Dynamic Algorithms in Computational Geometry*

Xiang-Yang Li†

Abstract. *Research on dynamic algorithms of geometry problems has received increasing attention in the last years, motivated by many important applications in network optimization, VLSI layout, computer graphics, and distributed computing. In this paper, we survey dynamic algorithms for some computational geometry problems such as point location, nearest neighbor, closest point pair, planar width and farthest point pair, etc.*

keywords. dynamic data structure, point location, nearest neighbor, closest point pair, planar width, point maxima.

1 Introduction

Research on dynamic algorithms of geometry problems has received increasing attention in the last years, motivated by many important applications in network optimization, VLSI layout, computer graphics, and distributed computing. In this paper, we survey dynamic algorithms for some computational geometry problems.

Dynamic computation considers updating the solution of a problem when the problem instances is modified. Many applications are incremental in the sense that we only add objects to the data structures, which is much easier then the problem where we not only add objects but also delete objects. Considerable savings could be made if the new solution need not be generated from scratch. There are two kind of dynamic problem: *semi-dynamic*, if the repertory of update operations consist only of insertion or only of deletion of objects; and *fully-dynamic*, if the algorithm supports an intermixed sequences of insertions and deletions. Consider any update sequence $\overline{U}$. Define its signature[28] $\delta = \delta(\overline{U})$ to be the string of $+$ and $-$ symbols that the ith symbol is $+$ (or $-$) if and only if the ith operation in $\overline{U}$ is insertion (or deletion). Let N be the set of objects involved in $\overline{U}$. Then we say that update $\overline{U}$ is an $(\overline{U}, \delta)$-sequence. An update sequence signature is *valid* if and only if for any given time, the number of deletions is no more than that of insertions. An update is *random* $(\overline{U}, \delta)$-sequence if it is chosen from a uniform distribution on all valid $(\overline{U}, \delta)$-sequences. An update is *weakly monotonic* if, after the ith operation, the number of the objects remained is $\Omega(i)$.

We survey the dynamic algorithms and data structures in the area of computational geometry. Our work has twofold purposes: it introduces the area to the nonspecialist and reviews the state-of-the-art for the specialist. Section 2 reviews some fundamental data structures of balanced search trees. In section 3, we review the general techniques used in the dynamization. We review the

*This is the term survey paper for geometry data structures class offered by Jeff Erickson.

†Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA. `xli2@cs.uiuc.edu`, `http://www-sal.cs.uiuc.edu/~ xli2`

recent development of dynamic point location algorithms in section 4. In section 5, we will survey the dynamic data structures to maintain the width of the point set , the data structure to maintain the neatest neighbor query, the data structure to maintain the closest point pair, the farthest point pair,etc.

Previous tutorial and survey work in the area are as follows. F. Yao [39] first had a review of the major topics in computational geometry. Fundamental dynamic geometry data structures and techniques also can be found in the books by Edelsbrunner [16], Mehlhon [27], and by Preparata and Shamos [31]. Overmars' thesis and Chiang's thesis[10] are two excellent references for dynamic computational geometry data structures and techniques. The recent survey given by Agarwal and Erickson[2] provides an excellent opportunity to recapitulate the current status of geometric range searching and to summarize the recent progress in its relatives.

2 Fundamental Data Structures

In this section, we define some terminology and briefly describe some fundamental data structures that are used as ingredients of a variety of algorithms.

2.1 Balanced Trees

The idea of balancing a search tree is due to Adel'son-Vel'skii and Landis, who introduced a class of balanced search tree called *AVL trees* in 1962. Balance is maintained in AVL tress by rotations, but as many as $\Theta(\log n)$ rotations may be required after an insertion to maintain the balance of n-node tree. Hopcroft introduced another class of trees called $2-3$-trees, whose balance is maintained by manipulating the degree of the nodes in the tree. Bayer and McCreight generalized the idea to introduce the $B-trees$, which is widely used in database. Bayer also introduce the basic idea of the red-black trees. But the red/black color convention is introduced by Guibas and Sedgewick. A binary search tree is a red-black tree if it satisfies the following *red-black* properties:

- Every node is either red or black.
- Every leaf(NIL) is black.
- If a node is red, then both its children is black.
- Every simple path from a node to a descendant leaf contains the same number of black nodes, which is called the *black height* of the node.

The height of n-nodes red-black tree is at most $2\log(n+1)$. Then all operations can be done in $O(\log n)$ time. The other kinds of balanced trees are weight-balanced trees, height-balanced trees, the variant of the B-trees.

Sleator and Tarjan[6, 36, 37] introduced the *splay trees*, which is perhaps the most intriguing variations of the balanced binary tree because of the *self-adjusting* property. Splay trees maintain the balance of the tree without explicit balance condition such as the color of red black tree. Instead, splay operations are performed within a tree when an access of a node is made[37]. Through the splay operations, the often visited the nodes will always stay near the root of the tree, which achieves the optimal amortized access cost in many cases.

2.2 Randomized Trees

Aragon and Seidel[3] present randomize 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. It combines the features of other type of search trees including red-black trees, biased search trees, [1] and BB[α] trees. [2] The query and update times are $O(\log n)$, re-balancing is done with $O(1)$ rotations in red-black trees.

3 General Dynamization Technique

For certain computations, it is necessary to use so-called *dynamic* data structures to solve the problem because the given objects set is not static anymore. Some ojects are removed while other new objects are inserted. Even it is possible that the objects of the set are moving, i.e., the coordinates of the points are changed. The data structure should be efficient in inserting and deleting objects, and also be efficient in answering the query. The performance of the dynamic data structures is measured by the amount of the storage required, and the time needed to perform a sequence of insertions, deletions, and queries. For later convenience, we will use $p(n)$, $s(n)$, $t(n)$ to denote the preprocessing time, space requirement, and the time for querying. The modification of a static data structure to a dynamic one is commonly called *dynamization*. Several techniques have been developed to construct the dynamic data structures from the static one. *Balancing* or *local rebuilding* denotes the technique that maintain the logarithmic height of the tree after a sequence of the operations. We can use the *partial rebuilding* method to reconstruct part of the tree to maintain the balance if the tree is out of balance. *Global rebuilding* is the technique to rebuild the whole tree after a sequence of operations. It is often used in conjunction with the data structure which support weak self balancing: it inserts or deletes the node from the data structure but not re-balance it. An example of weak update is *lazy deletion*, which does not remove the deleted object from the data structure, but instead marks it. A data structure support weak updates if there exist some constant b,c such that after $b * n$ updates on a newly built data structure with size n, the query time, the update time, and the space requirement are at most a factor c larger than the corresponding quantities in S.

3.1 Decomposable Query

A search problem on a set of objects S is *decomposable* if for any partition (S_1, S_2) of S, the answer to the query on S can be obtained in constant time from the answers to queries on S_1 and S_2[7, 15]. For example, the search for a point p in a point set S to minimize $||p - q||$, where q is a given query point, is decomposable. For decomposable query questions, we can maintain a collection of static

[1] It is a binary tree whose leaves store weighted items. Let w be the sum of all weights. Then the depth of a leaf with weight w_i is at most $\log(w/w_i) + 2$, and each of the following operations can be done in $O(\log w)$ time: change the weight of an item; insertion/deletion of an item, and split/splice of two biased trees.

[2] A BB[α]-tree, where α is a fixed real number satisfies $1/4 < \alpha \leq 1 - \sqrt{2}/2$, is a binary search tree that has the following properties: a BB[α]-tree with n nodes has height $O(\log n)$;the amortized time of update operation in a sequence of n insertions and deletions, starting from initially empty tree is $O(\log^2 n)$, if the tree is augmented with a secondary structure at each node, and the updating time for the secondary data structure with l leaves takes $O(l \log l)$ time.

data structures that operate on blocks of items of S. The following common used dynamization methods are developed for decomposable search problems.

- *Equal block partition method* partitions the data set S into $b(n)$ subsets of approximately equal size. Each subset is represented by a static data structures. Then the query time is $O(b(n)t(n/b(n)))$ plus the time to get the final result from $b(n)$ query result based on these subsets. The space requirement is $O(b(n)s(n/b(n)))$. And the update time is $O(p(n/b(n)) + p(n)/n)$.

- *Logarithmic partition method* partitions S into $O(\log n)$ sets of size given by the powers of 2 in the binary representation of n. For this representation, if the weak deletion is supported, it gives a dynamic data structure with a query time $O(t(n)\log n)$, insertion time $O(p(n)\log n/n)$, deletion time $O(\log n + d(n) + p(n)/n)$, where $d(n)$ is the weak deletion time.

4 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 an edge of P, then that vertex or edge is reported. The most often used subdivisions of the plane are that every region of the plane is *monotone* [3], or *convex*, or *triangular*, or *trapezoid.* [4] From Euler's formula, the number of vertices, the number of edges, and the number of the regions of a planar subdivision is all with a constant factor. [5] For static point location, we have many optimal algorithms with $O(\log n)$ query time, $O(n \log n)$ preprocessing time and $O(n)$ space requirement. For connected subdivisions, by Chazelle's triangulation result[8], the preprocessing time can be further improved to $O(n)$ time.

Update operations for planar point location consist of inserting or deleting a vertex, or en edge. An *expansion* splits a vertex v into two new vertices connected by an edge; each new vertex inherits a subsequence of the edges previously incident on v. A *contraction* merges two adjacent vertices v_1, v_2 into a new vertex, which inherits the edges v_1v_2.

4.1 General Technique of Point Location

For connected subdivisions of the plane, Overmas[29] shows how to use the segment tree to support the dynamic point location with $O(n \log n)$ space, and $O(\log^2 n)$ query and update time. [6] 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. A query in T identifies the first edge of P hit by a downward ray originating at the query point q, which will take $O(\log^2 n)$ 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. Adding and removing

[3] A polygon chain is monotone if any vertical/horizontal line intersects it in a single point or in a single interval or not at all. A simple polygon is monotone if its boundary consists of two monotone chains.

[4] The trapezoid decomposition induced by S is obtained by drawing from each vertex v of S two horizontal rays that either remain unbounded or terminate when it hit any edges of S. The resulting segments are called *splitters.*

[5] A planar graph with n vertices, has at most $3n - 6$ edges, and at most $2n - 4$ regions.

[6] For inserting a new edge, the edge has to have at least one endpoint already exist in the subdivision.

edges correspond to performing $O(1)$ concatenable-queue operations on the boundary trees, which will take $(\log n)$ time.

Fries and Melhorn also present a data structure for connected subdivision 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.

Cheng and Janardan present two methods for dynamic planar point location in connected subdivision. Their first method has $O(\log^2 n)$ query time, $O(\log n)$ for inserting and deleting a vertex, and $O(k \log n)$ time inserting and deleting a chain of k edges. Their second method achieves $o(\log n \log\log n + k)$ time for inserting/deleting monotone chains, but cost $O(\log n \log\log n)$ time to insert/delete a vertex, and the increase the query time to $O(\log^2 n \log\log n)$.

4.2 Monotone Planar Point Location

The data structure of Preparata and Tamassia[32] for monotone subdivision is based on the chain method. The repertory of update operations includes inserting vertices on edges, inserting monotone chains of edges and their inverses. The space requirement is $O(n)$, and the query time is $O(\log^2 n)$. Vertices can be inserted and deleted in time $O(\log n)$, and the monotone chain with k edges can be inserted or deleted in time $O(\log^2 n + k)$. All bounds are worst case. the data structure can also be extended to support expansion and contraction operations in time $O(\log^2 n)$[].

Goodrich and Tamassia[22] then show how to maintain a monotone subdivision dynamically to achieve $O(n)$ space, $O(\log^2 n)$ query time, $O(\log n)$ insertion/deletion time, and $O(\log n + k)$ time for inserting/deleting a k-edges monotone chain. This improved method is based on the interlaced spanning trees, one for the subdivision itself and one for the graph-theoretic dual, to answer queries. The Link and cut tree of Sleator and Tarjan[36] is used to dynamize the method.

4.3 Trapezoid Planar Point Location

Chiang and Tamassia[12] present a fully dynamic data structure for point location in a monotone subdivision based on the trapezoid method. The operations supported are insertion and deletion of vertices and edges, and horizontal translation of vertices. Point location queries take $O(\log n)$ time, while updates take $O(\log^2 n)$ time, which is amortized for vertex insertion/deletion and the worst for other update operations. The space requirement is $O(n \log n)$. This is the first fully dynamic point location data structure for monotone subdivisions that achieves the optimal $O(\log n)$ query time with the $O(\log n)$ factor space increase.

Chiang, Preparata, and Tamassia[11] then describe a new technique for dynamically maintaining the trapezoidal decomposition of a connected planar map M with n vertices, and apply it to the development of a unified dynamic data structure that supports point location, ray-shooting, and the shortest path [7] queries in M.

The space requirement is $O(n \log n)$. Point location queries take time $O(\log n)$. Ray-shooting and shortest path queries take time $O(\log^3 n)$(plus $O(k)$ if there are k edges in the shortest path to be reported.). Updates, consisting of insertion/deletion of the vertices and edges, take time $(\log^3 n)$, which is amortized for vertices updates.

[7]The *shortest path problem* can be stated as follows: Given S, a simple polygon, or a collection of simple polygon, we want to determine the Euclidean, or other metric, shortest path between two given query point q_1, and q_2, avoiding the edges of S.

4.4 Triangulation Planar Point Location

Tamassia[38] presents a technique for point location in triangulations which allows a tradeoff between query and update time. It is based on the dynamization methods for decomposable problems. The update operations inserting a star(a vertex and three edges) inside a region and swapping the diagonal of a convex quadrilateral formed by adjacent regions. It is shown that for any smooth nondecreasing function $f(n)$ with $2 \leq f(n) \leq \sqrt{n}$, there exists a dynamic point location data structure with $O(n)$ space, $O((\log^2 n)/\log f(n))$ query time, and $O((\log^2 n) f(n)/\log f(n))$ update time.By setting $f(n)$ to different values, we can get different algorithms allowing a tradeoff between space and query time. [8]

4.5 I/O Efficient Planar Point Location

the planar subdivision arising in many applications , such as geographic information system, are too massive to fit in the internal memory and must reside on the external disks. In such instances, the I/O communication is the bottleneck instead of the CPU running time. Most of the work presented before focused on minimizing the CPU running time under the assumption that the planar subdivision fits in the main memory.

Agarwal, Arge, Brodal and Vitter recently give an efficient external-memory dynamic data structure for point location in monotone planar subdivision[1]. Their data structures uses $O(N/B)$ disk blocks to store the monotone subdivision of size N, where B is the disk block size. It supports queries in worst case $O(\log_B^2 N)$ I/O and updates in amortized $O(\log_B^2 N)$ I/O per edge/vertex. Here they assume that an update operation is admissible, which means that the subdivision remains planar and monotone after the operation. Their algorithm first detects whether an update operation is admissible, and carries out it only if it is. The main ingredient of their algorithm is a new variant of B-trees, called *level-balanced B-trees*, which allows insertion, deletion,merging, and split operation in $O((1 + \frac{b}{B} \log_{M/B}(N/B)) \log_b N)$ amortized I/O's, where $2 \leq b \leq B/2$. [9]

5 Nearest Neighbor, Closest Point Pair and Planar Width

In this section, we consider problems involving the distances between points, which are typically measured using a L_r metric. [10] Given a set S of points in R^d, we have to store them in a data structure to solve some specified problem efficiently. The problems that we are interested are the following.

- *Nearest Neighbor*: For a query point q, report the point $p \in S$, such that the distance $d(p,q)$ is minimized among all points in S.
- *Closest Point Pair*: Find the points pair $p \in S$, $q \in S$, such that the distance $d(p,q)$ is minimized.

[8] For example, if setting $f(n) = 2$, then we have $O(\log^2 n)$ query and update time. Also we can set $f(n) = \log n$ or $f(n) = n^\epsilon$ for some $\epsilon > 0$.

[9] We can choose b to optimize the cost. For example, if $b = B/\log B$, then the amortized update time is $O(\log_B^2 N)$ I/O. It uses $O(N/B)$ blocks to store N elements.

[10] The distance between d dimensional points $P = (p_1, p_2, ...p_d)$ and $Q = (q_1, q_2, ...q_3)$ is given by $(\sum_{i=1}^{d} |p_i - q_i|^r)^{\frac{1}{r}}$, if r is finite, else by $max_{i=1}^{d} |p_i - q_i|$.

- *All-Nearest Neighbor*: For each point $p \in S$, find the point $q \in S$, such that the distance $d(p,q)$ is minimized.

- *Planar Width*: find the minimum distance between two parallel supporting lines.

5.1 Nearest Neighbor, All-Nearest Neighbor

The all nearest neighbor problem asks for the closest point(excluding itself) for each point in the set S. In a static environment, this problem can be solved efficiently by constructing the *Voronoi Diagram* for S. [11] The Voronoi diagram of S can be constructed in $O(n \log)$ time using $O(n)$ space. Then we can do closest point queries in $O(\log n)$ time if we represent the Voronoi diagram with a point location data structure such as the bottom triangulation. Also, we can solve the all nearest neighbors problem in $O(n)$ time[34]. A nearest neighbor query for a set of points under Euclidean metric can be formulated as an instance of the ray shooting problem in a convex polyhedron in R^{d+1}. [12]

Mumuley[28] 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 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 algorithms 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 neighbor queries. The query time is about $O(k \log^d n)$ for $d = 2, 3$. The expected time of a random update is about $O(k^2 + \log n)$ for $d = 2$, and $O(k^4 n \log^2 n)$ for $d = 3$. Mumuley also shows how to build a dynamic data structure that can be used to answer k-nearest neighbor 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 updating sequences N.

5.2 Closest Point Pair

Clustering has long been a mainstay of statistical analysis, and clustering-based methods have attention in many fields such as computational biology, scientific simulation, information retrieving. We can use divide and conquer algorithm to compute the closest point pair in $O(n \log n)$ time for static point set, if the distance is measured by Euclidean distance.

Maintaining the closest pairs in a dynamic set of objects is a key subroutine used in several clustering algorithms. There are several obvious approaches to this dynamic closest pair problem. It can be solved by brute force in time $O(n^2)$ per update with space $O(n)$; or by a priority queue of the distances in time $O(n \log n)$ per update with space $O(n^2)$. If each deletion and insertion is likely, then we can maintain the closest pair at amortized time $O(n)$ per update with $O(n)$ space. It

[11] It is defined as following. For each point $p \in S$, the *Voronoi Polygon* of p is the region of the plane formed by the points closer to p than to any other point in S. The partition of the plane induced b the Voronoi polygons of all points in S is the *Voronoi Diagram* of S.

[12] We map each point $p = (p_1, p_2, ..., p_d) \in S$ to a hyperplane $\overline{p}$ in R^{d+1}, which is the graph of the function

$$f_p(x_1, x_2, ...x_d) = 2p_1x_1 + ... + 2p_dx_d - (p_1^2 + ... + p_d^2).$$

Then p is a closest neighbor of a query point $q = (q_1, q_2, ...q_d)$ if and only if

$$f_p(q_1, q_2, ...q_d) = max_{r \in S} f_r(q_1, q_2 ..., q_d).$$

is amazing that most of the practical clustering software use the brute force to maintain the closest pair[17]. Smid[35] gives two fully dynamic techniques for maintaining the minimal L_r distance of a point set in d dimensional spaces. The data structure 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. Dickerson and Drysdale[14] shows how to compute the n smallest distances in $O(n\log n)$ time, and update the data structures in $O(n^{1/2}\log n)$ time, for arbitrary dimension.

D. Eppstein assume a model that a bivariate function $d(s,t)$ measuring the "distance" between two objects s and t. Noting that this function need not satisfies the triangle inequality or other common properties of the distance. The only requirement is that the function values are totally ordered and it can be computed and compared in constant time. The task of the data structure is to maintain the minimum value $d(s,t)$ among all objects in the set S. Eppstein gave a $O(n\log^2 n)$ time per update with space $O(n)$ algorithm. In the case the computation of the distance may consist of complicated dynamic programming routines, then the space requirement is $O(n^2)$ by precomputing and storing the distance matrix[17].

The data structure consists of a partition of the dynamic set S into $k \leq \log n$ subsets S_1, S_2, ..., S_k together with a digraph G_i for each subset S_i. Initially, all points are in S_1, and g_1 has $n-1$ edges. G_i may contain edges with neither endpoint in S_i. If the number of the edges in all graphs grows to $2n$, the data structure is rebuilt by moving all points to S_1 and recomputing G_1. The closest pair is represented by an edge in some G_i, hence we can find it by scanning the edges in all graphs. If the update operations cause the partition number k to be too large relative to n, we will choose subsets S_i and S_j with approximately same size, and merge them into one, then recompute the corresponding graph G_i.

5.3 Planar Width

The width of a geometric object is the minimum distance between two parallel supporting hyperplanes. The width of a planar points set can be found from its convex hull by a simple linear time algorithm that sweeps through all possible slopes, finding the points of tangency of the two supporting lines for each slope[31]. If the point set undergoes the insertions and deletions, there are many attempts to maintain the planar width dynamically. The methods by Janardan[24], Rote, Schwarz and Snoeyink[33] maintain only an approximation to the true width. An algorithm of Agarwal *et al.* can maintain the exact width, but requires superlinear time per update. Eppstein[18] present an algorithm for maintaining the exact width dynamically takes time $O(kn^\epsilon)$ per update, where k is the amount of change the update causes in the convex hull of the point set, n is the number of point in the set, ϵ is a arbitrarily small constant. In particular, if only insertions or deletions occur, the total change to the convex hull is at most linear, and then the algorithms takes $O(n^\epsilon)$ amortized time per update. For randomized model of insertions and deletions, the expected value of k is also $O(1)$, so the amortized update time is still $O(n^\epsilon)$. The approach is to use a data structure of Eppstein for maintaining the minimum of a bivariate function defined on a dynamic set of objects. The width is defined as a minimum of a function defined on the features of the point set's convex hull by maintaining a dynamic nearest neighbor querying on subsets of features.

5.4 Moving Points

In the dynamic simulation of virtual environments, collision detection between the moving objects is frequently and important. Most of the algorithms discussed before are not directly applicable to the set of objects moving continuously. The dynamic algorithms allow insertions and deletions from a set of objects. It is not immediately obvious how to use these classical algorithms to maintain the closest point pair as the points move continuously. Guibas et al[5] proposed a kinetic data structure to solve this kind of problems by capturing the continuous moving property. They assume that each object follows a posted *flight plan*, but the plan can be changed at any moment through a *flight plan update*. A kinetic data structure maintains a *configuration function* of interest(in our case, this will be the closest pair) by watching for critical events as the objects move. The events correspond to the times at which elementary conditions, called *certificates*, switch from being true to being false. When a certificate fails, the proof structure needs to be modified and the combinatorial description of the configuration function may need to be updated also. A good data structure for moving objects should have the following desiderata conditions[5].

- *Responsible*: the kinetic data structure takes advantage of continuity of the object motion to select certificate structures which are easy to update at these critical events.
- *efficient*: meaning that the number of events it preprocesses is not much greater than the number of combinatorial changes in the configuration function itself.
- *compact*: the number of active certificates at any time is roughly linear in the complexity of the moving objects.
- *local*: a flight plan update for any object affects only a small number of certificates.

The results of Guibas et al are based on kinetizing a certain type of multidimensional range search tree(MDRS) used to query the value of some specified function on subsets of the points, by assuming that the motions of the points are algebraic of fixed degree, or that they satisfy certain Davenport-Schinzel type conditions. Their result implies a algorithm for static point set with $O(n \log^d n)$ time complexity.

6 Other Problems

6.1 Maximal Points

Let S be a set of n points in the plane. Point $p = (x_p, y_p)$ *dominates* point $q = (x_q, y_q)$ if $(x_p \geq x_q)$ and $y_p \geq y_q$. A point is the maximal point of S if there are no any other points dominate it. The set of maximal points of S form a structure which monotonically decreases in the y-direction as the x coordinates of the points increases. Such a structure is called a *staircase structure*. 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. For the static point set S, it has been shown that the staircase structure can be computed in $O(n \log n)$ time eliminating dominated points[9]. Overmars and Van Leeuwen[30] have designed a data structure which requires splitting and merging balanced trees when points are inserted and deleted. The $O(\log^2 n)$ operations are required for each insertion and deletion. Fredrickson and Rodger[21] and Janardan[23] have designed a scheme which maintains the staircase structure of the set of maximas and allows the insertion in $(\log n)$, and deletions in

$O(\log^2 n)$ operations. Testing whether a point is inside, on, or outside the m-contour takes $O(\log n)$ time. Let k be the size of the m-contour. Then the m-contour can be reported in $O(k)$ by algorithm of Fredrickson and Rodger[21], and $O(k + \log n)$ by algorithm of Janardan.

S. Kapoor[25] present an improved data structure that maintains the staircase structure in $O(\log n)$ time per insertion and deletion. A point can be tested to be a maxima in $O(\log n)$ time in their representation. The staircase structure representing the set of maximas can be listed out in time linear in k, the size of the set of maximas, but with $O(\log n)$ operations to be spent on changes in the report of the maximal element, i.e., the output complexity is $O(k + c * \log n)$, where c is the number of changes since last report. The space requirement is $O(n)$.

References

[1] P.K.Agarwal, L.Arge, G.S.Brodal J.S. Vitter. I/O-efficient Dynamic Point Location in Monotone Planar Subdivisions. http://www.cs.duke.edu/ pankaj.

[2] P.K.Agarwal, J. Erickson. Geometry Range searching and its relatives. Contemporary Mathematics.

[3] A.Aragon, R.G. Seidel. Randomized Search Trees, Proc. IEEE Symp. on Foundations of Computer Science, 1989,540-545.

[4] S.W.Bent. Dynamic Weighted Data structures, Stanford Univ., Ph.D dissertation, 1982.

[5] J.Basch, L.J.Guibas, and L.Zhang. Proximity Problems on Moving Points. Proc. of 13th Computational Geometry, 1997,344-351.

[6] S.W.Bent, D.D.Sleator, and R.E.Tarjan. Biased Search Trees. SIAM Journal of Computing 14, 1985, 545-568.

[7] J.L.Bently, and J.Saxe. Decomposable Searching Problems I:Static to Dynamic Transformations. Journal of Algorithms 1, 1980,301-358.

[8] B.Chazelle, Triangulating a Simple Polygon in Linear Time. Proc. of IEEE Symp. on Foundation of Computer Science, 1990,220-230.

[9] T.H.Cormen, C.E.Leiserson, and R.L.Rivest. Introduction to Algorithms. The MIT Electrical Engineering and Computer Science Series,1997.

[10] Y.-J. Chiang. http://www.cs.brown.edu/ yjc.

[11] Y.-J. Chiang, F.P. Preparata, and R. Tamassia. A Unified Approach to Dynamic Point Location, Ray Shooting and Shortest Paths in Planar Maps. SIAM Journal on Computing, vol. 25(1), pp. 207–233, 1996.

[12] Y.-J. Chiang and R. Tamassia. Dynamization of the Trapezoid Method for Planar Point Location in Monotone Subdivisions. Int. J. of Computational Geometry and Applications, Special Issue, D.T. Lee (Ed.), vol. 2(3), pp. 311-333, 1992.

[13] Y.-J. Chiang and R. Tamassia. Dynamic Algorithms in Computational Geometry. Proceedings of the IEEE, Special Issue on Computational Geometry, G. Toussaint (Ed.), vol. 80(9), pp. 1412-1434, 1992.

[14] M.T.Dickerson, and R.S.Drysdale, Enumerating k distances for n points in the plane. Proc. ACM Symp. on Computation Geometry, 1991,234-238.

[15] Herbert Edelsbrunner. Optimizing the Dynamization of Decomposable Searching Problems. IIG,Technische Univ. Graz, Austria, Rep35,1979.

[16] Herbert Edelsbrunner. Algorithms in combinatorial geometry. Monographs on theoretical computer science, 1987, Springer-Verlag.

[17] D.Eppstein. Fast Hierarchical Clustering and other Applications of Dynamic Closest Pairs. Proc. of 9th ACM-SIAM SODA, San Francisco,1998,619-628.

[18] D.Eppstein. Incremental and Decremental Maintenance for Planar Width. Computational Geometry,1999.

[19] J. Erickson, L.J. Guibas, J.Stolfi, L.Zhang. Separation-sensitive Collision Detection for Convex Objects. SODA99.

[20] S. Fortune. A sweep line algorithm for Voronoi diagrams. *Algorithmica*, 2:153-174, 1987.

[21] G.N.Fredrickson, and S.Rodger. A new approach to the dynamic maintenance of maximal points in the plane. Discrete and Computational Geometry, 5, 1990,365-374.

[22] M.T.Goodrich and R.Tamassia, Dynamic trees and Dynamic Point Location. Proc. of 23th ACM Symp. on Theory of Computing,1991,525-533.

[23] R. Janardan. On the dynamic maintenance of maximal points in the plane. 40(2),1991,59-64.

[24] R. Janardan. On maintaining the width and diameter of a planar point-set online. Int. J. of Computational Geometry and Applications 3:331-344, 1993.

[25] Sanjiv Kapoor. Dynamic Maintenance of Maximas of 2-d Point Sets. Proc. of 10th Computational Geometry, 1994,140-149.

[26] J. Matousek. Geometric range searching. ACM computing surveys, Vol.26, No.4, December 1994.

[27] Kurt Melhorn. Multi-dimensional searching and computational geometry. Monographs on theoretical computer science, 1984, Springer-Verlag.

[28] Ketan Mulmuley. Computational Geometry: an introduction through randomized algorithms. 1994, Prentice-Hall Inc.

[29] M.H.Overmars, Range Searching in a Set of Line Segments. Proc. ACM Symp. on Computational Geometry, 1985,177-185.

[30] M.H.Overmars, and J.L.Van Leeuwen. Maintenance of Configurations in the Plane. Journal of Comp. System Sci., 23, 1981,253-257.

[31] F.P.Preparata, M.I. Shamos. Computational geometry: an introduction. texts and Monographs in computer science, 1988, Springer-Verlag.

[32] F.P.Preparata and R.Tamassia, Fully Dynamic Point Location in Monotone Subdivision. SIAM Journal of Computing 18,1989 811-830.

[33] G. Rote, C.Schwarz, and J. Snoeyink. Maintaining the approximate width of a set of points in the plane. Proc. 5th Canad. Conf. Computational Geometry, 258-263, 1993.

[34] M.I.Shamos. Computational Geometry. Doctoral Dissertation, Yale University, 1978.

[35] M.Smid. Maintaining the Minimal Distance of Point Set in Less than Linear Times. Algorithms Review 2(1),1991,33-44.

[36] D.D.Sleator, and Robert E. Tarjan. A Data Structure for Dynamic Trees. Journal of Computer and System Sciences 24, 1983,362-381.

[37] Robert E. Tarjan. Data structures and Network Algorithms. Society for Industrial and Applied Mathematics, 1983.

[38] R.Tamassia, An Incremental Reconstruction Method for Dynamic Planar Point Location. Information Processing Letters 37, 1991,79-83.

[39] F.F.Yao, Computational Geometry. in handbook of Theoretical Computer Science, Volume A, Algorithms and Complexity, Elsevier/MIT press, 1990, 343-390.