). Fig. 1 gives an example of two-dimensional range searching.

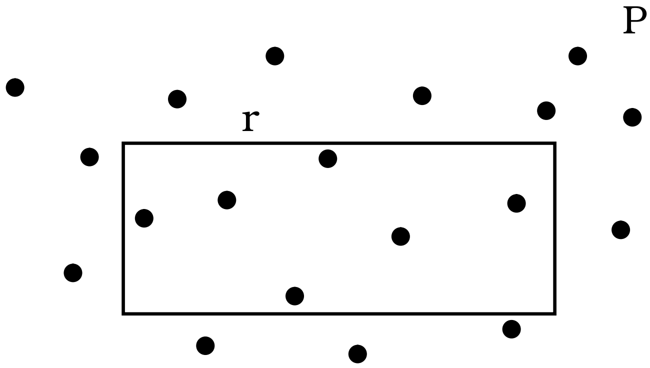

Figure 1: An example of two-dimensional range searching.

4.1 One-Dimensional Range Searching

Balanced search trees support one-dimensional range searching in time $O(\log n + k)$, where k is the number of reported answers, with $O(n)$ space requirement and $O(\log n)$ update time: Let T be a balanced binary tree whose internal nodes are associated with the points of P, and whose leaves are associated with the open intervals delimited by the points. We thread the nodes of T in symmetric order. A range query for the interval (x', x'') is performed as follows (see Fig. 2):

- search for x' and x'' in T, which gives the nodes μ' and μ'' whose associated points or intervals respectively contain x' and x'';
- follow the thread pointers from μ' to μ'' and report the points of P associated with all the internal nodes encountered.

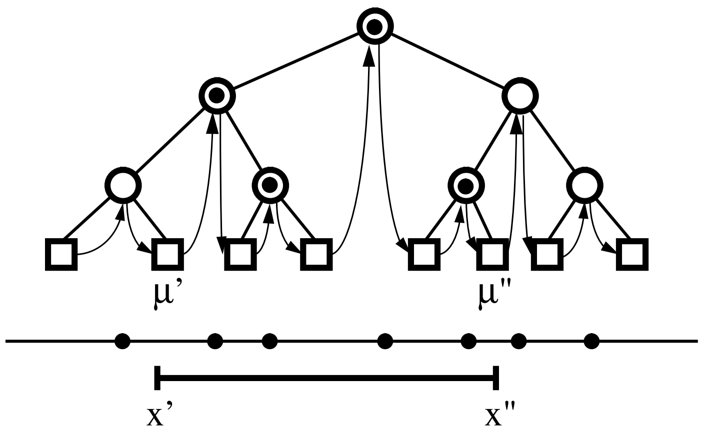

Figure 2: Performing one-dimensional range searching.

Note that the thread pointers are not affected by rotations, and can be updated in $O(1)$ time after an insertion or a deletion.

4.2 Two-Dimensional Range Searching

Range queries in two dimensions can be answered using a two-level tree structure called *range-tree*. The primary structure is a balanced search tree T whose leaves are associated with the points of P, sorted by x-coordinate. For each internal node μ, let $P(\mu)$ be the set of points associated with the leaves of T in the subtree of μ, called the *proper points* of μ (see Fig. 3). The secondary structure of μ is a one-dimensional data structure for range searching by y-coordinate in the set $P(\mu)$. Every point p of P is stored in the secondary structures of the $O(\log n)$ nodes on the path from the root to the leaf associated with p. Hence, the space requirement is $O(n \log n)$. By presorting the points both by x- and y-coordinate, $O(n \log n)$ preprocessing time can be achieved [16].

Let $R = (x_1, x_2) \times (y_1, y_2)$ be the query range. A node μ of T is an *allocation node* for (x_1, x_2) if the vertical strip (x_1, x_2) contains $P(\mu)$ but not $P(\nu)$, where ν is the parent of μ. Hence, (x_1, x_2) has $O(\log n)$ allocation nodes, the sets of proper points of the allocation nodes are disjoint, and their union consists of the points of P in the strip (x_1, x_2). The points of P inside R can be found by performing one-dimensional range queries for the range (y_1, y_2) in the secondary structures of the allocation nodes of (x_1, x_2) (see Fig. 4). The total time complexity is $O(\log^2 n + k)$.

When adding a point p to P, we add a new leaf to T, and then insert p into the secondary structures of the nodes from the new leaf to the root. This takes $O(\log^2 n)$ time. Next, we rebalance

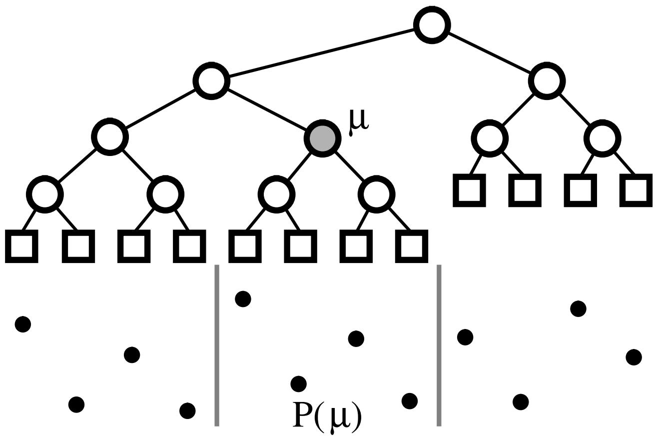

Figure 3: Definition of the set $P(\mu)$.

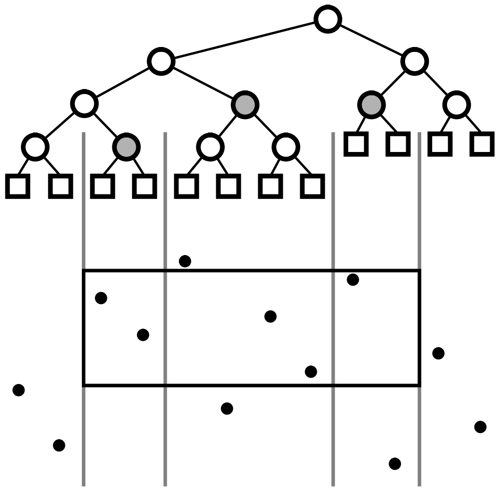

Figure 4: Performing two-dimensional range searching.

T by means of rotations. When performing a rotation at a node μ, we rebuild the secondary structures of the nodes involved in the rotation, which takes time proportional to the number of leaves in the subtree of μ. Hence, by using a weight-balanced tree for T, the amortized rebalancing time is $O(\log n)$, and thus the amortized insertion time is $O(\log^2 n)$. Analogous considerations hold for deletions.

The query and update times can be reduced using fractional cascading. Here, each node μ of T stores catalog $P(\mu)$, and query subgraphs are root-to-leaf paths. In the static case we obtain $O(\log n + k)$ query time [103,168,170]. In the dynamic case we have $O(\log n \log\log n + k)$ query time and $O(\log n \log\log n)$ update time [111].

This technique readily extends to d-dimensional space R^d, yielding a data structure with $O(n \log^{d-1} n)$ space requirement, $O(\log^{d-1} n \log\log n)$ update time, and $O(\log^{d-1} n \log\log n + k)$

query time ($O(\log^{d-1} n + k)$ in the static case).

Mehlhorn [109] intorduces a variation of the above data structure, the *range tree with slack parameter m*: instead of augmenting each node with a secondary structure, only the nodes with depth divisible by m have secondary structures. Smid [154] gives a complete analysis of the data structure, and shows that the space used is $O(n(\log n/f(n))^{d-1})$, the query time is $O((2^{f(2n)}/f(n))^{d-1}\log^d n + k)$ and the update time is $O(\log^d n/f(n)^{d-1} + \log^{d-1} n/f(n)^{d-3})$ amortized, where $f(n)$ plays the role of the slack parameter and is a smooth non-decreasing integer function such that $f(n) = O(\log n)$. In this way, we obtain many interesting trade-offs between space and query time: for $f(n) = 1$, we get the original range tree as described earlier (whose query and update time can be further improved by fractional cascading as stated before); for $f(n) = \lceil \epsilon \log n/(d-1) \rceil$, we get a linear size data structure with query time $O(n^\epsilon \log n + k)$ and amortized update time $O(\log^2 n)$; for $f(n) = \lceil \epsilon \log\log n/(d-1) \rceil$, we get a range tree of size $O(n(\log n/\log\log n)^{d-1})$, with query time $O(\log^{d+\epsilon} n/(\log\log n)^{d-1} + k)$ and amortized update time $O(\log^d n/(\log\log n)^{d-1})$.

4.3 Priority Search Tree

The *priority search tree* of McCreight [108] efficiently supports a restricted type of range queries, where the query rectangle has at least one side at infinity. It is a hybrid of a heap (for the y-coordinates) and of a balanced search tree (for the x-coordinates).

A static priority search tree T for a set P of n points in the plane is built as follows. The root ρ of T stores the highest point of P, denoted $p(\rho)$, and the median x-coordinate of the remaining points, denoted $x(\rho)$. The left subtree is a priority search tree for the points of $P - \{p(\rho)\}$ to the left of the vertical line $x = x(\rho)$, and the right subtree is a priority search tree for the remaining points.

First, we show how to perform a range query for a query rectangle $R = (-\infty, x^*) \times (y^*, \infty)$ unbounded on top and to the left. If $p(\rho)$ lies in R, then report it, else if $p(\rho)$ is below y^*, then stop. If $x^* < x(\rho)$, then recursively call the procedure on the left subtree, since all the points in the right subtree are outside R. Else ($x^* \geq x(\rho)$), recursively call the procedure on the left and right subtrees. (Note that the recursive calls in the left subtree scan the subtree from top to bottom and terminate whenever a point below y^* is reached.) The entire query takes $O(\log n + k)$ time, using $O(n)$ space, where k is the number of points reported.

This procedure can be generalized to handle ranges with three sides (unbounded on top) by traversing two paths instead of one, with the time bound remaining the same. Note that the heap structure of the priority search tree is based on the top-down order of the points, so that it supports queries in rectangles unbounded on top; if the query rectangles are unbounded on bottom, we build the tree using the bottom-up order. The other two cases (unbounded to the left or right) are analogous.

The priority search tree discussed so far is static; to dynamize it, we replace the fixed tree structure with a red-black tree T, whose leaves store the points of P sorted by x-coordinate. Each internal node also stores the highest point among the leaves below it that is not stored in any ancestor of that node (heap property). Note that some interior nodes may not store any point. Since the red-black tree has height $O(\log n)$, the query time is $O(\log n + k)$ with space complexity $O(n)$. To insert a point, we add a new leaf, restructure the path from the leaf to the root to maintain the heap property, and perform the necessary rotations. A rotation at a node μ requires updating the path from μ to the root. Since a red-black tree can be rebalanced with only $O(1)$ rotations, insertion of a point takes $O(\log n)$ time. To delete a point, we remove from the tree the nodes (a leaf and at most one internal node) storing the point. Hence, deletion of a point takes $O(\log n)$ time.

4.4 Overview of Dynamic Techniques for Range Searching

Bentley [15,16] shows that there is a static data structure for d-dimensional range searching with $P(n) = S(n) = O(n \log^{d-1} n)$ and $Q(n) = O(\log^d n)$. By treating range searching as a decomposable searching problem and applying the static-to-dynamic transformation of Bentley and Saxe [20], one gets a technique with query time $O(\log^{d+1} n)$, amortized update time $O(\log^d n)$, and space $O(n \log^{d-1} n)$. Lueker and Willard [102] illustrate an ad hoc construction using the Nievergelt-Reingold [119] notion of bounded balance, which gives a data structure with $O(\log^d n)$ query time and amortized update time, using space $O(n \log^{d-1} n)$. Willard and Lueker [169] give a general transformation method that adds range searching capability to any dynamic data structure for a decomposable searching problem with a factor of $O(\log n)$ increase in time and space usage; their method improves the update time of [102] from amortized to worst-case, while all the other bounds remain the same. The query and update time can be further improved by using fractional cascading, as described in Section 4.2.

Willard [172] studies the following variation of range search: let P be a set of n points in d-dimensional space, each with a value from a semigroup. An *aggregate query* consists of computing the semigroup sum of the values of the points in the query range. The technique of [172] modifies the *first-generation k-fold tree* of Bentley and Shamos [21] and achieves $O(\log^d n)$ query and update time, with space $O(n(\log n / \log\log n)^{d-1})$.

The idea of *k-dimensional binary trees*, or kd-trees, was first presented in [13]. It is a k-dimensional generalization of balanced search trees. For simplicity, consider the case $k = 2$. First, the point set is partitioned evenly by the vertical line through the median of the x-coordinates. Next, each of the two halves is further split evenly by the horizontal line through the median of its y-coordinates, and so on, letting the direction of the cutting line alternate at each step. This decomposition process is represented by a binary tree. Detailed discussion on kd-trees can be found in [13,84,96,133]. Dynamic *k-d* trees are discussed in Overmars' thesis [121] (section 5.3.2). A local reorganization technique improving the performance of kd-trees is presented in [47].

Bentlely and Mauer [19] and Willard [171] describe linear space data structures of pragmatic interest. [19] gives the historically first data structure to obtain $O(n^\epsilon)$ query time for range searching in $O(n)$ space, and the shearing method of [171] is an interesting alternative to [19]. All these data structures have no worse than $O(\log^2 n)$ update time under the static-to-dynamic transformation.

A *functional approach* that results in dynamic data structures with linear size for aggregate query, range counting and range reporting is given by Chazelle [30]. The query and update time are both $O(\log^4 n)$ for aggregate query, both $O(\log^2 n)$ for range counting, and respectively $O(k(\log(2n/k))^2)$ and $O(\log^2 n)$ for range reporting.

Van Kreveld and Overmars [92,93] give two methods for range searching. In [93], they study techniques to make existing data structures concatenable, and obtain a d-dimensional data structure for range searching that supports queries in $O(n^{1-1/d} \log n + k)$ time, insertions and deletions in $O(\log n)$ time, and splits and concatenations on all coordinates in $O(n^{1-1/d} \log n)$ time, using linear space. In [92], they present a variant of the k-d tree, called *divided k-d tree*, to achieve the following performance: in the plane, $O(\sqrt{n \log n} + k)$ time for range queries, $O(\log n)$ time for insertions/deletions, using $O(n)$ space; in the d-dimensional case, range queries take $O(n^{1-1/d} \log^{1/d} n + k)$ time, while all the other bounds remain the same.

Scholten and Overmars [144] considers general methods for adding range restrictions to decomposable searching problems. They describe two classes of data structures for this problem, and dynamize them by using a new type of weight-balanced multiway trees as underlying structures. They thus obtain efficient dynamic data structures for orthogonal range searching, with trade-offs.

Klein *et al* [91] study the following *dynamic fixed windowing problem* for planar point set: for a given planar point set S under insertions and deletions and a fiexed window W (a fixed planar region), perform the *window-queries*, that is, for an arbitrary query translate $W_q = W + q$, report all points of S that lie in W_q. They give a dynamic data structure based on priority search trees, and achieve $O(n)$ space, $O(\log n)$ update time and $O(\log n + k)$ query time for fixed polygonal window W, where k is the size of the reported answers.

Icking, Klein and Ottman [81] have studied the problem of realizing priority search trees in external memory, where data are transferred in blocks of size B. They present two techniques derived from B-trees and a generalized version of red-black trees. Smid [151], Overmars, Smid, de Berg and van Kreveld [128] and Smid and Overmars [157] consider the problem of maintaining range trees in secondary memory. They give a number of schemes to partition range trees into parts that are stored in consecutive blocks in secondary memory, and also study lower bounds for the partitions so that many of the proposed partitions are proved to be optimal.

The *half space range query* problem is the following: Given a set of points in R^d, and given a query half space, report all the points that lie in the query half space. Schipper and Overmars [143] give a technique for the planar case, by supporting weak deletions on the *conjugation tree* and then applying the global rebuilding technique of Overmars [121]. The resulting technique uses linear space, supports half plane range queries and counting queries in time $O(n^{\log_2(1+\sqrt{5})-1} + k)$ and $O(n^{\log_2(1+\sqrt{5})-1})$, respectively, with insertion time $O(\log^2 n)$ and deletion time $O(\log n)$, where k is the number of the reported answers.

Mulmuley [115] gives a randomized algorithm for the half space range queries, under the "comminust" model, where a *random sequence* of insertions and deletions is defined as follows. Given an update sequence $\bar{u}$, define its signature $\delta = \delta(\bar{u})$ to be the string of $+$ and $-$ symbols of the same length, where the $+$ symbols correspond to insertions in $\bar{u}$ and $-$ symbols correspond to deletions.

We say that $\bar{u}$ is an (N, δ)-sequence, where N is the set of objects involved in $\bar{u}$, and the size of N is m. We say that $\bar{u}$ is a random (N, δ)-sequence if it is chosen from a uniform distribution on all valid (N, δ)-sequences. Let k be the number of points that lie in the query half space. The expected query time is $O(k \log n)$, and the total expected cost of maintaining the data structure over a random (N, δ)-sequence is $O(m^{\lfloor d/2 \rfloor + \epsilon})$ for any $\epsilon > 0$.

Matoušek [106] consider the following *simplex range searching* problem: given a set of points in R^d, each with a semigroup weight, report the points or compute the cumulative weight of the points that lie in a given query simplex. He proposes an efficient partition scheme for point sets, and applies the dynamization techniques for decomposable searching problems of [15] and [121] to get a data structure with $O(n)$ space, $O(n^{1-1/d}(\log n)^{O(1)})$ query time, $O(\log n)$ amortized deletion time and $O(\log^2 n)$ amortized insertion time.

5 Intersections

5.1 Segment Tree

The *segment tree* data structure was introduced by Bentley [14]. Let X be a set of N points on a line, and S a set of n segments with endpoints in X. A *segment tree* $T = T(X, S)$ supports the following operations:

- report all the segments of S that contain a query point x (point enclosure);
- count the number of segments of S that contain a query point x;
- insert a new segment, with endpoints in X, into S;
- remove a segment from S.

The primary component of segment tree T is a balanced binary tree with $N - 2$ internal nodes and $N - 1$ leaves. Each leaf λ of T corresponds to an elementary interval $I(\lambda)$ between consecutive points of X, and each internal node μ corresponds to a nonextreme point $x(\mu)$ of X. Also, each node μ of T is associated with the interval $I(\mu)$ formed by the union of the elementary intervals of the leaves in the subtree of μ (see Fig. 5). Given a segment s with endpoints in X, the *allocation nodes* of s are the nodes of T such that s contains $I(\mu)$ but not $I(\nu)$, where ν is the parent of μ (see Fig. 6). It can be shown that there are at most two allocation nodes of s at any level of T, so that s has $O(\log N)$ allocation nodes.

To answer report-queries we add to T a secondary component that stores at each node μ the point $x(\mu)$ and the set of segments $S(\mu)$ that are allocated at μ (see Fig. 5). The k segments containing a query point x are the segments allocated at the nodes in the path of T from the root to the leaf λ such that $x \in I(\lambda)$, and thus can be reported in $O(\log N + k)$ time using $O(N + n \log N)$ space. Count-queries are answered by replacing the set of proper segments at each node μ with its size, so that queries take $O(\log N)$ time using $O(N)$ space. Inserting or deleting a segment takes $O(\log N)$ time.

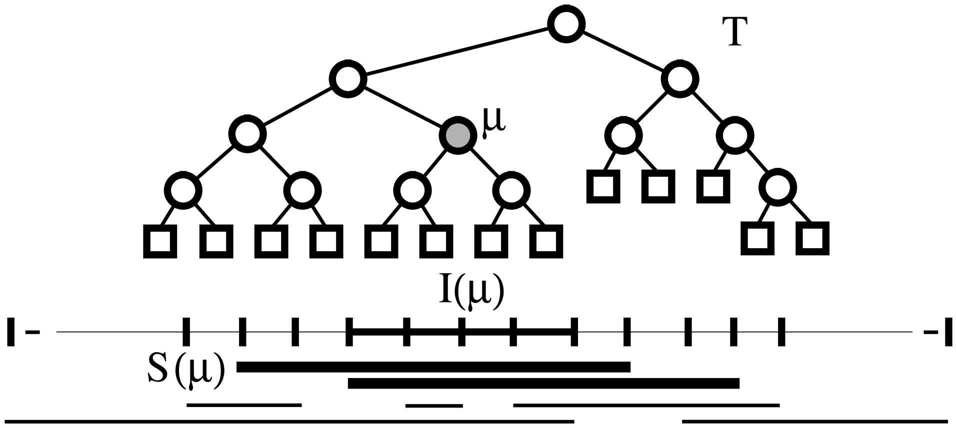

Figure 5: The interval $I(\mu)$ and the set $S(\mu)$ of a node μ of a segment tree.

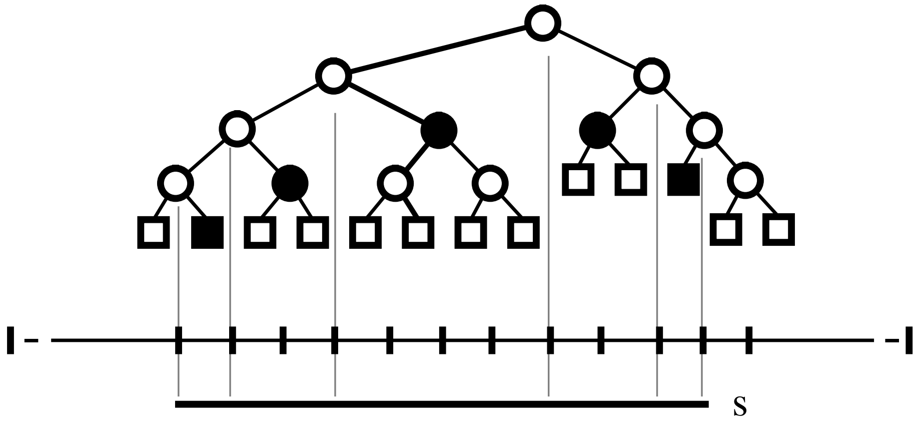

Figure 6: The allocation nodes of segment s.

We can remove the restriction that the set X of endpoints be fixed by using a weight-balanced tree for the primary structure of T. A rotation at a node μ takes time proportional to the number of leaves in the subtree rooted at μ. Hence, rebalancing takes $O(\log n)$ amortized time, and update operations take $O(\log n)$ amortized time.

5.2 Concatenable Segment Tree

A *concatenable* version of the segment tree is given by van Kreveld and Overmars [94], which can be used to answer the following one-dimensional *stabbing queries*: given a query segment s on the same line where the segments of S lie, report the k segments in S intersected by s. In addition to the stabbing queries and standard updates (insertion and deletion of segments), the data structure supports split and concatenate operations. Namely, a concatenate operation joins two segment trees T_1 and T_2 such that each segment in T_1 is to the left of each segment in T_2, and a split operation is the inverse of a concatenation. The space requirement is $O(n \log n)$, a stabbing query takes $O(\log n + k)$ time, insertions, concatenations, and splits take $O(\log n)$ amortized time, and deletions take $O(\log n \cdot a(i, n))$ amortized time, where $a(i, n)$ is the row inverse of Ackermann's function $A(\cdot, \cdot)$, i.e., $a(i, n) = \min\{j \mid A(i, j) \geq n\}$ for a fixed constant i; for example, $a(i, n) = \Theta(\log n)$ for $i = 1$ and $a(i, n) = \Theta(\log^* n)$ for $i = 2$. This method is based on a new *union-copy* data structure for sets, which generalizes the well-known union-find structure.

5.3 Interval Tree

Here we present the *interval tree*, a data structure due to Edelsbrunner [55], which is called a *1-fold rectangle tree* in the original paper. Let X be a set of N points on a line, and S a set of n segments with endpoints in X. An *interval tree* T for X and S is a data structure that supports the following operations:

- report all the segments of S that contain a query point x (point enclosure);
- insert a new segment, with endpoints in X, into S;
- remove a segment from S.

The primary structure of T is a balanced binary tree whose internal nodes store the points of X, sorted from left to right, and whose leaves represent intervals between consecutive points of X. Each segment s of S is allocated at the least common ancestor of the nodes associated with the endpoints of s. The set of segments allocated at a node μ, denoted by $S(\mu)$, is represented by two linked lists that store the left endpoints sorted from left to right, and the right endpoints sorted from right to left. Hence, the space requirement is $O(n + N)$.

A query for point x is answered by traversing a path in T from the root to the node whose associated point or interval contains x. At each node μ of this path we scan the left or right endpoint list of $S(\mu)$ depending on whether the left or right child of μ is on the path. The time spent at node μ is $O(1 + k_\mu)$, where k_μ is the number of segments of $S(\mu)$ containing point x. Hence the total query time is $O(\log N + k)$. To support insertion and deletion of segments, we replace the endpoint lists with inorder-threaded balanced search trees. Hence, the update time is $O(\log N + \log n)$.

The restriction on the fixed set of endpoints can be removed, as usual, by using a $BB[\alpha]$ tree for the primary structure. We obtain $O(n)$ space, $O(\log n + k)$ query time, and $O(\log n)$ amortized update time.

The interval tree can be modified to support one-dimensional segment intersection queries, which consist of reporting the segments intersected by a given query segment.

The technique of making data structures concatenable due to van Kreveld and Overmars [93] results in a concatenable interval tree that supports stabbing queries in $O(\sqrt{n} \log n + k)$ time, insertions, deletions, splits, and concatenations in $O(\sqrt{n} \log n)$ time, with $O(n)$ space.

5.4 Orthogonal Segment Intersection Queries

Consider a set S of n horizontal segments in the plane. An *orthogonal segment intersection query* on S consists of reporting the k segments of S intersected by a given vertical query segment.

The segment tree can be extended to support orthogonal segment intersection queries by representing each set $S(\mu)$ with a balanced search tree sorted by y-coordinates. This data structure uses $O(n \log n)$ space and supports queries in time $O(\log^2 n + k)$. The update time is $O(\log^2 n)$. Static fractional cascading can be used to improve the query time to $O(\log n + k)$ [167].

Lipski [100,101] shows how to achieve $O(\log n \log\log n + k)$ query time, $O(\log n \log\log n)$ update time and $O(n \log n)$ space by applying the techniques of van Emde Boas [26,27] to the secondary structures of the segment tree. However, updates are restricted to a fixed set of $O(n)$ coordinates for the endpoints, and the query segments are known in advance.

Dynamic fractional cascading eliminates these restrictions and achieves $O(n \log n)$ space, $O(\log n \log\log n + k)$ query time, and $O(\log n \log\log n)$ update time [111] for the general problem.

Imai and Asano [82,83] give two semi-dynamic techniques for the case in which updates are restricted to insertions only or deletions only, respectively; both techniques require that the y-coordinates of the segments be known in advance. The linear-time set-union and set-splitting algorithms of Gabow and Tarjan [68] are used for the secondary structures of the segment tree. They achieve $O(n \log n)$ space, $O(\log n + k)$ query time, and $O(\log n)$ amortized update time.

McCreight [108] gives an $O(n)$-space data structure for orthogonal segment intersection with $O(\log^2 n + k)$ query time and $O(\log n)$ update time, using priority search trees as the secondary structure of an interval tree [108]. Updates are restricted to a fixed set of $O(n)$ coordinates for the segment endpoints. This restriction can be removed by using a $BB[\alpha]$ tree for the primary interval tree.

Cheng and Janardan [37] give an algorithm for orthogonal segment intersection queries in the plane, with $O(n)$ space, $O(\log^2 n + k)$ query time and $O(\log n)$ insertion/deletion time. They propose a solution for a special kind of *segment intersection* problem, where a query segment is of a fixed slope (discussed in Section 5.5). Then the orthogonal segment intersection problem is treated as a special case of their proposed problem.

Orthogonal rectangle intersection queries consist of reporting the rectangles of a set R intersected by a query rectangle r, where rectangles are of the type $[a_1, b_1] \times [a_2, b_2]$. We have that a rectangle r' of R is intersected by r if and only if one of the following mutually exclusive cases arises (see Fig. 7):

(a) the bottom-left corner of r' is in r;

(b) r' contains the bottom-left coner of r;

(c) the bottom side of r' is intersected by the left side of r;

(d) the left side of r' is intersected by the bottom side of r.

Therefore, with a combination of segment and range trees, we can reduce the orthogonal rectangle intersection queries to range search queries for the bottom-left corners of the rectangles of R contained in r (case (a)), point enclosure queries for the rectangles of R containing the bottom-left corner of r (case (b)), and orthogonal segment intersection queries (cases (c) and (d)).

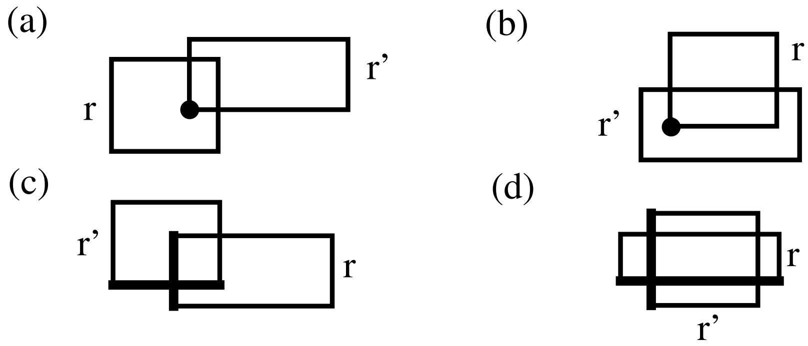

Figure 7: Four mutually exclusive cases for rectangle intersection.

The d-dimensional orthogonal rectangle intersection queries can be computed using a *d-fold rectangle tree* [54,55], which is based on the interval tree. The d-fold rectangle tree uses $O(n \log^{d-1} n)$ space and supports queries in time $O(\log^{2d-1} n + k)$. *Off-line updates*, in which the elements to be inserted and deleted belong to a fixed set of size n, are supported in $O(\log^d n)$ time.

5.5 Ray Shooting

Let S be a set of segments (with arbitrary directions and possibly intersecting) in the plane. *Segment intersection queries* ask one to report the segments of S intersected by an arbitrary query segment. *Ray-shooting queries* ask one to determine the first segment hit by an arbitrarily-directed ray r emanating from a query point q.

Schipper and Overmars [143] give a technique for segment intersection queries that works for the case where the segments in S do not intersect each other. They apply the dynamization technique of decomposable searching problem to *segment partition trees*, and achieve the followsing result: $O(n \log n)$ space, $O(\log^2 n)$ insertion/deletion time and $O(n^{\log_2(1+\sqrt{5})-1} + k)$ query time, where k is the number of the reported answers.

Cheng and Janardan [40] improve and dynamize the static techniques of [3] for segment intersection and ray-shooting. Their method makes extensive use of a *spanning path of low stabbing number*, which is similar in concept to a *spanning tree of low stabbing number*, illustrated as follows: Let S' be a set of n points in the plane. A *spanning tree of stabbing number $c(n)$* for S' is a spanning tree for S' whose edges are line segments, such that any line stabs (i.e. intersects) only $c(n)$ edges of the tree. Let $ST(n)$ be the time required to construct such a tree. It is shown in [36] that a spanning tree of stabbing number $c(n)$ can be transformed in linear time into a possibly self-intersecting spanning path with stabbing number at most $2c(n)$, thus the time $SP(n)$ to construct such a path is $O(ST(n))$. Let $w(n)$ be the working storage needed to construct a spanning path of stabbing number $c(n)$. The data structure of [40] uses space $O(n \log^2 n + w(n))$ and supports segment intersection and ray-shooting queries in $O(c(n) \log^2 n + k \log^2 n)$ and $O(c(n) \log^2 n)$ time, respectively. If $SP(n) = \Omega(n^\delta)$ for some $\delta > 1$, then the update time is $O(SP(n)/n)$, otherwise the deletion time is $O((SP(n)/n) \log n + \log^4 n)$ and the insertion time is $O((SP(n)/n) \log^2 n + \log^4 n)$. The actual performance bounds are obtained by combining the results on constructing a spanning tree of low stabbing number. For example, Matoušek [105] gives an algorithm with $ST(n) = O(n\sqrt{n} \log^2 n)$ and $c(n) = O(\sqrt{n})$, Agarwal and Sharir [2] give a method with $ST(n) = O(n \log n)$ and $c(n) = O(n^{\frac{1}{2}+\epsilon})$ for any $\epsilon > 0$, and Matoušek [106] presents a result with $ST(n) = O(n \log n)$ and $c(n) = O(\sqrt{n} 2^{O(\log^* n)})$.

In addition to giving an algorithm for constructing a spanning tree of low stabbing number, Agarwal and Sharir [2] also show how to perform segment intersection and ray-shooting in the plane dynamically, based on the space partitioning technique of Chazelle, Sharir and Welzl [35]. For any given $\epsilon > 0$ and a storage parameter s such that $n^{1+\epsilon} \leq s \leq n^{2+\epsilon}$, their data structure has size s, supports insertions and deletions of segments in amortized time $O(s/n^{1-\epsilon})$, and answers segment intersection and ray-shooting queries in $O(n^{1+\epsilon}/\sqrt{n} + k)$ and $O(n^{1+\epsilon}/\sqrt{n})$ time, respectively.

Cheng and Janardan [37] consider two special kinds of segment intersection problems: given a set S of nonintersecting but possibly touching line segments in the plane under insertions and deletions, report the k segments that are intersected by (i) a query segment of fixed slope; (ii) a query segment whose supporting line passes through a fixed point. They use their point location data structure ([38]) to solve a dynamic visibility problem, and obtain the following performance: $O(n)$ spcae, $O(\log n)$ update time, and $O(\log^2 n + k)$ time for both kinds of queries. Since the orthogonal segment intersection queries (see Section 5.4) are a special case of the queries of the first kind, they are also solved within the same complexity bounds.

Chiang, Preparata and Tamassia [41] present a fully dynamic technique for ray-shooting and segment intersection in a connected planar subdivision, using a unified approach (see Section 6.2). In particular, a *location structure* and a *hull structure* are used to perform the queries, and a *normalization structure* is used to ensure efficient updates, which consist of insertions and deletions of vertices and edges. The data structure uses space $O(n \log n)$, supports ray-shooting and segment intersection queries in time $O(\log^3 n)$ and $O(k \log^3 n)$, respectively, with update time $O(\log^3 n)$ worst-case for edge updates and amortized for vertex updates.

6 Point Location

Planar *point location* is a classical problem in computational geometry. Given a *planar subdivision* P with n vertices, i.e., a partition of the plane into polygonal regions induced by a planar graph drawn with straight-line edges, we want to determine the region of P that contains a given query point q. If q lies on a vertex or edge of P, then that vertex or edge is reported (see Fig. 8). Monotone, convex, and triangulated subdivisions are planar subdivisions such that every region is a monotone polygon, a convex polygon, or a triangle, respectively. By Euler's formula for planar graphs, a planar subdivision with n vertices has no more than $3n - 6$ edges and $2n - 4$ regions.

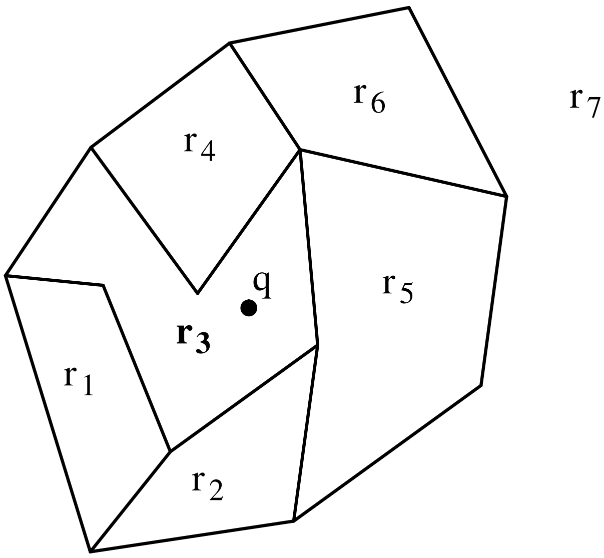

Figure 8: An example of planar point location: region r_3 is reported for a given query point q.

Point location is closely related to a variation of the ray-shooting problem. Namely, if we store with each edge of a planar subdivision P the name of the region above it, we can locate a query point q by finding the first edge hit by a downward vertical ray originating at q.

Optimal solutions for static planar point location are presented in [57,89,142]. All of them have $O(\log n)$ query time, $O(n \log n)$ preprocessing time, and $O(n)$ space requirement; by Chazelle's triangulation result [31], the preprocessing time of [57] and [89] can be further improved to $O(n)$ for connected subdivisions.

Update operations for planar point location consist of inserting or deleting a vertex or an edge into the subdivision. While edge insertion is uniquely defined, vertex insertion can be performed either by creating an isolated vertex, or by inserting a vertex along an existing edge, which is split into two edges. The dual operations of edge insertion and deletion are also useful [76]: an *expansion* splits a vertex v into two new vertices connected by an edge; each new vertex inherits a subsequence of the edges formerly incident on v. A *contraction* merges two adjacent vertices v_1 and v_2 into a new vertex, which inherits the edges incident on both v_1 and v_2.

6.1 Dynamic Point Location with a Segment Tree

In this section we describe a simple dynamic point-location method for connected subdivisions, based on the segment tree. It is reported in Overmars [122]. The update operations supported are inserting and deleting edges with at least one endpoint vertex already in the subdivision. This method uses $O(n \log n)$ space and has $O(\log^2 n)$ query and update times.

The data structure consists of two components: a segment tree T storing the segments of P, and a collection of balanced trees, called *boundary trees*, representing the boundaries of the regions of P. The segment tree is used to answer ray-shooting queries for downward rays, while the boundary trees allow one to determine the region above a given edge.

The primary structure of tree T is based on the x-coordinates of the vertices of P. The allocation of the edges of P to the nodes of T is done by considering the projections of the edges to the x-axis. Hence, for every node μ of T, the proper edges of μ are nonintersecting segments that span the vertical strip defined by the x-interval $I(\mu)$. The secondary structure of T stores at each node μ a balanced tree $S(\mu)$ storing the proper edges of μ sorted from bottom to top. Inserting and deleting edges correspond to standard segment tree operations on T, and thus take $O(\log^2 n)$ time, amortized for updates that add a new vertex or remove an existing vertex.

A query in T identifies the first edge of P hit by a downward vertical ray originating at query point q as follows (see Fig. 9): Trace a path from the root of T to the leaf λ whose elementary interval contains the x-coordinate of q. For each node μ of this path, determine the highest edge below q by searching in the balanced tree $S(\mu)$. Among the edges found by the previous step, return the one closest to q. Hence, a query takes time $O(\log^2 n)$, since $O(\log n)$ tree searches are performed, each taking $O(\log n)$ time.

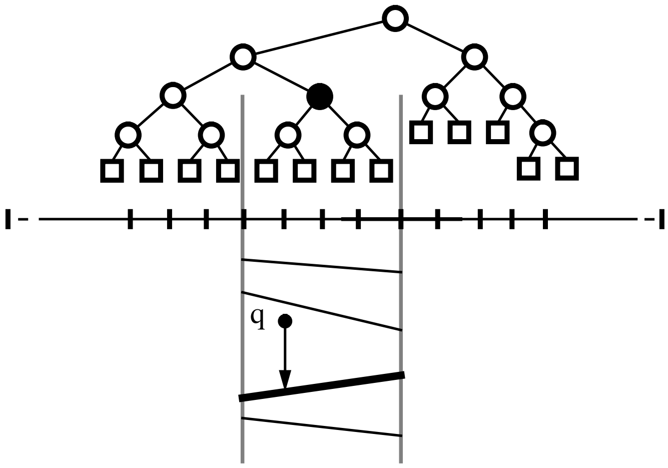

Figure 9: Point location with a segment tree.

The boundary tree for a region r is a balanced tree $B(r)$ storing the edges on the boundary of r sorted according to their circular sequence (starting at any edge). The root of $B(r)$ stores the name of r. Every edge e has exactly two representatives in the boundary trees of the regions on the two sides of e. We store with each edge e a pointer to its representative e' in the boundary tree of the region above e. Hence, given an edge e we can determine the region above e by accessing the representative e' 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. Adding and removing edges correspond to performing $O(1)$ concatenable-queue operations on the boundary trees, which take time $O(\log n)$.

6.2 Overview of Dynamic Point Location Techniques

The dynamic point-location technique of Overmars [122] is described in the previous section. As presented in the original paper, it deals with subdivisions whose regions have a bounded number of edges, such as triangulations. However, this restriction can be removed by using the boundary trees as discussed above.

Fries and Mehlhorn [66,67] present a data structure for connected 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 [95]. The update operations supported are inserting and deleting edges and isolated vertices. If only insertions are performed, the update time is reduced to $O(\log^2 n)$ [109 (pp. 135–143)].

Preparata and Tamassia give two dynamic techniques for monotone and convex subdivisions, respectively [135,136].

The data structure for monotone subdivisions [135] is based on the chain-method [95]. The repertory of update operations includes 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). All bounds are worst-case [135]. As shown in [134,137], the data structure can also be extended to support expansion and contraction operations in time $O(\log^2 n)$, with applications to three-dimensional point location.

The data structure for convex subdivisions [136] is based on the trapezoid method [131]. 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 $O(\log n \log N)$ [136].

Tamassia presents a technique for point location in triangulations that allows a tradeoff between query and update time, and can be used in conjunction with any of the known static point location data structures [159]. It is based on the dynamization methods for decomposable search problems [20,121]. The update operations are inserting a "star" (vertex and three edges) inside a region (which partitions it into three new regions), and swapping the diagonal of a convex quadrilateral formed by adjacent regions. Tamassia shows 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. 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 [38] present two methods for dynamic planar point-location in connected subdivisions. 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 of their methods use $O(n)$ space and are based on a search strategy derived from the priority search tree [108] (see Section 4.3). They dynamize this approach with the $BB[\alpha]$ tree data structure [110], using the approach of Willard and Lueker [169] to spread local updates over future operations, and the method of Overmars [121] to perform global rebuilding (at the same time) before the "current" data structure becomes too unbalanced. As emphasized by the authors, such methods are mainly of theoretical interest, because they involve rather complex manipulations of data structures.

Goodrich and Tamassia [70] show how to maintain a monotone subdivision dynamically 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 with k edges. This method is based on a new optimal static point-location data structure, which uses interlaced spanning trees, one for the subdivision and one for its graph-theoretic dual, to answer queries. Queries are performed by using a centroid decomposition of the dual tree to drive searches in the primal tree. The link-cut tree data structure of Sleator and Tarjan [149] is used to dynamize the method, which is relatively easy to implement. This approach also allows one to implement the dual operations of *expand* and *contract*, with applications to three-dimensional point location. A variation of this method supports updates in a semi-dynamic environment in which only insertions are performed. In this case the centroid decomposition of the dual tree is explicitly maintained in a $BB[\alpha]$ tree [110], and a simple version of fractional cascading [34] is applied to improve the query time to $O(\log n \log\log n)$ while also improving the complexity of updates to $O(1)$ amortized time.

Chiang and Tamassia [42] present a fully dynamic data structure for point location in a monotone subdivision, based on the trapezoid method [131]. The operations supported are insertion and deletion of vertices and edges, and horizontal translation of vertices. It extends the previous dynamic trapezoid method of Preparata and Tamassia [136] in two aspects: it supports monotone subdivisions instead of convex subdivisions, and removes the restriction that the vertices lie on a fixed set of horizontal lines. To achieve the first extension, it introduces a new type of partitioning objects, called *spanning tangents*, and modifies the dynamic convex hull technique of Overmars and van Leeuwen [124] to compute and maintain the spanning tangents. To achieve the second extension, a $BB[\alpha]$ tree is used to control the cost of rebuilding the secondary structures involved in updates. It achieves optimal query time $O(\log n)$, update time $O(\log^2 n)$ and space $O(n \log n)$, where the time bounds are amortized for insertion and deletion of vertices, and worst-case for the others.

Smid [155] considers the d-dimensional rectangular point location problem, where a set of n non-overlapping d-dimensional axes-parallel hyperrectangles, or *d-boxes*, are stored in a data structure to support point location queries. The proposed data structure is based on the *skewer tree* of Edelsbrunner, Haring and Hilbert [58] and uses dynamic fracional cascading [111]. It uses linear space, answers queries in $O(\log^{d-1} n \log\log n)$ time, and supports insertion, deletion, splitting and merging of d-boxes in $O(\log^2 n \log\log n)$ amortized time.

Several randomized algorithms are given by Mulmuley and Sen [118] and Mulmuley [115,116]. [118] presents two algorithms for dynamic point location in arrangements of d-dimensional hyperplanes, where the updates consist of insertions and deletions of hyperplanes. Their algorithms use random bits in the construction of the data structure but do not make any assumption about the update sequence or the hyperplanes in the input. The algorithms are simple and are based on the *dynamic sampling* technique. For the first algorithm, the query time is $\tilde{O}(polylog(n))$, the space requirement is $\tilde{O}(n^d)$, the update time is $\tilde{O}(n^{d-1}\log n)$, and the expected update time is $O(n^{d-1})$, where $\tilde{O}$ notation indicates that the bound holds with high probability $1 - 1/n^{\alpha}$ for some $\alpha \geq 1$. The probability and the expectation are solely with respect to randomization in the data structure. They also give a related algorithm that supports queries in optimal $\tilde{O}(\log n)$ time, with expected $O(n^d)$ space and amortized $O(n^{d-1})$ expected update time. Mulmuley [116] gives yet another algorithm for the same problem, based on *dynamic shuffling*, with $\tilde{O}(polylog(n))$ query time, $O(n^d)$ space (deterministic), and $O(n^{d-1})$ expected update time, where the expectation is again solely with respect to randomization in the data structure. Mulmuley [115] presents a dynamic point location algorithm in three dimensional partitions induced by a set of possibly intersecting polygons in R^3. This is a generalization of the randomized algorithm for the planar case given in [117]. The algoritm is based on random sampling and achieves $\tilde{O}(\log^2 n)$ query time, with the expected running time on a random (N, δ)-sequence of updates (see Section 4.4) close to optimal.

Devillers, Teillaud and Yvinec [48] describe a randomized algorithm for dynamic point location in an arrangement of line segments in the plane. They generalize the *influence graph* data structure of [28] to dynamically maintain the trapezoidal map of an arrangement of segments, and assume that the insertion sequences are evenly distributed among the $n!$ possible sequences of n segments, and any already inserted segment can be deleted with the same probability. Let a denote the current size of the arrangement. The data structure requires $O(n+a)$ expected space and supports queries in $O(\log n)$ expected time. A segment can be inserted in $O(\log n + a/n)$ expected time and deleted in $O((1 + a/n)\log\log n)$ expected time.

Baumgarten, Jung and Mehlhorn [10] give a fully dynamic data structure for planar point location in general subdivisions, where the updates consist of insertions and deletions of non-intersecting, except possibly at endpoints, line segments. They combine interval trees, segment trees, fractional cascading and the data structure of [38], and achieve $O(n)$ space, $O(\log n \log\log n)$ query and insertion time and $O(\log^2 n)$ deletion time, where the time bounds for updates are amortized.

Chiang, Preparata and Tamassia [41] present a fully dynamic technique for planar point location in connected subdivisions, where the update operations are insertions and deletions of vertices and edges. The central idea of the technique is to transform (i.e., *normalize*) a connected subdivision into a monotone subdivision by adding horizontal segments (called *lids*), under the control of a *normalization structure*, so that each inserted edge crosses $O(\log n)$ lids and thus the size of the update is controlled. A variation of the dynamic tree of Sleator and Tarjan [149] is used to maintain the normalization structure, where each solid path is represented by double threads. Moreover, a *hull structure*, which is a modification of the dynamic convex hull structure of Overmars and van Leeuwen [124], is augmented to the normalization structure, so that the *spanning tangents* introduced in Chiang and Tamassia [42] can be computed efficiently. Point location queries are supported by a separate *location structure*, which is a dynamic version of the trapezoid tree [131]: it is not explicitly balanced but instead represented by a dynamic tree [149]. The method achieves optimal query time $O(\log n)$, with space $O(n \log n)$ and update time $O(\log^3 n)$, where the time bounds are amortized for the vertex update and worst-case for the other update operations. This unified approach also supports fully dynamic *ray-shooting* queries (see Section 5.5), and *shortest path* queries (see Section 9.4).

7 Convex Hull

Given a set of points S in d-dimensional space, the *convex hull* H of S is defined as the smallest convex polygon containing all the points of S (see Fig. 10 for a two-dimensional example). Computing the convex hull of a set of points is not a searching problem, so the corresponding dynamic problem needs to be defined. In a dynamic environment, we consider a set S of points that is updated by means of insertions and deletions. The following query operations arise naturally:

- find if a given point of S is on the convex hull H of S;
- find if a query point is internal or external to the convex hull H of S;
- find the tangents to the convex hull H of S from an external query point;
- find the intersection of the convex hull H of S with a given query line;
- report the points on the convex hull H of S.

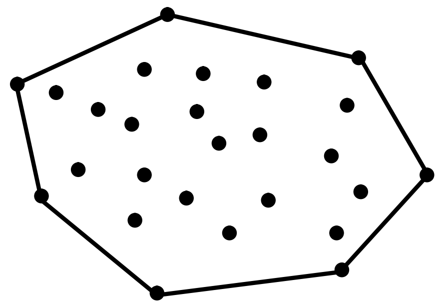

Figure 10: An example of two-dimensional convex hull.

7.1 Semi-Dynamic Maintenance of Planar Convex Hulls

In this section we present a variation of the technique of Preparata [130] for maintaining the convex hull H of a set of points S in the plane under a sequence of insertions. The space requirement is $O(n)$. Each update and find-query takes $O(\log n)$ time, and a report-query takes $O(n)$ time, where

n is the number of vertices currently in the convex hull H of S. The time for the first type of query can be reduced to $O(1)$ at the expense of making the insertion time amortized.

First, we observe that we can maintain separately the upper and lower hulls of S, defined as the subchains U and L of H that are delimited by the leftmost and rightmost points of S. The vertices of L and U are stored into balanced trees T_L and T_U, sorted by x-coordinate. Find-type queries are answered in $O(\log n)$ time essentially by searching in T_L and T_U.

For example, given a query point q we can determine in $O(\log n)$ time the segments of L and U intersected by a vertical line through q by searching for $x(q)$ in in T_L and T_U. If neither of these segments exists, then q is outside the convex hull H. Otherwise, q is in H if and only if it lies vertically between these segments, which can be checked in $O(1)$ time.

Because of symmetry considerations, we describe only how to maintain the upper hull U. Let $v_1 \cdots v_{k-1}$ be the vertices of U going from left to right, and let $v_0 = (x(v_1), -\infty)$ and $v_k = (x(v_{k-1}), -\infty)$ be points at infinity. After adding a new point p we need to update the upper hull U only if p is outside H. In this case, a subchain of U is replaced by p and one or two edges incident on p that are tangent to U. If p is above U, then p has two tangents to U, called left and right tangents. If p is to the left (resp. right) of U, then p has only the right (resp. left) tangent.

To find the vertices supporting the left and right tangents from p to U, we classify the vertices of U with respect to p as follows (see Fig. 11):

- vertex v_i is *left* (resp. *right*) *supporting* if v_{i-1} and v_{i+1} are on the left (resp. right) side of the line from p to v_i;
- vertex v_i is *reflex* if v_{i-1} and v_{i+1} are on different sides of the line through p and v_i, and p is inside the external angle formed by v_{i-1}, v_i, and v_{i+1};
- v_i is *inflex* if v_{i-1} and v_{i+1} are on different sides of the line through p and v_i, and p is inside the internal angle formed by v_{i-1}, v_i, and v_{i+1}.

To find the left supporting vertex we traverse a path in the tree T_U. Let μ be the current node and v_i its associated vertex. If v_i is left supporting, we stop and return v_i. If v_i is reflex, we branch left. If v_i is inflex, then we branch left or right depending on whether p is to the left or right of v_i, respectively.

The right supporting vertex can be found similarly. After having computed the supporting vertices, we can update T_U by means of a constant number of split and splice operations.

Clearly, it is possible to determine in $O(1)$ time whether v is supporting, reflex, or inflex with respect to p. Hence, finding the supporting vertices and updating the convex hull takes $O(\log n)$ time.

Some deletion-only data structures with linear size and $O(\log n)$ amortized deletion time for planar convex hulls are given by Chazelle [29] and by Hershberger and Suri [77].

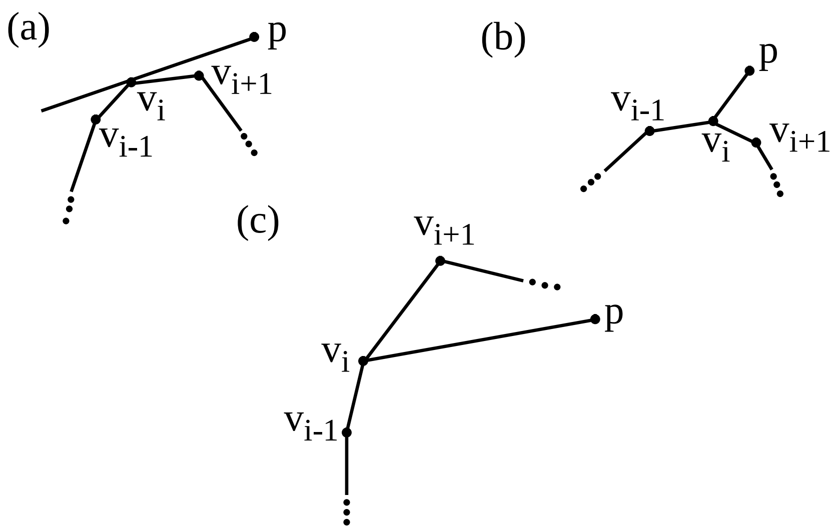

Figure 11: Classification of vertex v_i with respect to p: (a) left supporting; (b) reflex; (c) inflex.

7.2 Fully Dynamic Maintenance of Planar Convex Hulls

Now we describe the fully dynamic convex hull technique of Overmars and van Leeuwen [124]. In the insert-only scheme of the previous section, points found to be internal to the current convex hull can be thrown away; however, in the fully dynamic setting, we must keep *all* points, since the deletion of a current hull point may cause some internal points to resurface on the hull boundary. As before, we show how to maintain the upper hull (U-hull for short) U in a tree T_U; the lower hull L is dealt with similarly.

T_U is a balanced binary tree, whose leaves store the points sorted by their x-coordinates from left to right. Each internal node μ of T_U represents the U-hull U_μ of its leaves. The secondary structure of node μ stores in a balanced tree (supporting concatenable-queue operations) the points of U_μ that are not in U_ν, for any ancestor ν of μ. The data structure uses $O(n)$ space since each point is stored twice (in a leaf and in some internal node). Queries can be performed using the secondary structure of the root of T_U, which represents the U-hull of all the points.

Before discussing insertion and deletion, we first describe some basic primitive operations. Let (S_1, S_2) be a partition of S such that all the points in S_1 are to the left of those in S_2. Also, let the U-hulls U_1, U_2 of S_1 and S_2 be maintained in T_1 and T_2, respectively. To compute the U-hull U of S from U_1 and U_2, we compute the points $q_1 \in U_1$ (which partitions U_1 into U_{11} and U_{12} to the left and right of q_1, with $q_1 \in U_{11}$) and $q_2 \in U_2$ (which again partitions U_2 into U_{21} and U_{22}, with $q_2 \in U_{22}$) such that $\overline{q_1q_2}$ is the supporting line to both U_1 and U_2. The U-hull U then consists of U_{11}, $\overline{q_1q_2}$, and U_{22}; we construct the tree T_U of U by creating a root node ρ and making T_1 and T_2 the left and right subtrees of ρ. The root ρ stores the points of U_{11} and U_{22}, while the left and right children of ρ store the points of U_{12} and U_{21}, respectively. We call this operation *bridging* and its inverse operation *de-bridging*. Suppose that q_1 is the i-th point of U_1 from left to right (and thus the i-th point of U); we also store the integer i in ρ to facilitate de-bridging.

To perform bridging, we must compute q_1 and q_2. Given any two points $p_1 \in U_1$ and $p_2 \in U_2$, each of these two points can be classified with respect to segment $\overline{p_1 p_2}$ as either inflex, reflex or supporting. This results in nine cases; we take $q_1 = p_1$ and $q_2 = p_2$ if $(p_1, p_2) =$ (supporting, supporting). In the case $(p_1, p_2) =$ (inflex, inflex), we define l_1 to be the line going through p_1 and its right neighbor in U_1, l_2 the line going through p_2 and its left neighbor in U_2, and then use l_1 and l_2 to decide which portion of U_1 and/or U_2 to eliminate. For the remaining cases, deciding which portion of U_1 and/or U_2 to discard can be easily done. In summary, during the computation of q_1 and q_2, we can always eliminate a portion of one or both hulls. Recall that U_1 and U_2 are stored in the secondary trees of the roots of T_1 and T_2. Therefore, starting from the points of the roots of the two secondary trees, we can find q_1 and q_2 in $O(\log n)$ time. We then *split* U_1 and U_2 by q_1 and q_2, keeping U_{12} and U_{21} in the left and right children of ρ, and *splice* U_{11} and U_{22} into ρ, all in $O(\log n)$ time. Thus bridging can be done in $O(\log n)$ time. The de-bridging is performed easily: split U into U_{11} and U_{22} by the i-th point q_1, splice U_{11} into the left child of ρ and U_{22} into the right child, and remove root ρ. Hence de-bridging also can be done in $O(\log n)$ time.

To insert a point, we first search in the tree T_U along a root-to-leaf path and insert it as a new leaf, according to its x-coordinate. For each node visited, we perform a de-bridging operation, so that its children contain the complete U-hulls of their leaves. We then perform a sequence of bridgings along this leaf-to-root path to restore the property that each point is stored only once in the internal nodes of T_U. Finally, rotations may be needed to rebalance the tree, each corresponding to a constant number of splits and splices of the secondary structures involved. Since there are $O(\log n)$ bridgings, de-bridgings, splits, and splices, each taking $O(\log n)$ time, the total time for insertion is $O(\log^2 n)$. A deletion can be performed similarly, with the same time bound $O(\log^2 n)$.

7.3 Overview of Dynamic Convex Hull Techniques

Let (S_1, S_2) be a partition of S such that all the points in S_1 are to the left of those in S_2. Given the convex hulls of S_1 and S_2, the convex hull of S can be computed in $O(\log n)$ time in two dimensions [124] (the *bridging* of the previous section), and in $O(n)$ time in three dimensions [132]. This property allows one to apply the dynamization technique for order-decomposable problems. Hence, planar convex hulls can be dynamically maintained using $O(n)$ space, $O(\log^2 n)$ update time, $O(\log n)$ time for a find-query, and $O(k)$ for a report-query [121]. For three-dimensional convex hulls, the space requirement is $O(n \log\log n)$, the update time is $O(n)$, and the query time is $O(\log n)$ for a find-query, and $O(k)$ for a report-query. A tree-based data structure for planar convex hulls that implements these ideas is presented in [124], as described in the previous section.

Karp, Motwani, and Raghavan [87] consider the problem of answering a sequence of on-line queries on a static data set. Let t be the number of queries, which is not known in advance. The conventional preprocessing method, which at once constructs a data structure for the entire data set, takes $P(n) + tQ(n)$ time to answer t queries, where $P(n)$ and $Q(n)$ are the preprocessing and query time, respectively. For example, t convex hull membership queries are performed in time

$O((n+t)\log n)$. This method may be wasteful when t is small. Namely, if $t = o(\log n)$, it is more convenient to avoid preprocessing and execute each query with an $O(n)$-time algorithm. In the *deferred data structuring* method of [87] the processing is query-driven, and builds up the data structure piece by piece, only when required for answering a query. It is shown that t convex hull membership queries can be performed in optimal time $O((n+t)\log\min(n,t))$, using as a subroutine the Kirkpatrick-Seidel top-down convex hull algorithm [90]. Several *dynamic deferred data structuring* methods for some membership problems are given by Smid [151] and Ching, Mehlhorn and Smid [43].

An off-line dynamic convex hull problem is studied by Hershberger and Suri [78]. They show that a sequence of n operations, each an insertion, deletion, or query, can be processed in time $O(n\log n)$ using $O(n)$ space, provided all the operations are known in advance. Alternatively, a sequence of n insert and delete operations can be preprocessed in $O(n\log n)$ time and space such that queries in history (i.e., with respect to the point-set at a given time in the past) can be answered in $O(\log n)$ time. The method of [78] can be extended to the problems of maintaining off-line the maxima of a point-set, the intersection of a set of half-spaces, and the kernel of a simple polygon.

The technique of van Kreveld and Overmars [93] to make data structures concatenable provides a data structure for reporting the convex hull within a query rectangle in time $O(\sqrt{n\log n}\log n + k)$; insertions and deletions take $O(\log^2 n)$ time and the space complexity is $O(n)$.

Seidel [147] gives a randomized incremental semi-dynamic algorithm for constructing convex hulls, which takes expected $O(n\log n)$ time for $d = 2,3$, and $O(n^{\lfloor d/2\rfloor})$ time for $d > 3$, to execute a random sequence of insertions. Mulmuley [116] presents a fully dynamic randomized algorithm for maintaining convex hulls, which takes expected $O(\log n)$ time for $d = 2,3$, and $O(n^{\lfloor d/2\rfloor - 1})$ time for $d > 3$, to execute an update of a random (N,δ)-sequence (see Section 4.4).

8 Proximity

In this section, we consider problems involving distances between points, which are typically measured using an L_r metric ($1 \le r \le \infty$), where the distance between d-dimensional points $p = (p_1,\cdots,p_d)$ and $q = (q_1,\cdots,q_d)$ is given by

$$\left(\sum_{i=1}^{d} |p_i - q_i|^r\right)^{\frac{1}{r}},$$

if $r < \infty$, and, for $r = \infty$, by

$$\max_{1\le i\le d} |p_i - q_i|.$$

The usual Euclidean metric is L_2.

The *closest* (resp. *farthest*) *points problem* consists of finding a pair of points of a set S (called *sites*) that are at minimum (resp. maximum) distance under a given metric. The *closest point* *searching* problem consists of finding the site closest to a given query point. The *all-nearest-neighbors* problem asks for the closest site of each site (excluding itself).

In a static environment, these problems in the planar case can be efficiently solved by constructing a *Voronoi diagram* for S, which is defined as follows: for each site $p \in S$, consider the region of the plane formed by the points closer to p than to any other site; we call this region the *Voronoi polygon* of p, and the partition of the plane induced by the Voronoi polygons of all sites the *Voronoi diagram* of S. The Voronoi diagram of S uses $O(n)$ space and can be constructed in $O(n \log n)$ time. If we represent the Voronoi diagram with a point-location data structure, closest point queries take $O(\log n)$ time (see [133]). Also, given the Voronoi diagram of S, the all-nearest-neighbors problem can be solved in $O(n)$ time [148].

Aggarwal *et al.* [5] show that in a planar Voronoi diagram, points can be inserted and deleted in $O(n)$ time; this also leads to an update time of $O(n)$ for maintaining the minimal distance of a point set, using $O(n)$ space.

Guibas, Knuth, and Sharir [74] present an algorithm that incrementally constructs the Voronoi diagram of n sites in the plane by adding the sites one at a time in random order (and by updating the diagram each time) in total expected time $O(n \log n)$ using $O(n)$ space. Note that the worst-case time for the incremental construction is $\Theta(n^2)$.

Mulmuley [114] gives an randomized incremental algorithm, which can be made on-line without changing the running time, for constructing Voronoi diagrams of order 1 to k in R^d. The expected running time over a random sequence of n insertions is $O(k^2 n + n \log n)$ for $d = 2$ and $O(k^{\lceil d/2 \rceil + 1} n^{\lceil d/2 \rceil})$ for $d > 2$. For $d = 2$, Aurenhammer and Schwarzkopf [9] give an on-line algorithm for maintaining k-th order Voronoi diagrams with $O(k^2 n \log n + kn \log^3 n)$ expected running time and optimal space $O(k(n-k))$. Mulmuley [116] gives a dynamic algorithm for maintaining Voronoi diagrams of order 1 to k in R^d. The expected time of an update in a random (N, δ)-sequence (see Section 4.4) is $O(k^2 + \log n)$ for $d = 2$ and $O(k^{\lceil d/2 \rceil + 2} n^{\lceil d/2 \rceil - 1})$ for $d > 2$. The algorithm is also extended to obtain a dynamic algorithm for point location in the k-th order Voronoi diagram, which is equal to answering the k-nearest neighbour queries. The query time is $\tilde{O}(k \log^2 n)$ for $d = 2$ and $\tilde{O}(k \log^3 n)$ for $d = 3$. The expected time of a random update is $O(k^2 + \log n)$ for $d = 2$ and $O(k^4 n \log^2 n)$ for $d = 3$. A randomized algorithm for dynamically maintaining a data structure to support point location queries in planar Voronoi diagrams, based on dynamic sampling, is also given in [117].

Mulmuley [115] shows how to build a dynamic data structure that can be used to answer k-nearest neighbour queries in R^d for any d, with $O(k \log n)$ expected query time and $O(m^{\lceil d/2 \rceil + \epsilon})$ expected time for executing a random (N, δ)-sequence, for any $\epsilon > 0$, where m is the size of N.

Bentley [17,18] considers several proximity searching problems in a semi-dynamic point set. In [17], kd-trees are used to support the nearest neighbour query and the *fixed-radius nearest neighbour* query, which asks for reporting all points within a fixed radius of a given query point. The point set is semi-dynamic in the sense that deletions and undeletions (of the previously deleted points) are supported but insertions of new points are not allowed. It is shown that several operations that require $O(\log n)$ expected time in general kd-trees may be performed in $O(1)$ expected time in semi-dynamic trees. Also, a sampling technique reduces the time to build a tree from $O(kn \log n)$ to $O(kn + n \log n)$. [18] shows how proximity searches in semi-dynamic point sets are able to yield efficient practical algorithms for problems such as minimum spanning trees, approximate matchings, and a wide variety of traveling salesman heuristics.

Closest point searching with updates restricted to insertions only is a decomposable searching problem. Hence, the logarithmic method presented in Section 3 can be applied to yield $O(n)$ space and $O(\log^2 n)$ query and update times. Earlier results in this direction are presented in [20,72,99]. By [150], based on the method of Overmars' thesis [121], the time for insertions and queries can be improved to $O(\log^2 n / \log \log n)$ as follows. Two Voronoi diagrams can be merged in linear time. Therefore, we can apply the logarithmic method with $P(n) = O(n)$. If we replace the logarithmic method by a method that writes the size of S in the number system with base $\log n$, then both the query time and the insertion time are $O(\log^2 n / \log \log n)$.

Supowit [158] shows how to dynamize heuristic algorithms for closest-point and farthest-point problems in the presence of deletions.

Maintaining the minimum and maximum distance between points under a semi-on-line sequence of updates can be handled using the general technique of Dobkin and Suri [51] (see Section 3). They show that in the plane, such updates can be performed in $O(\log^2 n)$ amortized time. Using [152], this update time can be made worst-case.

Smid [155], Schwarz and Smid [145], and Schwarz, Smid and Snoeyink [146] give several techniques for maintaining the closest pair of a point set under insertions only or under a semi-on-line sequence of updates. In [155], an algorithm is given that uses linear space and supports insertions only, in $O(\log^{d-1} n)$ amortized time. It only uses algebraic functions and thus is optimal for the planar case. Another method of [155] is a variation of Dobkin and Suri [51], and gives a linear size data structure that supports each semi-on-line update in $O(\log^2 n)$ time for arbitrary fixed dimension. If only insertions take place, the update time can be improved to $O(\log^2 n / \log \log n)$. The data structure of [145] has linear size and supports insertion-only updates in $O(\log n \log \log n)$ amortized time for arbitrary fixed dimension; it is also shown that the technique can be extended to support semi-on-line updates in $O(\log^2 n)$ worst-case time. [146] gives an algorithm for any fixed dimension that uses only algebraic functions and thus is optimal. The data structure uses linear space and maintains the closest pair in $O(\log n)$ amortaized time per insertion. It also solves the problem of computing on-line the closest pair that existed over the history of a fully dynamic point set in $O(\log n)$ amortized time per insertion or deletion, using linear space.

Smid [153,156] gives two fully dynamic techniques for maintaining the minimal L_r distance of a point set in d-dimensional space. The data structure of [153] uses $O(n)$ space and supports updates in $O(n^{2/3} \log n)$ time, by giving a method to compute the $O(n^{2/3})$ smallest distances defined by a set of n points in $O(n \log n)$ time. By [49,141], which show how to compute the n smallest distances in $O(n \log n)$ time, the update time is improved to $O(n^{1/2} \log n)$, for arbitrary dimension. In [156], the update time is reduced to $O(\log^d n \log \log n)$ amortized, while the space is slightly increased to $(n \log^d n)$.

We briefly describe the main idea of [156] for two dimensions. Let $d(A, B)$ be the minimal distance between a point in set A and a point in set B. Hence $d(S,S)$ denotes the minimal distance between *distinct* points in S. We partition S into two equal-sized subsets S_1 and S_2 by a vertical line t, and define variables δ_1, δ_2, and δ_{12} as follows:

- $\delta_1 = d(S_1, S_1)$,
- $\delta_2 = d(S_2, S_2)$, and
- δ_{12} is such that $\delta_{12} \geq d(S,S)$ and, if $d(S_1,S_2) = d(S,S)$, then $\delta_{12} = d(S_1, S_2)$.

We have that $\delta = d(S,S) = \min(\delta_1, \delta_2, \delta_{12})$. The data structure for S consists of two recursively defined structures for S_1 and for S_2, maintaining δ_1 and δ_2, respectively, and a data structure for the pair (S_1, S_2), maintaining δ_{12}. We now partition S by a horizontal line l into two equal-sized sets, then t and l define four quadrants. The two pairs of left-right adjacent quadrants are maintained in two recursively defined structures as for (S_1, S_2). The two pairs of opposite quadrants are maintained as follows. Let (A, B) be one such pair with B in the first quadrant and A in the third. Assume that $|B| \geq |A|$. Let $C_+ = [0,s] \times [0,s]$ be the smallest square containing at least 10 points of B, and let $C_- = [-t, 0] \times [-t, 0]$ be the smallest square containing at least 10 points of A. Assume w.l.o.g. that $s \leq t$. Let B' (resp. A') be the set of 10 points of B (resp. A) with the smallest L_∞-distance to the origin. Since $d(A - A', B) \geq t \geq s$, $d(A, B - B') \geq s$, and $d(S,S) \leq d(B', B') < s$, to maintain the variable δ_{AB} for (A,B), it suffices to compare points in A' and B'. It can be shown that, if we take $\delta = d(A', B')$, then the third condition above is satisfied. The complete data structure resembles the range tree, and updates can be performed by rotations. Moreover, dynamic fractional cascading can be applied.

The *width* of a point set is the minimum width of a strip (formed by two parallel lines) that encloses all the points. Agarwal and Sharir [4] study the following problem of *off-line dynamic maintenance of width*: given a real $W > 0$ and a sequence of n insert/delete operations on the point set S, determine whether there is any i such that after the i-th operation, the width of S is at most W. They make use of the dynamic convex hull technique of Overmars and van Leeuwen [124] (as discussed in Section 7.2), so that the answer can be computed in $O(n \log^3 n)$ time, using $O(n)$ space.

Janardan [86] presents two approximation algorithms to respectively compute the width and the Euclidean distance between the furthest points (the *diameter*) of a planar point set S, under insertions and deletions. Let W and D be the width and the diameter of S, respectivley, and $\alpha \geq 1$, $\beta \geq 1$ be two integer-valued parameters. The first algorithm uses $O(\alpha n)$ space, supports updates in $O(\alpha \log^2 n)$ time and reports in $O(\alpha \log^2 n)$ time a pair of supporting lines of S with distance $\hat{W}$, where $\hat{W}/W \leq \sqrt{1 + \tan^2 \frac{\pi}{4\alpha}}$. The second algorithm uses space $O(\beta n)$, supports updates in $O(\beta \log n)$ time and reports in $O(\beta)$ time a pair of points of S at distance $\hat{D}$, where $\hat{D}/D \geq \sin(\frac{\beta}{\beta+1}\frac{\pi}{2})$.

9 Other Problems

9.1 Maximal Points

Let S be a set of n points in the plane. A *maximal point* p of S is one such that no other point of S has both its x- and y-coordinates greater than those of p. The *m-contour* of S is the "staircase" chain delimiting the points of the plane that are dominated by at least one maximal point of S. One can test if a point is maximal by performing a range query, using, e.g., the data structure of Willard and Lueker [169] or the priority search tree [108]. Frederickson and Rodger [61] consider the problem of maintaining the m-contour of S. Their data structure consists of a balanced tree that stores the points sorted by x-coordinate, plus several additional pointers. It uses $O(n)$ space, supports insertions in $O(\log n)$ time and deletions in $O(\log^2 n)$ time. Testing whether a point is inside, on, or outsided the m-contour takes $O(\log n)$ time. The m-contour can be reported in time $O(m)$, where m is the number of points on the contour. Janardan [85] gives a technique that supports all operations of [61] within the same space and time bounds. In addition, it can report the m-contour of the points of S that lie to the left of a vertical query line, in $O(\log n + k)$ time, where k is the size of the answer. This method uses a variation of the dynamic convex hull data structure of Overmars and van Leeuwen [124] (as discussed in Section 7.2) combined with the priority search tree [108], plus some additional pointers.

9.2 Union of Intervals

Given a set of intervals on a line, which is dynamically updated by insertions and deletions, we consider the problems of answering the following queries:

- report the union of the intervals as a sequence of k disjoint intervals sorted from left to right;
- test if a query point is inside the union;
- test if a query interval is contained in the union;
- find all the ℓ intervals in the union intersected by a query interval;

The technique of Overmars [120] uses $O(n)$ space, supports updates in time $O(\log^2 n)$, and answers these queries in time $O(k)$, $O(\log k)$, $O(\log k)$, and $O(\log k + \ell)$, respectively. Cheng and Janardan [39] improve the update time to $O(\log n)$, while the time for queries increases by replacing the $\log k$ terms with $\log n$. Their data structure also supports the maintenance of the measure of the union, which can be reported in $O(1)$ time.

9.3 Hidden Line or Surface Elimination

Bern [24] considers a hidden-line removal problem in which the scene consists of n rectangles with sides parallel to the coordinate axes, and the viewpoint at $z = \infty$. In [24] two methods are given for this problem. The first is for static scenes, based on segment tree and heap structures, with running time $O(n \log n + k \log n)$, using space $O(n \log n)$, where k is the number of line segments in the output. The second one is dynamic, further supporting insertions and deletions of rectangles. $BB[\alpha]$-trees, priority search trees and dynamic fractional cascading are used, so that it achieves $O(\log^2 n \log\log n + k \log^2 n)$ amortized time per insertion or deletion, using space $O(n \log^2 n + q \log n)$. Here k is the number of visible line segments that change, and n and q are respectively the number of rectangles and the number of visible line segments at the time of the update. Also, the visible rectangle in the display containing a given query point can be found in time $O(\log n \log\log n)$.

Preparata, Vitter and Yvinec [129,139] give two hidden-line elimination techniques for displaying a scene of three-dimensional isothetic parallelepipeds (3D-rectangles). The method of [139] considers the case where the scene is viewed from infinity along one of the coordinate axes (axial view). The algorithm is *scene-sensitive* and runs in time $O(n \log^2 n + k \log n)$, where n is the number of the 3D-rectangles and k is the number of edges of the display. In [129], the scene is formed from a perspective view. The primary data structure is a simple alternative to dynamic fractional cascading for use with augmented segment trees and range trees when the universe is fixed beforehand. The algorithm is scene-sensitive and runs in time $O((n+k) \log n \log\log n)$, where n is the number of the 3D-rectangles and k is the number of edges of the display.

Preparata and Vitter [138] consider the problem of hidden-line elimination in terrains. Their main data structure implicitly maintains a lower convex hull. At any time, the current convex hull can be printed in time linear in its size. It is simpler than the structure of Overmars and van Leeuwen [124], since it requires fewer balanced trees (one rather than a linear number) and still supports the same applications. This structure is used to perform hidden-line elimination in time $O((n+k) \log^2 n)$ for terrains, where k is the number of edges of the display.

De Berg [22] presents an output-sensitive algorithm to maintain the view of a set of c-oriented polyhedra under insertions and deletions, where a set of polyhedra is *c-oriented* if the number of different orientations of its edges is bounded by some constant c. The algorithm allows the polyhedra to intersect as well as to cyclically overlap in the scene. The method is based on new dynamic data structures for ray-shooting and range searching *for c-oriented objects*, namely, a ray-shooting query is given by a c-oriented query ray and a range searching query is given by a c-oriented polygon (of constant size). The method maintains the visibility map at the cost of $O((k+1) \log^3 n)$ time per insertion or deletion, and use $O((n+K) \log^2 n)$ space, where k is the number of changes in the view, and K is the size of the visibility map.

9.4 Shortest Path

The *shortest path* problem can be stated as follows: Given a simple polygon P, we want to determine the Euclidean shortest path (or its length) between two given query points q_1 and q_2 inside P, avioding the edges of P. For the static case, an optimal algorithm is given by Guibas and Hershberger [73] that supports shortest path and shortest path length queries in $O(\log n + k)$ and $O(\log n)$ time, respectively, with $O(n)$ space and $O(n)$ preprocessing time, where k is the number of edges in the reported answer.

Chiang, Preparata and Tamassia [41] use a unified approach (see Section 6.2) to solve the dynamic shortest path problem in a connected planar subdivision M, where each region of M is a simple polygon; the updates consist of insertions and deletions of vertices and edges. Clearly, if the two query points q_1 and q_2 are not in the same region, then they can not reach each other; this can be checked by performing *point location* queries on q_1 and q_2. The *hull structure* is used to dynamically maintain the *hourglasses* introduced in [73] from which the shortest path can be computed. Moreover, efficient updates are ensured by the *normalization structure*. The approach uses space $O(n \log n)$, supports shortest path and shortest path length queries in time $O(\log^3 n + k)$ and $O(\log^3 n)$, respectively, with update time $O(\log^3 n)$ worst-case for edge updates and amortized for vertex updates, where k is the number of edges in the reported answer.

9.5 Constructive Solid Geometry

Complex geometric solids are represented in Constructive Solid Geometry (CSG) as a hierarchical combination of primitive objects (e.g., spheres and tetrahedra) using set operations such as union, intersection, and difference. The resulting compound object is described by a CSG-tree whose leaves are the primitive objects and whose internal nodes are set operators. CSG representations are a fundamental tool in computer graphics. Several efficient algorithms are given in [69] for CSG classification problems, such as determining whether a query point is inside a compound object.

Cohen and Tamassia [45] study a dynamic point inclusion problem in which a collection of compound objects is modified by changing their primitive constituents and by update operations, such as link and cut, on their CSG-trees. Queries consist of reporting whether a given compound object contains a fixed point p. By applying their *dynamic expression trees* to the technique of [69], they show that the dynamic point inclusion problem on a collection of compound objects defined by CSG-trees of total size n can be solved using an $O(n)$-space data structure that supports query and update operations in $O(\log n)$ time.

9.6 Approximation

Fortune [59] considers geometric algorithms implemented using approximate arithmetic. An algorithm is *robust* if it always produces an output that is correct for some perturbation of its input; it is *stable* if the perturbation is small. An assertion of stability should be accompanied by a measure of the relative perturbation bound. Perturbation can be measured as a function of the problem size n and the relative accuracy ε of the approximate arithmetic. In [59] a technique to maintain the triangulation of a planar point set is presented, in which the triangulation is stored as a combinatorial planar graph embedding. The following operations are supported: point location, adding and deleting points, and changing edges. It achieves the stability assertion that, at any time, there is a relative perturbation of the points of size at most $O(n^2\varepsilon)$ that makes the embedding actually planar.

Franciosa and Talamo [60] study the problem of approximating the convex hull of a set of points whose coordinates are real numbers, and thus may be of infinite length. They define an ε-approximated convex hull, whose vertices are within distance ε from the corresponding ones of the convex hull, and vice versa. A sequence of approximations such that ε halves at each step is constructed, where the k-th approximation is computed from the first k bits of each coordinate. In a RAM with logarithmic costs, the algorithm computes the s-th approximation in $O(ns(\log n + s))$ time and $O(n)$ space, building the k-th approximation from the $(k-1)$st one in $O(n(\log n + k))$ time.

9.7 Graph Drawing

A *drawing* Γ of a graph G maps each vertex of G to a distinct point of the plane and each edge (u, v) of G to a simple Jordan curve with endpoints u and v. We say that Γ is a *polyline* drawing if each edge is a polygonal chain; Γ is a *straight-line* drawing if each edge is a straight-line segment; Γ is an *orthogonal* drawing if each edge is a chain of alternating horizontal and vertical segments. A *grid* drawing is such that the vertices and bends along the edges have integer coordinates. *Planar* drawings, where edges do not intersect, are especially important because they improve the readability of the drawing. An *upward* drawing of an acyclic digraph has all the edges flowing in the same direction, e.g., from bottom to top. A *visibility representation* maps vertices into horizontal segments and edges into vertical segments that intersect only the two corresponding vertex segments. Graph drawing algorithms are surveyed in [52]. In the following we denote with n and m the number of vertices and edges of a given graph.

Cohen et al. [44] describe a framework for dynamic graph drawing algorithms. At a first glance, it appears that updating a drawing may require $\Omega(n+m)$ time in the worst case, since one may have to change the coordinates of all vertices and edges. Their approach is to consider graph drawing problems in a "query/update" setting. Namely, an *implicit* representation of the drawing of a graph G is maintained such that the following operations can be efficiently performed:

- *Drawing queries* that return the drawing of a subgraph S of G consistent with the overall drawing of G. They aim at an output-sensitive time complexity for this operation, i.e., a polynomial in $\log n$ and k, where k is the size of S. Ideally, the time complexity should be $O(k + \log n)$. A special case of this query ($S = \{v\}$) returns the coordinates of a single vertex v.

- *Point location queries* in the subdivision of the plane induced by the drawing of G. Such queries are defined when the drawing of G is planar.

- *Window queries* that return the portion of the drawing inside a query rectangle.

- *Local update* operations, e.g., insert and delete vertices and edges, and update the (implicit) representation of the drawing accordingly.

- *Global update* operations, such as combining two graphs into one, or replacing an edge of a graph by another graph.

There are two types of quality measures in dynamic graph drawing: the "aesthetic" properties of the drawing being maintained, and the space-time complexity of queries and updates. There is an inherent tradeoff between the two. For example, it is very easy to maintain the drawing by a graph in which the vertices are randomly placed on the plane and the edges are drawn as straight-line segments. However, the aesthetic quality of the drawings produced by this simple strategy is typically not satisfactory. On the other hand, if we want to guarantee optimal drawings with respect to some aesthetic criteria, e.g., planarity, symmetry, etc., the update/query operations may require high time complexity.

Work on dynamic graph drawing is as follows. Moen [113] considers trees and presents a technique that restructures the drawing of a tree in time proportional to its height, and hence linear in the worst case. Cohen et al. [44] present several techniques for dynamic drawing of trees and planar graphs. The dynamic technique for upward drawings of rooted trees uses $O(n)$ space and supports updates and point location queries in $O(\log n)$ time; drawing queries take time $O(k+\log n)$ for a subtree, and $O(k \log n)$ for an arbitrary subgraph; and window queries take time $O(k \log n)$. The dynamic algorithm for upward straight-line drawings of series-parallel digraphs uses $O(n)$ space and supports updates in $O(\log n)$ time; drawing queries take time $O(k + \log n)$ for a series-parallel subgraph, and $O(k \log n)$ for an arbitrary subgraph; point location queries take $O(\log n)$ time; and window queries take $O(k \log^2 n)$ time. Finally, dynamic data structures for polyline drawings of planar st-graphs and general planar graphs use $O(n)$ space, and support updates in $O(\log n)$ time and drawing queries in $O(k \log n)$ time.

9.8 Lower Bounds

General lower bound techniques for dynamic algorithms are discussed in [23,64]; some relevant lower bounds are also given in [32,62,63,173]. [164] gives an upper bound that precisely matches the lower bounds of [63] and [173]; the difference between these two is that they obtain the same quantitative lower bounds under different models of computation. [88] gives an upper bound that nearly matches the lower bound of [62]. Fredman and Willard [65,174] explain how *fusion trees* refute many conjectures about lower bounds. For instance, [174] shows that one can speed up the priority trees of [108] by a factor of $\log \log n$.

10 Open Problems

Despite the increasing interest in the area of incremental computation, efficient dynamic algorithms are not known for many geometric problems. Further research is needed both in the design of dynamic data structures and in the development of lower-bound arguments for incremental computation. The two most important open problems in the area appear to be the following:

Convex Hull: find a fully dynamic data structure for maintaining the convex hull of a set of points in the plane that uses $O(n)$ space and supports queries and updates in $O(\log n)$ time.

Point Location: find a fully dynamic data structure for planar point location that uses $O(n)$ space and supports queries and updates in $O(\log n)$ time.

References

[1] G.M. Adel'son-Vel'skii and E.M. Landis, "An Information Organization Algorithm," *Doklady Akad. Nauk SSSR* 146 (1962), 263–266.

[2] P. Agarwal and M. Sharir, "Applications of a New Space Partitioning Technique," *Lecture Notes in Computer Science* 519 (1991), 379–391.

[3] P.K. Agarwal, "Ray Shooting and Other Applications of Spanning Trees with Low Stabbing Number," *Proc. ACM Symp. on Computational Geometry* (1989), 315–325.

[4] P.K. Agarwal and M. Sharir, "Planar Geometric Location Problems and Maintaining the Width of a Planar Set," *Computational Geometry: Theory and Applications* 1 (2) (1991), 65–78.

[5] A. Aggarwal, L. Guibas, J. Saxe, and P.W. Shor, "A Linear-time Algorithm for Computing the Voronoi Diagram of a Convex Polygon," *Discrete and Computational Geometry* 4 (1989), 591–604.

[6] A. Andersson, "Improving Partial Rebuilding by Using Simple Balance Criteria," *Proc. WADS' 89,* LNCS 382 (1989), 393–402.

[7] A. Andersson, "Efficient Search Trees," Dept. of Computer Science, Lund Univ., Sweden, Ph.D. dissertation, 1990.

[8] C. Aragon and R.G. Seidel, "Randomized Search Trees," *Proc. IEEE Symp. on Foundations of Computer Science* (1989), 540–545.

[9] F. Aurenhammer and O. Schwarzkopf, "A Simple Online Randomized Incremental Algorithm for Computing Higher Order Voronoi Diagrams," *Proc. ACM Symp. on Computational Geometry* (1991), 142–151.

[10] H. Baumgarten, H. Jung, and K. Mehlhorn, "Dynamic Point Location in General Subdivisions," *Proc. of ACM-SIAM Symp. on Discrete Algorithms* (1992), 250–258.

[11] S.W. Bent, "Dynamic Weighted Data Structures," Stanford Univ., Ph.D. Dissertation, Report STAN-CS-82-916, 1982.

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

[13] J.L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Comm. ACM* 18 (1975), 509–517.

[14] J.L. Bentley, "Algorithms for Klee's Rectangle Problems," Dept. of Computer Science, Carnegie-Mellon Univ., unpublished notes, 1977 .

[15] J.L. Bentley, "Decomposable Searching Problems," *Information Processing Letters* 8 (1979), 244–251.

[16] J.L. Bentley, "Multidimensional Divide-and-Conquer," *Communication ACM* 23 (1980), 214–228.

[17] J.L. Bentley, "K-d Trees for Semidynamic Point Sets," *Proc. ACM Symp. on Computational Geometry* (1990).

[18] J.L. Bentley, "Fast Algorithms for Geometric Traveling Salesman Problems," AT & T Bell Lab., Computing Science Technical Report No. 151, 1990 .

[19] J.L. Bentley and H.A. Mauer, "Efficient Worst-case Data Structures for Range Searching," *Acta Informatica* 13 (1980), 155–168.

[20] J.L. Bentley and J. Saxe, "Decomposable Searching Problems I: Static to Dynamic Transformations," *J. Algorithms* 1 (1980), 301–358.

[21] J.L. Bentley and M.I. Shamos, "A Problem in Multivariate Statistics: Algorithms, Data Structure, and Applications," *Proc. 15th Allerton Conference on Communication, Control, and Computing* (1977), 193–201.

[22] M. de Berg, "Dynamic Output-Sensitive Hidden Surface Removal for c-Oriented Polyhedra," Dept. of Computer Science, Utrecht Univ., Report RUU-CS-91-6, 1991.

[23] A.M. Berman, M.C. Paull, and B.G. Ryder, "Proving Relative Lower Bounds for Incremental Algorithms," *Acta Informatica* 27 (1990), 665–683.

[24] M. Bern, "Hidden Surface Removal for Rectangles," *J. Computer and System Sciences* (1990), 49–69.

[25] N. Blum and K. Mehlhorn, "On the Average Number of Rebalancing Operations in Weight-Balanced Trees," *Theoretical Computer Science* 11 (1980), 303–320.

[26] P. van Emde Boas, "Preserving Order in a Forest in less than Logarithmic Time," *Proc. 16th Symp. on Foundations of Computer Science* (1975), 75–84.

[27] P. van Emde Boas, "Preserving Order in a Forest in less than Logarithmic Time and Linear Space," *Information Processing Letters* 6 (1977), 80–82.

[28] J.D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec, "Applications of Random Sampling to On-line Algorithms in Computational Geometry," *Discrete and Computational Geometry* (to appear).

[29] B. Chazelle, "On the Convex Layers of a Planar Set," *IEEE Trans. Inf. Theory* IT-31 (4) (1985), 509–517.

[30] B. Chazelle, "A Functional Approach to Data Structures and its Use in Multidimensional Searching," *SIAM J. Computing* 17 (3) (1988), 427–462.

[31] B. Chazelle, "Triangulating a Simple Polygon in Linear Time," *Proc. of IEEE Symp. on Foundation of Computer Science* (1990), 220–230.

[32] B. Chazelle, "Lower Bounds for Orthogonal Range Search II. The Arithmetic Model," *Journal of ACM* 37 (1990), 439–463.

[33] B. Chazelle and L.J. Guibas, "Fractional Cascading: II. Applications," *Algorithmica* 1 (1986), 163–191.

[34] B. Chazelle and L.J. Guibas, "Fractional Cascading: I. A Data Structuring Technique," *Algorithmica* 1 (1986), 133–162.

[35] B. Chazelle, M. Sharir, and E. Welzl, "Quasi-optimal Upper Bounds for Simplex Range Searching and New Zone Theorems," *Proc. ACM Symp. on Computational Geometry* (1990), 23–33.

[36] B. Chazelle and E. Welzl, "Quasi-optimal range searching in space with finite VC-dimension," *Discrete and Computational Geometry* 4 (1989), 467–489.

[37] S.W. Cheng and R. Janardan, "Efficient Dynamic Algorithms for Some Geometric Intersection Problems," *Information Processing Letters* 36 (1990), 251–258.

[38] S.W. Cheng and R. Janardan, "New Results on Dynamic Planar Point Location," *Proc. 31st IEEE Symp. on Foundations of Computer Science* (1990), 96–105.

[39] S.W. Cheng and R. Janardan, "Efficient Maintenance of the Union of Intervals on a Line, with Applications," *J. Algorithms* 12 (1991), 57–74.

[40] S.W. Cheng and R. Janardan, "Algorithms for Ray-shooting and Intersection Searching," manuscript, 1991. See also: "Space-efficient Ray-shooting and Intersection Searching: Algorithms, Dynamization, and Applications," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1991), 7-16.

[41] Y.-J. Chiang, F.P. Preparata, and R. Tamassia, "A Unified Approach to Dynamic Point Location, Ray Shooting and Shortest Paths in Planar Maps," Dept. of Computer Science, Brown Univ., Technical Report No. CS-92-07, 1992.

[42] Y.-J. Chiang and R. Tamassia, "Dynamization of the Trapezoid Method for Planar Point Location," *Proc. ACM Symp. on Computational Geometry* (1991), 61–70.

[43] Y.T. Ching, K. Mehlhorn, and M.H.M. Smid, "Dynamic Deferred Data Structuring," *Information Processing Letters* 35 (1990), 37–40.

[44] R.F. Cohen, G. Di Battista, R. Tamassia, I.G. Tollis, and P. Bertolazzi, "A framework for dynamic graph drawing," *Proc. ACM Symp. on Computational Geometry* (1992), 261–270.

[45] R.F. Cohen and R. Tamassia, "Dynamic Expression Trees and their Applications," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1991), 52–61.

[46] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, McGraw-Hill, MIT Press, 1990.

[47] W. Cunto, G. Lau, and P. Flajolet, "Analysis of KDT-Trees: KD-Trees Improved by Local Reorganizations," *Proc. WADS' 89,* LNCS 382 (1989), 24–38.

[48] O. Devillers, M. Teillaud, and M. Yvinec, "Dynamic Location in an Arrangement of Line Segments in the Plane," *Algorithms Review* 2 (3) (1992), 89–103.

[49] M.T. Dickerson and R.S. Drysdale, "Enumerating k distances for n points in the plane," *Proc. ACM Symp. on Computational Geometry* (1991), 234–238.

[50] P.F. Dietz and R. Raman, "Persistence, Amortization and Randomization," *Proc. 2nd ACM-SIAM Symp. on Discrete Algorithms* (1991), 78–88.

[51] D. Dobkin and S. Suri, "Dynamically Computing the Maxima of Decomposable Functions, with Applications," *Proc. 30th IEEE Symp. on Foundations of Computer Science* (1989), 488–493.

[52] P. Eades and R. Tamassia, "Algorithms for Automatic Graph Drawing: An Annotated Bibliography," Dept. of Computer Science, Brown Univ., Technical Report CS-89-09, 1989.

[53] H. Edelsbrunner, "Optimizing the Dynamization of Decomposable Searching Problems," *IIG, Technische Univ. Graz, Austria, Rep 35* (1979).

[54] H. Edelsbrunner, "A New Approach to Rectangle Intersections, Part II," *Int. J. Computer Mathematics* 13 (1983), 221–229.

[55] H. Edelsbrunner, "A New Approach to Rectangle Intersections, Part I," *Int. J. Computer Mathematics* 13 (1983), 209–219.

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

[57] H. Edelsbrunner, L.J. Guibas, and J. Stolfi, "Optimal Point Location in a Monotone Subdivision," *SIAM J. Computing* 15 (1986), 317–340.

[58] H. Edelsbrunner, G. Haring, and D. Hilbert, "Rectangular Point Location in d Dimensions with Applications," *The Computer Journal* 29 (1986), 76–82.

[59] S. Fortune, "Stable Maintenance of Point-set Triangulations in Two Dimensions," *Proc. 30th Symp. on Foundations of Computer Science* (1989), 494–499.

[60] P.G. Franciosa and M. Talamo, "An On-Line Convex Hull Algorithm on Reals," *ALCOM Workshop on Data Structure, Graph Algorithms, and Computational Geometry, Berlin (Germany), October, 1990* .

[61] G. Frederickson and S. Rodger, "A New Approach to the Dynamic Maintenance of Maximal Points in a Plane," *Discrete and Computational Geometry* 5 (1990), 365–374.

[62] M.L. Fredman, "The Inherent Complexity of Dynamic Data Structures Which Accomodate Range Queries," *Proc. IEEE Symp. on Foundations of Computer Science* (1980), 191–199.

[63] M.L. Fredman, "A Lower Bound on the Complexity of Orthogonal Range Queries," *Journal of ACM* 28 (1981), 696–706.

[64] M.L. Fredman and M.E. Saks, "The Cell Probe Complexity of Dynamic Data Structures," *Proc. 21st ACM Symp. on Theory of Computing* (1989), 345–354.

[65] M.L. Fredman and D.E. Willard, "Blasting through the Information Theoretic Barrier with Fusion Trees," *Proc. ACM Symp. on Theory of Computing* (1990), 1–7.

[66] O. Fries, "Suchen in dynamischen planaren Unterteilungen," Univ. des Saarlandes, Ph.D. Thesis, 1990.

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

[68] H.N. Gabow and R.E. Tarjan, "A Linear Time Algorithm for a Special Case of Disjoint Set Union," *J. Computer and System Sciences* 30 (1985).

[69] M.T. Goodrich, "Applying Parallel Processing Techniques to Classification Problems in Constructive Solid Geometry," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1990), 118–128.

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

[71] G. Gowda and D.G. Kirkpatrick, "Exploiting Linear Merging and Extra Storage in the Maintenance of Fully Dynamic Geometric Data Structures," *Proc. 18th Annual Allerton Conf. on Communication, Control, and Computing* (1980), 1–10.

[72] I.G. Gowda, D.G. Kirkpatrick, D.T. Lee, and A. Naamad, "Dynamic Voronoi Diagrams," *IEEE Trans. Information Theory* IT-29 (5) (1983), 724–731.

[73] L.J. Guibas and J. Hershberger, "Optimal Shortest Path Queries in a Simple Polygon," *Proc. 3rd ACM Symposium on Computational Geometry* (1987), 50–63.

[74] L.J. Guibas, D.E. Knuth, and M. Sharir, "Randomized Incremental Construction of Delauney and Voronoi Diagrams," *Automata, Languages and Programming (Proc. 17th ICALP), Lecture Notes in Computer Science* (1990), 414–431.

[75] L.J. Guibas and R. Sedgewick, "A Dichromatic Framework for Balanced Trees," *Proc. 19th IEEE Symp. on Foundations of Computer Science* (1978), 8–21.

[76] L.J. Guibas and J. Stolfi, "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams ," *ACM Trans. on Graphics* 4(2) (1985), 75–123.

[77] J. Hershberger and S. Suri, "Applications of a Semi-Dynamic Convex Hull Algorithm," *Proc. SWAT'90, Lecture Notes in Computer Science* 447 (1990), 380–392.

[78] J. Hershberger and S. Suri, "Offline Maintenance of Planar Configurations," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1991), 32–41.

[79] S. Huddleston and K. Mehlhorn, "Robust Balancing in B-Trees," *Lecture Notes in Computer Science* 104 (1981), 234–244.

[80] S. Huddleston and K. Mehlhorn, "A New Data Structure for Representing Sorted Lists," *Acta Informatica* 17 (1982), 157–184.

[81] C. Icking, R. Klein, and T. Ottmann, "Priority Search Trees in Secondary Memory," *Graph-Theoretic Concepts in Computer Science (Proc. Int. Workshop WG '87, Kloster Banz, June 1987)* (1988), 84–93.

[82] H. Imai and T. Asano, "Efficient Algorithms for Geometric Graph Search Problems," *SIAM J. Computing* 15 (1986), 478–494.

[83] H. Imai and T. Asano, "Dynamic Orthogonal Segment Intersection Search," *J. Algorithms* 8 (1987), 1–18.

[84] S.S. Iyengar, R.L. Kashyap, V.K. Vaishnavi, and N.S.V. Rao, "Multidimensional Data Structures: Review and Outlook," *Advances in Computers* 27 (1988), 69–94.

[85] R. Janardan, "On the Dynamic Maintenance of Maximal Points in the Plane," manuscript, 1991.

[86] R. Janardan, "On Maintaining the Width and Diameter of a Planar Point-set Online," manuscript, 1991.

[87] R. Karp, R. Motwani, and P. Raghavan, "Deferred Data Structuring," *SIAM J. Computing* 17(5) (1988), 883–902.

[88] M.D. Katz and D.J. Volper, "Data Structures for Retrieval on Square Grids," *SIAM J. Computing* 15 (1986), 919–931.

[89] D.G. Kirkpatrick, "Optimal Search in Planar Subdivisions," *SIAM J. Computing* 12 (1983), 28–35.

[90] D.G. Kirkpatrick and R. Seidel, "The Ultimate Planar Convex Hull Algorithm?," *SIAM J. Computing* 15 (1986), 287–299.

[91] R. Klein, O. Nurmi, T. Ottman, and D. Wood, "A Dynamic Fixed Windowing Problem," *Algorithmica* 4 (1989), 535–550.

[92] M. van Kreveld and M.H. Overmars, "Divided K-D Trees," *Tech. Report RUU-CS-88-28, Dept. of Computer Science, Univ. of Utrecht, Netherlands* (1988).

[93] M. van Kreveld and M.H. Overmars, "Concatenable Structures for Decomposable Problems," *Tech. Report RUU-CS-89-16, Dept. of Computer Science, Univ. of Utrecht, Netherlands* (1989).

[94] M.J. van Kreveld and M.H. Overmars, "Union-copy Structures and Dynamic Segment Trees," Dept. of Computer Science, Utrecht Univ., Report RUU-CS-91-5, 1991. See also: "Concatenable Segment Trees," *Proc. STACS 89, Lecture Notes in Computer Science*,349 (1989), 493-504.

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

[96] D.T. Lee and C.K. Wong, "Worst-case Analysis of Region and Partial Region Searches in Multi-dimensional Binary Search Trees and Balance Quad Trees," *Acta Informatica* 9 (1977), 23–29.

[97] J. van Leeuwen and H.A. Maurer, "Dynamic Systems of Static Data Structures," Inst. f. Informationsverarbeitung, TU Graz, Tech. Report No. 42, 1980.

[98] J. van Leeuwen and M.H. Overmars, "The Art of Dynamizing," *Proc. Symp. on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science* 118 (1981), 121–131.

[99] J. van Leeuwen and D. Wood, "Dynamization of Decomposable Searching Problems," *Information Processing Letters* 10 (1980), 51–56.

[100] W. Lipski, "Finding a Manhattan Path and Related Problems," *Networks* 13 (1983), 399–409.

[101] W. Lipski, "An $O(n \log n)$ Manhattan Path Algorithm," *Information Processing Letters* 19 (1984), 99–102.

[102] G. Lueker and D.E. Willard, "A Data Structure for Dynamic Range Queries," *Information Processing Letters* 15 (1982), 209–213.

[103] G.S. Lueker, "A Data Structure for Orthogonal Range Queries," *Proc. 19th IEEE Symp. on Foundations of Computer Science* (1978), 28–34.

[104] D. Maier and S.C. Salveter, "Hysterical B-Trees ," *Information Processing Letters* 12 (1981), 199–202.

[105] J. Matoušek, "More on cutting arrangements and spanning trees with low crossing number," Dept. of Computer Science, Charles Univ. Czechoslovakia, Tech. Report B-90-2, 1990.

[106] J. Matoušek, "Efficient Partition Trees," *Proc. ACM Symp. on Computational Geometry* (1991), 1–9.

[107] H.A. Maurer and T. Ottmann, "Dynamic Solutions of Decomposable Searching Problems," *Discrete Structures and Algorithms* (1979), 17–24.

[108] E.M. McCreight, "Priority Search Trees," *SIAM J. Computing* 14 (1985), 257–276.

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

[110] K. Mehlhorn, *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag, New York, 1984.

[111] K. Mehlhorn and S. Näher, "Dynamic fractional cascading," *Algorithmica* 5 (1990), 215–241.

[112] K. Mehlhorn and M.H. Overmars, "Optimal Dynamization of Decomposable Searching Problems," *Information Processing Letters* 12 (1981), 93–98.

[113] S. Moen, "Drawing Dynamic Trees," *IEEE Software* 7 (1990), 21–28.

[114] K. Mulmuley, "On Levels in Arrangements and Voronoi Diagrams," *Discrete and Computational Geometry* 6 (4) (1991).

[115] K. Mulmuley, "Randomized Multidimensional Search Trees: Further Results in Dynamic Sampling," *Proc. IEEE Symp. on Foundations of Computer Science* (1991), 216–227.

[116] K. Mulmuley, "Randomized Multidimensional Search Trees: Lazy Balancing and Dynamic Shuffling," *Proc. IEEE Symp. on Foundations of Computer Science* (1991), 180–196.

[117] K. Mulmuley, "Randomized Multidimensional Search Trees: Dynamic Sampling," *Proc. ACM Symp. on Computational Geometry* (1991), 121–131.

[118] K. Mulmuley and S. Sen, "Dynamic Point Location in Arrangements of Hyperplanes," *Proc. ACM Symp. on Computational Geometry* (1991), 132–141. Revised version: technical report, Univ. of Chicago, July 1991.

[119] J. Nievergelt and E.M. Reingold, "Binary Search Trees of Bounded Balance," *SIAM J. Computing* 2 (1973), 33–43.

[120] M.H. Overmars, "Dynamization of Order Decomposable Set Problems," *J. Algorithms* 2 (1981), 245–260.

[121] M.H. Overmars, "The Design of Dynamic Data Structures," *Lecture Notes in Computer Science* 156 (1983).

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

[123] M.H. Overmars and J. van Leeuwen, "Two General Method for Dynamizing Decomposable Searching Problems," *Computing* 26 (1981), 155–166.

[124] M.H. Overmars and J. van Leeuwen, "Maintenance of Configurations in the Plane," *J. Computer and System Sciences* 23 (1981), 166–204.

[125] M.H. Overmars and J. van Leeuwen, "Dynamization of Decomposable Searching Problems Yielding Good Worst-case Bounds," *Lecture Notes in Computer Science* 104 (1981), 224–233.

[126] M.H. Overmars and J. van Leeuwen, "Worst-case Optimal Insertion and Deletion Methods for Decomposable Searching Problems," *Information Processing Letters* 12 (1981), 168–173.

[127] M.H. Overmars and J. van Leeuwen, "Some Principles for Dynamizing Decomposable Searching Problems," *Information Processing Letters* 12 (1981), 49–54.

[128] M.H. Overmars, M. Smid, M.T. de Berg, and M.J. van Kreveld, "Maintaining Range Trees in Secondary Memory, Part I: Partitions," *Acta Informatica* 27 (1990), 423–452.

[129] F. P. Preparata, J. S. Vitter, and M. Yvinec, "Output-Sensitive Generation of the Perspective View of Isothetic Parallelepipeds," Brown University, Department of Computer Science, Technical Report 89-50, 1989.

[130] F.P. Preparata, "An Optimal Real Time Algorithm for Planar Convex Hulls," *Comm. ACM* 22 (1979), 402–405.

[131] F.P. Preparata, "A New Approach to Planar Point Location," *SIAM J. Computing* 10 (1981), 473–483.

[132] F.P. Preparata and S.J. Hong, "Convex Hulls of Finite Sets of Points in Two and Three Dimensions," *Comm. ACM* 2(20) (1977), 87–93.

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

[134] F.P. Preparata and R. Tamassia, "Efficient Spatial Point Location," *Algorithms and Data Structures (Proc. WADS '89)* (1989), 3–11.

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

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

[137] F.P. Preparata and R. Tamassia, "Efficient Point Location in a Convex Spatial Cell Complex," *SIAM J. Computing* 21 (1992), 267–280.

[138] F.P. Preparata and J. S. Vitter, "A Simplified Technique for Hidden-Line Elimination in Terrains," *Proc. 1992 Symposium on Theoretical Aspects of Computer Science (STACS), Lecture Notes in Computer Science* (1992).

[139] F.P. Preparata, J.S. Vitter, and M. Yvinec, "Computation of the Axial View of a Set of Isothetic Parallelepipeds," *ACM Transactions on Graphics* 3 (1990), 278–300.

[140] N.S.V. Rao, V.K. Vaishnavi, and S.S. Iyengar, "On the Dynamization of Data Structures," *BIT* 28 (1988), 37–53.

[141] J.S. Salowe, "Shallow Interdistance Selection and Interdistance Enumeration," *Lecture Notes in Computer Science* 519 (1991), 117–128.

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

[143] H. Schipper and M.H. Overmars, "Dynamic Partition Trees," *Proc. SWAT'90, Lecture Notes in Computer Science* 447 (1990), 404–417.

[144] H.W. Scholten and M.H. Overmars, "General Methods for Adding Range Restrictions to Decomposable Searching Problems," *J. Symbolic Computation* 7 (1989), 1–10.

[145] C. Schwarz and M. Smid, "An $O(n \log n \log\log n)$ Algorithm for the On-line Closest Pair Problem," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1992), 280–285.

[146] C. Schwarz, M. Smid, and J. Snoeyink, "An Optimal Algorithm for the On-line Closest-Pair Problem," *Proc. ACM Symp. on Computational Geometry* (1992), to appear.

[147] R. Seidel, "Linear Programming and Convex Hulls Made Easy," *Proc. ACM Symp. on Computational Geometry* (1990), 211–215.

[148] M.I. Shamos, "Computational Geometry," *Doctoral Disertation, Yale Univ.* (1978).

[149] D.D. Sleator and R.E. Tarjan, "A Data Structure for Dynamic Trees," *J. Computer and System Sciences* 24 (1983), 362–381.

[150] M. Smid, private communication.

[151] M. Smid, "Dynamic Data Structures on Multiple Storage Media," University of Amsterdam, Ph.D. Thesis, 1989.

[152] M. Smid, "A Worst-case Algorithm for Semi-online Updates on Decomposable Problems," Fachbereich Informatik, Universität des Saarlandes, Report A 03/90, 1990. See also: "Algorithms for Semi-online Updates on Decomposable Problems", *Proc. Canadian Conf. on Computational Geometry* (1990), 347-350.

[153] M. Smid, "Maintaining the Minimal Distance of a Point Set in Less Than Linear Time," *Algorithms Review* 2 (1) (1991), 33–44.

[154] M. Smid, "Range Trees with Slack Parameter," *Algorithms Review* 2 (2) (1991), 77–87.

[155] M. Smid, "Dynamic Rectangular Point Location, with an Application to the Closest Pair Problem," Max-Planck-Institut für Informatik, Saarbrücken, Report MPI-I-91-101, 1991. See also: Proc. 2nd International Symp. on Algorithms, 1991..

[156] M. Smid, "Maintaining the Minimal Distance of a Point Set in Polylogarithmic Time (revised version)," Max-Planck-Institut für Informatik, Saarbrücken, Report MPI-I-91-103, 1991. See also: *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1991), 1–6.

[157] M. Smid and M.H. Overmars, "Maintaining Range Trees in Secondary Memory, Part II: Lower Bounds," *Acta Informatica* 27 (1990), 453–480.

[158] K. Supowit, "New Techniques for some Dynamic Closest-Point and Farthest-Point Problems," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1990), 84–90.

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

[160] R.E. Tarjan, "Data Structures and Network Algorithms," *CBMS-NSF Regional Conference Series in Applied Mathematics* 44 (1983).

[161] R.E. Tarjan, "Updating a Balanced Search Tree in $O(1)$ Rotations," *Information Processing Letters* 16 (1983).

[162] R.E. Tarjan, "Amortized Computational Complexity," *SIAM J. Algebraic Discrete Methods* 6 (1985), 306–318.

[163] V.K. Vaishnavi, "Computing Point Enclosures," *IEEE Transactions on Computers* C-31 (1) (1982), 22–29.

[164] V.K. Vaishnavi, "Multidimensional Balanced Binary Trees," *IEEE Transactions on Computers* 38 (7) (1989), 968–985.

[165] V.K. Vaishnavi, "On k-dimensional Balanced Binary Trees," Dept. of Computer Information Systems, Georgia State Univ., Technical Report CIS-90-001, 1990.

[166] V.K. Vaishnavi, "Multidimensional (a, b)-Trees: An Efficient Dynamic Multidimensional File Structure," Dept. of Computer Information Systems, Georgia State Univ., Technical Report CIS-90-003, 1990.

[167] V.K. Vaishnavi and D. Wood, "Rectilinear Line Segment Intersection, Layered Segment Trees, and Dynamization," *J. Algorithms* 3 (1982), 160–176.

[168] D. Willard, "New Data Structures for Orthogonal Range Queries," *SIAM J. Computing* (1985), 232–253.

[169] D. Willard and G. Lueker, "Adding Range Restriction Capability to Dynamic Data Structures," *J. ACM* 32 (1985), 597–617.

[170] D.E. Willard, "The Super-B-Tree Algorithm," *Tech. Report TR-03-79, Aiken Computer Lab., Harvard University* (1979).

[171] D.E. Willard, "On the Application of Sheared Retrieval to Orthogonal Range Queries," *Proc. ACM Symp. on Computational Geometry* (1986), 80–89.

[172] D.E. Willard, "Multidimensional Search Trees That Provide New Types of Memeory Reductions," *J. ACM* 34 (4) (1987), 846–858.

[173] D.E. Willard, "Lower Bounds for the Addition-Subtraction Operations in Orthogonal Range Queries," *Information and Computation* 82 (1989), 45–64.

[174] D.E. Willard, "Applications of the Fusion Tree Method to Computational Geometry and Searching," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1992), 286–295.

[175] F.F. Yao, "Computational Geometry," in *Handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity*, Elsevier/MIT Press, 1990, 343–390.