Dynamic Algorithms in Computational Geometry

Yi-Jen Chiang
Roberto Tamassia

Department of Computer Science
Brown University
Providence, Rhode Island 02912

Technical Report No. CS-91-24
March 26, 1991

Dynamic Algorithms in Computational Geometry*

(Preliminary 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 overviews the state-of-the-art for the specialist.

(March 26, 1991)

*Research supported in part by the National Science Foundation under grant CCR-9007851, by the U.S. Army Research Office under grant DAAL03-91-G-0035, by the Office of Naval Research and the Defense Advanced Research Projects Agency under contract N00014-83-K-0146 and ARPA order 6320, amendment 1, and by Cadre Technologies, Inc.

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.

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 overviews the state-of-the-art for the specialist.

Section 2 reviews fundamental data structures such as balanced search trees. In Section 3 we overview general techniques for dynamization. Sections 4–8 focus on range searching, intersections, point location, convex hull, and proximity. Problems that do not fall in the above 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. Some fundamental dynamic geometric techniques are described in detail in the books by Mehlhorn [69] and by Preparata and Shamos [84]. Iyengar et al. [52] survey dynamic data structures for multidimensional searching. Overmar's thesis [73] 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,

2 Fundamental Data Strucutres

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* [101] will be used in the analysis of data structure where the worst-case time complexity of an operation is larger than the average time complexity over a sequence of operations. Namely, 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 [101].

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. Example of height balanced trees are AVL trees [1] and red-black trees [44]. Weight-balanced trees were first presented in [72] and include $BB[\alpha]$ trees [14]. All of these trees use $O(n)$ space and support each operation of the above repertory in time $O(\log n)$.

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 [47,48,65,99].

If updating the secondary structures takes more than logarithmic time, we can use instead 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 $O(1)$ [68]. 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 the 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 [24,68,100]. New techniques for rebuilding balanced tree structures are presented by Andersson [5,6].

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 [31,51,62,103,104]. Chazelle and Guibas [17,18] 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 say that G has *locally bounded degree* if 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 $x \in R(e)$.

Let k denote the size of the query tree T. The fractional cascading technique of Chazelle and Guibas [17,18] is static, supporting queries only, and assumes locally bounded degree of G. It achieves $O(\log n + k)$ query time. Mehlhorn and Naher [70] dynamize fractional cascading to further support insertions and deletions, so that the query time is $O(\log n + k \log\log n)$ and the update time is $O(\log\log n)$ amortized. If only insertions or deletions are allowed, the $O(\log\log n)$ factor reduces to $O(1)$. They also assume locally bounded degree. Dietz and Raman [26] improve the update time of fractional cascading to $O(\log\log n)$ worst-case, eliminating the amortization.

3 General Dynamization Methods

Several general techniques have been developed for constructing dynamic data structures from static ones. Overmar's thesis [73] describes in detail the results in this area until 1983.

Throughout this section, $P(n)$, $Q(n)$, $U(n)$, and $M(n)$ indicate preprocessing time, query time, update 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 [73]. 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 $O(M(n))$, and amortized update time $O(U(n) + P(n)/n)$. 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. Bentley and Saxe [10] have first introduced the following two dynamization methods 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 $O(\log n + P(n)/n + U(n/f(n))$, and space $O(f(n)M(n/f(n))$. If $Q(n) = O(\log n)$, $U(n) = O(n)$, and $P(n) = O(n \log n)$, 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. Combined with global rebuilding, it gives a dynamic data structures with query time $O(Q(n) \log n)$, update time $O((\log n)P(n)/n)$, and space $O(M(n))$.

Several variations of dynamization methods for decomposable problems are reported in [28,41, 60,66,71,73,76,77,78,79,80,89,94,105].

The dynamization technique of [89] 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''. Note that membership, range search, and range counting problems are deletion-decomposable. The 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.

Kreveld and Overmars [57] show how to obtain concatenable data structures from existing data structures, based on a technique called *ordered equal block method.* This can be applied to all data structures for decomposable search problems.

Dobkin and Suri [27] 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 is known in advance at the moment of its insertion. Their main result can be expressed as follows.

If the function f is decomposable and admits a static data structure that can be constructed in time $P(n)$ and allows to find the maximum of $f(x, y)$ over all $y \in S$ 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.

Hence, the performance of the data structure is suboptimal by a $\log n$ factor. As an immediate application, the diameter, or the closest pair, of a point-set can be maintained in time $O(\log^2 n)$ per update.

Vaishnavi [102] gives a multi-dimensional tree structure called *d-dimensional balanced binary tree.* It stores a set of n d-dimensional points in $O(d \cdot n)$ space and supports membership queries,

insertion, and deletions in $O(\log n + d)$ time, where only $O(d)$ rotations are needed to rebalance the tree after an insertion or deletion.

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.).

4.1 One-Dimensional Range Searching

Balanced search trees support one-dimensional range searching in time $O(\log n + k)$, 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_1, x_2) is performed as follows:

- search for x_1 and x_2 in T, which gives the nodes μ_1 and μ_2 whose associated points or intervals respectively contain x_1 and x_2;
- follow the thread pointers from μ_1 to μ_2 and report the points of P associated with all the internal nodes encountered.

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. 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 and preprocessing time are $O(n \log n)$.

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). 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

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 $O(\log n)$, an 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 [64,104,106]. In the dynamic case we have $O(\log n \log\log n + k)$ query time and $O(\log n \log\log n)$ update time [70].

This technique readily extends to arbitrary dimensions 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).

4.3 Priority Search Tree

The *priority search tree* of McCreight [67] 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 and right subtrees are priority search trees for the points of $P - \{p(\rho)\}$ to the left and right of the vertical line $x = x(\rho)$.

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 ($x^* > x(\rho)$), 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. Also, each internal node 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 [8,9] 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 apply the static-to-dynamic transformation of Bentley and Saxe [10], 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 [63] illustrate an ad hoc construction using the Nievergelt-Reingold [72] 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 [105] 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 [63] 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.

Iyengar *et al.* [52] survey data structures for range searching and other multi-dimensional searching problems.

Willard [107] 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 [107] modifies the *first-generation k-fold tree* of Bentley and Shamos [11] 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 [7]. 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 [7,52,84]. A local reorganization technique improving the performance of kd-trees is presented in [25].

Kreveld and Overmars [56,57] give two methods for range searching. In [57], 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 [56], 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.

Icking, Klein and Ottman [49] 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.

5 Intersections

5.1 Segment Tree

The *segment tree* data structure was introduced by Bentley [8]. Let X be a set of N points on a line, and S a set of n segments with endpoints on X. A *segment-tree* $T = T(X, S)$ supports the following operations:

- report all the segments of S that contain query point x;
- count the number of segments of S that contain 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. 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. 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. 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 at each node μ the set of proper segments with its size, so that they take $O(\log N)$ time using $O(N)$ space. Inserting or deleting a segment takes $O(\log N)$ time.

We can remove the restriction on the fixed set X of endpoints 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 $O(\log^2 n)$ amortized time.

5.2 Concatenable Segment Tree

A *concatenable* version of the segment tree is given by van Kreveld and Overmars [58]. In addition to stabbing queries and standard updates (insertion and deletion of segments), it supports split and concatenate operations. Namely, a concatenate operations 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 $O(\log n)$ time, and deletions take $O(\log^2 n)$ time. If counting stabbing queries are performed, then all operations take $O(\log n)$ time and the space requirement is reduced to $O(n)$. This method is based on a new *union-copy* data structures for sets.

5.3 Interval Tree

Here we present the *interval tree* data structure due to Edelsbrunner [29], which is called *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 with $S(\mu)$, 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 enpoint 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 Kreveld and Overmars [57] 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 $O(\log n + k)$ [103].

Lipski [61,62] 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 [15,16] 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 $O(n)$ and known in advance.

Dynamic fractional cascading eliminates the above 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 [70] for the general problem.

Imai and Asano [50,51] give two semi-dynamic techniques where updates are restricted to insertions only or deletions only, respectively. The linear-time set-union and set-splitting algorithms of Gabow and Tarjan [38] 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 [67] 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 [67]. 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.

Orthogonal segment intersection queries can be extended to higher dimensions. Let S be a set of d-dimensional rectangles (each a set of the type $[a_1, b_1]\times[a_2, b_2]\times\cdots\times[a_d, b_d]$). The rectangles of S intersected by a given (d-dimensional) query rectangle can be computed using a *d-fold rectangle tree* [29,30], 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*, where 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 to report the segment of S intersected by an arbitrary query segment. *Ray-shooting queries* ask to determine the first segment hit by an arbitrarily-directed ray r emanating from a query point q. Cheng and Jarnadan [21] improve and dynamize the static techniques of [2] for segment intersection and ray-shooting. Their method uses space $O(n\log^3 n)$

and supports updates in $O(\sqrt{n}\log^{\omega} n)$ time. Segment intersection and ray-shooting queries take $O(\sqrt{n}\log^2 n + k\log^2 n)$ and $O(\sqrt{n}\log^2 n)$ time, respectively.

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. If the query point lies on a vertex or edge of P, then that vertex or edge is reported. 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.

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 [31,54,90]. All of them have $O(\log n)$ query time, $O(n\log n)$ preprocessing time, and $O(n)$ space requirement.

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 [45]: an *expansion* splits a vertex v into two new vertices connected by edge, where 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 [74]. 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 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 corresponds to standard segment-tree operations on T, and thus takes $O(\log^2 n)$ 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: 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.

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 corresponds to performing $O(1)$ concatenable-queue operations on the boundary-trees, which takes time $O(\log n)$.

6.2 Overview of Dynamic Point Location Techniques

The dynamic point-location technique of Overmars [74] 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 [36,37] present a data structure for general subdivisions with $O(n)$ space, $O(\log^2 n)$ query time, and $O(\log^4 n)$ amortized update time, using an approach based on the static chain-method [59]. The update operations supported are inserting and deleting edges and isolated vertices. If only insertions are performed, the update time is reduced to $O(\log^2 n)$ [69 (pp. 135–143)]

Preparata and Tamassia give two dynamic techniques for monotone and convex subdivisions, respectively. [86,87].

The data structure for monotone subdivisions [86] is based on the chain-method [59]. 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 [86]. As shown in [85,88], the data structure can be extended to support also expansion and contraction operations in time $O(\log^2 n)$, with applications to three-dimensional point location.

The data structure for convex subdivision [87] is based on the trapezoid method [82]. 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 $O(\log n \log N)$ [87].

Tamassia presents a technique for point location in triangulations (i.e., subdivisions with triangular regions), which allows a tradeoff between query and update time, and can be used in conjunction with any of the known static point location data structures [98]. It is based on the dynamization methods for decomposable search problems [10,73]. 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. Namely, it is shown that for any smooth nondecreasing integer function $b(n)$ with $2 \leq b(n) \leq \sqrt{n}$, there exists a dynamic point location data structure with $O(n)$ space, $O((\log^2 n)/\log b(n))$ query time, and $O((\log^2 n)b(n)/\log b(n))$ update time. 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 [19] present two methods for dynamic planar point-location. Their first method achieves $O(\log^2 n)$ query time, $O(\log n)$ time for inserting/deleting a vertex, and $O(k \log n)$ time for inserting/deleting a chain of k edges. Their second method achieves $O(\log n \log\log n + k)$ time for inserting/deleting monotone chains, at the expense of increasing vertex insertion/deletion time to $O(\log n \log\log n)$ and increasing the query time to $O(\log^2 n \log\log n)$. Both of their methods use $O(n)$ space and are based on a search strategy derived from the priority search tree [67](see Section 4.3). They dynamize this approach with the $BB(\alpha)$ tree data structure [68], using the approach of Willard and Lueker [105] to spread local updates over future operations, and the method of Overmars [73] 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 [40] show how to dynamically maintain a monotone subdivision so as to achieve $O(n)$ space, $O(\log^2 n)$ query time, $O(\log n)$ time for vertex insertion/deletion, and $O(\log n + k)$ time for the insertion/deletion of a monotone chain 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 trees data structure of Sleator and Tarjan [93] are used to dynamize the method, which is relatively easy to implement. This approach also allows 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 where only insertions are performed. In this case the centroid decomposition of the dual tree is explicitly maintained in a $BB(\alpha)$ tree [68], and a simple version of fractional cascading [17] 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 [22] present a fully dynamic data structure for point location in a monotone subdivision, based on the trapezoid method [82]. 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 [87] in two aspects: it supports the monotone

subdivision instead of convex subdivision, 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 [75] 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 the optimal query time $O(\log n)$, with 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.

7 Convex Hull

Computing the convex hull H of a set of points S is not a searching problem, so that 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 naturally arise:

- find if a given point of S is on the convex hull H;
- 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.

7.1 Semi-Dynamic Maintenance of Planar Convex Hulls

In this section we present a variation of the technique of Preparata [81] 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 none of these segments exist, then q is outside the convex hull H. Otherwise, q is in H if and only if it lies vertically between such segments, which can be checked in $O(1)$ time.

Because of symmetry considerations, we only describe 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, which are tangent to U. If p is above U, then p has two tangents to U, called left and right tangent. 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 respect to p as follows:

- vertex v_i is *left* (resp. *right*) *supporting* if v_{i-1} and v_{i+1} are on the right (resp. left) 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.

7.2 Fully Dynamic Maintenace of Planar Convex Hulls

Now, we describe the fully dynamic convex hull technique of Overmars and van Leeuwen [75]. While in the insert-only scheme of the previous section points found to be internal to the current convex hull can be thrown away, 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 $\overline{q_1 q_2}$ is the supporting line to both U_1 and U_2. The U-hull U then consists of U_{11}, $\overline{q_1 q_2}$, 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. Recall that U_1 and U_2 are stored in the secondary trees of the roots of T_1 and T_2. Starting from the points of the roots of the two secondary trees, classify the line formed by them as inflex, reflex or supporting and move down left or right appropriately, 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, perform the 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 $O(\log n)$ number of bridings, de-bridgings, splits, and splices, each taking $O(\log n)$ time, thus the total time for insertion is $O(\log^2 n)$. The 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 [75] (the *bridging* of the previous section), and in $O(n)$ time in three dimensions [83]. This property allows 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 [73]. 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 with the same performance bounds is presented in [75], as described in the previous section.

Karp, Motwani, and Raghavan [53] 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 constructs at once 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 [53] 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 [55].

An off-line dynamic convex hull problem is studied by Hershberger and Suri [46]. It is shown 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 the 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 [46] 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 Kreveld and Overmars [57] 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 $O(n)$.

8 Proximity

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

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

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

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

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 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. By representing the Voronoi diagram with a point-location data structure, closest point queries take $O(\log n)$ time (see [84]). Also, given the Voronoi diagram of S, the all-nearest-neighbors problem can be solved in $O(n)$ time [92].

Aggarwal *et al.* [4] 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. [43] 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 $\Theta(n^2)$.

Closest point searching 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 [42,60,91].

Supowit [97] shows how to dynamize heuristic algorithms for closet-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 [27] (see Section 3). They show that in the plane, such updates can be performed in $O(\log^2 n)$ amortized time.

Smid [95,96] gives two techniques for maintaining the minimal L_p-distance of a point set in d-dimensional space. The data structure of [95] uses $O(n)$ space and supports updates in $O(n^{2/3} \log n)$ time. In [96], the update time is reduced to $O(\log^{d+2} n)$ amortized, while the space is slightly increased to $(n \log^d n)$.

We briefly describe the main idea of [96] 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 points in S. Partition S into two equal-sized subsets S_1 and S_2 by a vertical line t. 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 $\delta = d(S, S)$ is $min(\delta_1, \delta_2, \delta_{12})$. 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}. Partition S_1 and S_2 by a horizontal line l such that the larger of the two sets is divided evenly. Now, t and l define four quadrants. The two pairs of left-right adjacent quadrants are maintained in two recursively defined structures as that for (S_1, S_2). The two pairs of opposite quadrants are maintained as follows. Let (A, B) be one of such pairs 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 $|B|/2$ points of B, and let $C_- = [-s,0] \times [-s,0]$. Let B' and A' be the sets of points lying in the interior of C_+ and of C_-, respectively. The structure for the pair (A,B) consists of two structures for A and B (recursively defined as that for S), and a recursively defined structure for the pair (A',B'). Since $d(A-A',B) > s$ and $d(A,B-B') > s$, to maintain the variable δ_{AB} for (A,B), it suffices to compare points in A' and B'. It can be shown that the variable δ' for the pair (A',B') is equal to δ_{AB}, and that updates can be performed by a partial rebuilding technique.

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 [3] 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 [75], so that the answer can be computed in $O(n \log^3 n)$ time, using $O(n)$ space.

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 such that no other point of S that has both its x- and y-coordinates greater than those of p. The *m-contour* of S is the "staircase" chain delimiting the 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 [105] or the priority search tree [67]. Frederickson and Rodger [34] 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. Also the m-contour can be reported in time $O(m)$.

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 and van Leeuwen [75] uses $O(n)$ space, supports updates in time $O(\log^2 n)$, and answers the above queries in time $O(k)$, $O(\log k)$, $O(\log k)$, and $O(\log k + \ell)$, respectively. Cheng and Janardan [20] 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 Elimination

Bern [13] considers a hidden-line removal problem where the scene consists of n rectangles with sides parallel to the coordinate axes, and the viewpoint at $z = \infty$. In [13] two methods are given for this problem. The first is for static scene, based on segment tree and heap structures, with running time $O(n \log n \log\log n + k \log n)$, 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 $O(n \log^2 n + q \log n)$, where k is the number of visible line segments that change, 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 $O(\log n \log\log n)$.

9.4 Constructive Solid Geometry

Complex geometric solids are represented in Constructive Solid Geometry (CSG) as a hierarchic 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 [39] for CSG classification problems, such as determining whether a query point is inside a compound object.

Cohen and Tamassia [23] study a dynamic point inclusion problem where 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 [39], they show that the above 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.

9.5 Approximation

Fortune [32] consider geometric algorithms implemented using approximation 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 [32] a technique to maintain the triangulation of a planar point set is presented, where 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 [33] 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.

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. General lower bound techniques for dynamic algorithms are discussed in [12,35].

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

[3] P.K. Agarwal and M. Sharir, "Planar Geometric Location Problems and Maintaining the Width of a Planar Set," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1991), 449–458.

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

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

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

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

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

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

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

[11] J.L. Bentley and M.I. Shamos, "A Problem in Multi-variate Statistics: Algorithm, Data Structure and Applications," *Proc. 15th Allerton Conference on Communications, Control, and Computers* (1977), 193–201.

[12] A.M. Berman, M.C. Paull, and B.G. Ryder, "Proving Relative Lower Bounds for Incremental Algorithms," *Acta Informatica* (to appear).

[13] M. Bern, "Hidden Surface Removal for Rectangles," *Proc. 4th ACM Symp. on Computational Geometry* (1988), 183–192.

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

[15] 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.

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

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

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

[19] 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.

[20] S.W. Cheng and R. Janardan, "Efficient Maintenance of the Union Intervals on a Line, with Applications," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1990), 74–83.

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

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

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

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

[25] 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.

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

[27] 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.

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

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

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

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

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

[33] 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* .

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

[35] 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.

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

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

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

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

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

[41] 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.

[42] 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.

[43] 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), 414431.

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

[45] 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).

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

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

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

[49] 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.

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

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

[52] 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.

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

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

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

[56] 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).

[57] 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).

[58] M.J. van Kreveld and M.H. Overmars, "Concatenable Segment Trees," *Proc. STACS 89* (1989), 493–504.

[59] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[77] 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.

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

[79] 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.

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

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

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

[83] 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.

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

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

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

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

[88] F.P. Preparata and R. Tamassia, "Efficient Point Location in a Convex Spatial Cell Complex," *SIAM J. Computing* (1991 (to appear)).

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

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

[91] J. Saxe and J. Bentley, "Transforming Static Data Structures to Dynamic Structures," *Proc. 20th IEEE Symp. on Foundation of Computer Science* (1979), 148–168.

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

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

[94] M. Smid, "A Worst-case Algorithm for Semi-online Updates on Decomposable Problems," Fachbereich Informatik, Universität des Saarlandes, Report A 03/90, 1990.

[95] M. Smid, "Maintaining the Minimal Distance of a Point Set in Less Than Linear Time," Fachbereich Informatik, Universität des Saarlandes, Report A 06/90, 1990.

[96] M. Smid, "Maintaining the Minimal Distance of a Pont Set in Polylogarithmic Time," *Proc. ACM-SIAM Symp. on Discrete Algorithms* (1991), 1–6.

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

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

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

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

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

[102] K.V. Vaishnavi, "Multidimenstional Balanced Binary Trees," *Tech. Report CIS-86-007, Dept. of Computer Information Systems, Georgia State Univ.* (1986).

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

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

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

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

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