Average case analysis of dynamic geometric optimization

David Eppstein*
Department of Information and Computer Science
University of California, Irvine, CA 92717

May 19, 1995

Abstract

We maintain the maximum spanning tree of a planar point set, as points are inserted or deleted, in $O(\log^3 n)$ expected time per update in Mulmuley's average-case model of dynamic geometric computation. We use as subroutines dynamic algorithms for two other geometric graphs: the farthest neighbor forest and the *rotating caliper graph* related to an algorithm for static computation of point set widths and diameters. We maintain the former graph in expected time $O(\log^2 n)$ per update and the latter in expected time $O(\log n)$ per update. We also use the rotating caliper graph to maintain the diameter, width, and minimum enclosing rectangle of a point set in expected time $O(\log n)$ per update. A subproblem uses a technique for average-case orthogonal range search that may also be of interest.

*Work supported in part by NSF grant CCR-9258355. A preliminary version of this paper appeared at the 5th ACM-SIAM Symp. on Discrete Algorithms, 1994, pp. 77–86.

1 Introduction

Randomized incremental algorithms have become an increasingly popular method for constructing geometric structures such as convex hulls and arrangements. Such algorithms can also be used to maintain structures for *dynamic* input, in which points are inserted one at a time. Mulmuley [30, 31, 32] and Schwarzkopf [38] generalized this expected case model to fully dynamic geometric algorithms, in which deletions as well as insertions are allowed, and showed that many randomized incremental algorithms can be extended to this fully dynamic model. The resulting model makes only weak assumptions about the input distribution: the *order* in which points are inserted or deleted is assumed to be random, but both the points themselves and the times at which insertions and deletions occur are assumed to be a worst case. The model subsumes any situation in which points are drawn from some fixed but unknown distribution, and in this sense it is distributionless.

In many dynamic geometry problems a worst case efficient dynamic algorithm is impossible because the given geometric structure can undergo enormous change in a single update. In such a case the much smaller time bounds given by the average case analysis provide an indication that such pathological behavior is unlikely, and that a worst case analysis may therefore be inappropriate. The design of average case efficient algorithms then shows how to take advantage of this situation. But we would argue that average case analysis is important even when worst case algorithms are possible. The average case algorithms are often simpler, have better theoretical time bounds, and seem more likely to be practical.

Previous studies of average case updates in this model have focused on problems of computing geometric structures: convex hulls, arrangements, and the like. However problems of geometric optimization have been neglected; indeed Mulmuley's text on randomized geometric algorithms [32] does not even mention such basic optimization problems as minimum spanning trees or diameter. In this paper we show that the same model of average case analysis can be used to solve a number of problems of dynamic geometric optimization. We also address some fundamental data structure issues raised by such optimization problems. In particular, many worst case geometric optimization algorithms use a variety of reductions from one problem to another, such as the static-to-dynamic reduction of Bentley and Saxe [9]; we examine methods for performing this sort of reduction in a way that preserves the average case behavior of the update sequence.

Our work also represents the first use of this type of average case analysis in dynamic graph algorithms. After this paper first appeared, Alberts and Rauch Henzinger used a similar average case model for more purely graph-theoretic dynamic problems: in their model, edges are inserted and deleted from a graph with each insertion and deletion chosen randomly from a predetermined space of possible edges [5]. Cattaneo and Italiano [11] have implemented and analyzed several algorithms using the same average case edge update model. It would be interesting to extend these results to an average case model of vertex updates in non-geometric graphs, which would more closely resemble the geometric model we use here.

1.1 Motivation

To further motivate the average case analysis model, and point out some of its applications to geometric optimization, we first describe three problems for which an average case dynamic algorithm is an immediate corollary of known results, but for which the known worst case efficient algorithms were considerably more complicated.

Diameter. A dynamic planar diameter algorithm is known with worst case time per update $O(n^\epsilon)$ [2]. This algorithm first performs a reduction (with logarithmic overhead) to finding farthest neighbors of query points in dynamic point sets. This farthest neighbor problem can be further reduced by parametric search to a problem of testing halfspace emptiness in three dimensions. The latter problem can be solved using a complicated range query data structure based on deterministic sampling techniques; this final step is the one that determines the $O(n^\epsilon)$ worst case time bound. But the diameter is also the longest edge in the farthest point Delaunay triangulation, which can be maintained in $O(\log n)$ expected time per update using techniques of Mulmuley [31]. We will see that an even simpler algorithm can also maintain the diameter in $O(\log n)$ expected time per update.

Linear programming. In a previous paper [17], we described randomized algorithms for the dynamic linear programming problem in three dimensions. Our bounds have since been improved by Agarwal et al. [2] to $O(n \log^{O(1)} n/m^{1/\lceil d/2 \rceil})$ query time and $O(m^{1+\epsilon}/n)$ update time in any dimension, using m space (where the value of m can be chosen to trade off update and query time). But in the average case, if the objective function is fixed, Seidel's algorithm [39] can easily be adapted to

provide a constant expected time bound per update. The same result applies to the many related nonlinear optimization problems which can be solved by the same algorithm [6].

Minimum spanning tree. The planar minimum spanning tree problem can be reduced to a graph problem in a graph formed by a number of bichromatic closest pair problems, which could then be solved with the same techniques used for diameter [2]. By combining this idea with clustering techniques for graph minimum spanning trees [20, 21, 23], we were able to use this idea to solve the minimum spanning tree problem in worst case time $O(n^{1/2}\log^2 n)$ per update [2]. But in the average case, the minimum spanning tree can be maintained much more easily in expected time $O(\log n)$ per update by combining a dynamic Delaunay triangulation algorithm [31] with a dynamic planar graph minimum spanning tree algorithm [22].

1.2 New Results

We provide the first dynamic algorithms for the following problems:

Maximum spanning tree. Like the minimum spanning tree, the maximum spanning tree has applications in cluster analysis [7]. For graphs, the maximum spanning tree problem can be transformed to a minimum spanning tree problem and vice versa simply by negating edge weights. For geometric input, the maximum spanning tree is very different from the minimum spanning tree; for instance, although the minimum spanning tree is contained in the Delaunay triangulation [40] the maximum spanning tree is not contained in the farthest point Delaunay triangulation [28]. Another important difference from the minimum spanning tree is that, whereas the minimum spanning tree changes by $O(1)$ edges per update, the maximum spanning tree may change by $\Omega(n)$ edges. Hence a worst case efficient algorithm is impossible. But in the average case, the maximum spanning tree changes by an expected $O(1)$ edges per update (Lemma 1), so efficient algorithms may be possible. Monma *et al.* [28] compute a static maximum spanning tree in time $O(n\log n)$. We dynamize their algorithm, and produce a data structure which can update the maximum spanning tree in expected time $O(\log^3 n)$ per update.

Width. The problem of maintaining the width of a point set appears to be much more difficult than that of maintaining its diameter, and no satisfactory worst case solutions are known. Agarwal and Sharir [4] describe an algorithm for testing whether the width is above or below some threshhold, in the *offline* setting for which the entire update sequence is known in advance. Janardan [26] and Rote et al. [37] maintain approximations to the width in time $O(\log^2 n)$ per update. But none of these results is a fully dynamic algorithm for the exact width. We describe a very simple algorithm for maintaining the exact width in our fully dynamic average-case model, in expected time $O(\log n)$ per update. The same techniques can also be used to maintain the diameter as well as the minimum area or perimeter enclosing (not necessarily axis-aligned) rectangle. The diameter result could also be achieved using farthest point Voronoi diagrams, but our algorithm may be simpler. We are unaware of any dynamic algorithm for enclosing rectangles.

As part of our maximum spanning tree algorithm, we describe dynamic algorithms for two other geometric graphs: the farthest neighbor forest and also the *rotating caliper graph* related to an algorithm for static computation of widths and diameters. We maintain the former graph in expected time $O(\log^2 n)$ per update and the latter in expected time $O(\log n)$ per update. We also use the rotating caliper graph in our algorithm for maintaining the width of a point set and related quantities.

One further feature of our maximum spanning tree algorithm is of interest. The known dynamic minimum spanning tree algorithms all use some geometry to construct a subgraph of the complete graph, and then use a dynamic graph algorithm to compute spanning trees in that subgraph. The subgraphs for incremental and offline geometric minimum spanning trees [18] are found by computing nearest neighbors in certain directions from each point [42]; the subgraph for fully dynamic minimum spanning trees uses bichromatic closest pairs [1]; and the fully dynamic average case algorithm uses as its subgraph the Delaunay triangulation. In our maximum spanning tree algorithm, as in that of Monma *et al.* [28], no such strategy is used. Instead, the maximum spanning tree is computed directly from the geometry, with no need to use a graph maximum spanning tree algorithm.

1.3 Organization

This paper is organized as follows.

We first (Section 2) describe Mulmuley and Schwarzkopf's expected case model of dynamic geometry problems, and use the model to show that the maximum spanning tree and several other geometric graphs only incur a constant expected amount of change per update. We also discuss *decomposable searching problems* and describe a data structure for solving such problems that fits well with the average case model we have in mind.

Next (in Section 3), we describe a characterization of the maximum spanning tree, due to Monma *et al.* [28], in terms of farthest neighbors of points and diameters of certain sets of convex hull vertices. Basically, if one finds a *farthest neighbor forest* by connecting each point to the farthest other point in the set, the maximum spanning tree can be found by connecting the trees of this forest in a systematic way according to the ordering of the vertices on the convex hull of the input.

Finally, we describe the three parts of our maximum spanning tree algorithm. The first part is simply the construction and maintenance of the farthest neighbor forest; in Section 4 we perform this construction using the farthest point Voronoi diagram (which can be maintained dynamically using known techniques). We keep track of the diagram region containing each input point by applying data structures for dynamic convex hulls and point location. We then (in Section 5) use dynamic convex hulls a second time, together with a dynamic tree data structure due to Sleator and Tarjan, to keep track of the separate trees in this forest and to find the intervals they form on the convex hull of the input.

The second part of our maximum spanning tree algorithm (Section 6) is the maintenance of a second geometric graph, the *rotating caliper graph* which we define in terms of the action of the rotating caliper method for computing diameters and widths of convex polygons. This graph also changes very little in expectation per update. The rotating caliper graph can be maintained very simply using a dynamic convex hull structure.

Finally (in Section 7) we combine the decomposable searching method of Section 2 with the rotating caliper graph of Section 6 to solve the following question: given two trees in the farthest neighbor forest, what is the longest edge connecting the two? In Section 8 we show how to use all but one of the edges obtained by such queries to augment the farthest neighbor forest and form the maximum spanning tree.

We then (Section 9) show that several quantities including the width and

diameter of a point set can be maintained with logarithmic expected time per update, again using the rotating caliper graph. We conclude with some related open problems.

2 The Model of Expected Case Input

We now define a notion of the expected case for a dynamic geometry problem. The definition we use has been popularized in a sequence of papers by Mulmuley [30, 31, 32], and is a generalization of the commonly occurring *randomized incremental* algorithms from computational geometry [14, 16, 25, 27, 29, 39]. We discuss the latter concept first, to motivate our definition of the expected case.

An *incremental algorithm* maintains the solution to some problem as parts of the problem are added one at a time; for instance we might compute the minimum spanning tree of a point set by adding points one by one, reconstructing the minimum spanning tree of the partial point set after each addition. A *randomized incremental algorithm* adds to this idea a preprocessing stage in which the order in which points are added is selected at random. Such an algorithm may or may not behave well in the worst case, but typically has good behavior when the order in which the points are added is chosen uniformly among all possible permutations.

Randomized incremental algorithms have been used to construct a number of geometric configurations including convex hulls [14, 16, 39], Voronoi diagrams and Delaunay triangulations [25, 27], linear programming optima [39], and intersection graphs of line segments [29]. An important feature of these algorithms is that they do not depend on any special properties of the input point set, such as even distribution on the unit square, that might arise from other random input distributions. The fast expected time bound is in the average case over all random permutations of a worst-case point set.

These algorithms have typically been studied as static algorithms, which compute a random order and then perform incremental insertions using that order. Often the randomized incremental algorithm for a problem is simpler than the best known deterministic algorithms, and it may either match or improve the performance of those algorithms. However, such algorithms can also be used as dynamic incremental algorithms: if we wish to compute the sequence of solutions to some dynamic problem in which points are added one at a time, and the input comes from some random distribution such

that any permutation of the input points is equally likely, then a randomized incremental algorithms will perform well in expectation, typically taking as much time to compute the entire sequence of problem solutions as a static algorithm would take to compute a single solution. Such an average case input model, in which the only assumption is that the order of the points is random, is *distributionless*, in that it subsumes any model in which input points are drawn independently from some particular distribution.

Mulmuley [30, 31] and Schwarzkopf [38] generalized this notion of average case from incremental algorithms to fully dynamic geometric algorithms in which points can be deleted as well as inserted. They also showed that many randomized incremental algorithms can be extended to this fully dynamic model, by techniques involving searches through the history of the update sequence.

We define a *signature* of size n to be a set S of n input points, together with a string s of length at most $2n$ consisting of the two characters "+" and "−". Each "+" represents an insertion, and each "−" represents a deletion. In each prefix of s, there must be at least as many "+" characters as there are "−" characters, corresponding to the fact that one can only delete as many points as one has already inserted. Each signature determines a space of as many as $(n!)^2$ update sequences, as follows. One goes through the string s from left to right, one character at a time, determining one update per character. For each "+" character, one chooses a point x from S uniformly at random among those points that have not yet been inserted, and inserts it as an update in the dynamic problem. For each "−" character, one chooses a point uniformly at random among those points still part of the problem, and updates the problem by deleting that point.

The expected time for an algorithm on a given signature is the average of the time taken by that algorithm over all possible update sequences determined by the signature. Only the sequence of updates, and not the signature, is available to the algorithm. The expected time on inputs of size n is defined to be the maximum expected time on any signature of size n. We choose the signature to force the worst case behavior of the algorithm, but once the signature is chosen the algorithm can expect the update sequence to be chosen randomly from all sequences consistent with the signature. As a special case, the average case for randomized incremental algorithms is generated by restricting our attention to signatures containing only the "+" character.

This average case model can be used in situations for which no worst-case efficient algorithm is possible; for instance, the Delaunay triangulation may

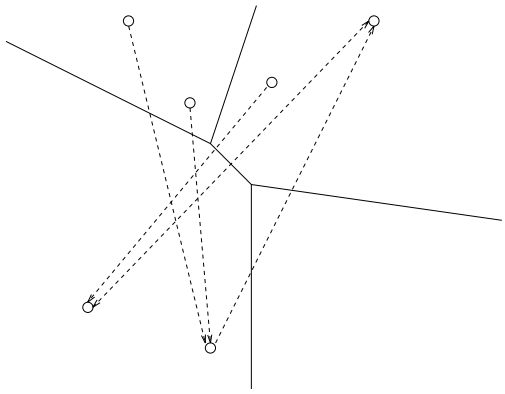

Figure 1. Farthest point Voronoi diagram and farthest neighbor forest.

change by $\Omega(n)$ edges per update in the worst case, but in the average case the expected amount of change is $O(1)$, and in fact an $O(\log n)$ expected time algorithm is possible [15, 30]. Alternately, average case analysis can be used to speed up the solution to problems for which the best known worst-case bound is too large; as an example, the minimum spanning tree has an algorithm with $O(n^{1/2}\log^2 n)$ worst case time per update [2, 19] but can be solved in $O(\log n)$ expected time per update as described earlier.

Our application is of the former type: the maximum spanning tree can change by $\Omega(n)$ edges in the worst case but only $O(1)$ in the expected case. To demonstrate the power of this expected case model, and derive a fact we will need in our maximum spanning tree algorithm, we prove this bound, and similarly bound the expected change in several other geometric graphs. We have already discussed the maximum spanning tree. The *farthest point Voronoi diagram* is a subdivision of the plane into regions, each consisting of the locus of points having some particular input point as farthest neighbor. Its planar dual is the *farthest point Delaunay triangulation*. The *farthest neighbor forest* is formed by connecting each point with its farthest neighbor. Figure 1 depicts the farthest point Voronoi diagram of a point set overlayed with its farthest neighbor forest; note that all points in a single cell of the diagram have the same neighbor in the forest. Although the farthest neighbor forest is most naturally a directed graph, we will ignore the orientations of its edges when using it as part of a maximum spanning tree.

Lemma 1. *The expected number of edges that change per update in the maximum spanning tree or farthest neighbor forest is* $O(1)$.

Proof: We first consider the change per insertion. Consider the state of the system after insertion i, consisting of some set of j points. Among all sequences of updates leading to the present configuration, any of the j points is equally likely to have been the point just inserted. Both the maximum spanning tree and the farthest neighbor forest have at most $n-1$ edges, where n denotes the size of the signature. Any edge will have just been added to the graph if and only if one of its endpoints was the point just inserted, which will be true with probability $2/j$. So the expected number of additional edges per insertion is at most $O(j) \cdot 2/j = O(1)$.

Since the number of edges in the maximum spanning tree is exactly $n-1$, and the number of directed edges in the farthest neighbor forest is exactly n, the number of existing edges removed in the insertion is less than the number of edges added. Thus the total expected change per insertion is $O(1)$.

The total change per deletion can be analysed by a similar argument that examines the graph before the deletion, and computes the probability of each edge being removed in the deletion. □

We will also need the following facts. The lemma below on the farthest point Delaunay triangulation can be seen via the standard "lifting transformation" as a special case of a similar result for convex hulls in $\mathbb{R}^3$. Similar bounds on the convex hull change per update are known in any dimension [16].

Lemma 2. *The expected number of convex hull vertices that change per update is $O(1)$.*

Proof: We bound the change per insertion; deletions follow a symmetric argument. In each insertion, the only vertex that can be added is the inserted point, so we need only worry about removed vertices. Consider the point set after the insertion. For each convex hull vertex v form a triangle connecting v to its neighbors on either side. Each input point is in at most two such triangles, and can only have been just removed as a hull vertex if the newly added point was one of two triangle vertices. This is true with probability $2/n$ so the expected number of vertices removed is no more than two. □

Lemma 3. *The expected number of edges that change per update in the farthest point Delaunay triangulation is $O(1)$.*

Proof: As in the previous lemmas it suffices to prove the result for insertions; deletions are symmetric. The farthest point Delaunay triangulation has $2h-3$ edges where h denotes the number of convex hull vertices. By the same argument used in Lemma 1, the expected number of edges added after an insertion is $O(1)$. The number of previously existing edges removed is proportional to the number of edges added, plus the difference in the total number of edges before and after the update. This difference is the same as the amount of change in the number of convex hull vertices, bounded in expectation by $O(1)$ in Lemma 2. □

A similar bound on the change in a fifth geometric graph, the *rotating caliper graph*, is proved later in Lemma 13.

We next describe a method for transforming any efficient data structure in this expected case model, to one which combines its queries with *orthogonal range searching*. I.e., we wish to ask for the answer to a query such as a farthest neighbor problem, with the queried set being a subspace of the input within some rectangle or higher dimensional box given as part of the query. We consider any *decomposable search problem* [9] for which the answer for a given input set can be found quickly by combining answers in disjoint subsets of the input. We describe a solution for the one-dimensional case, *interval range searching*; higher dimensional box range queries can be solved by iterating our construction once per dimension.

Many techniques are known for performing orthogonal range searching in decomposable search problems, but these techniques often are based on complicated balanced binary tree data structures that do not lend themselves to easy expected case analysis. We instead generalize a technique which Mulmuley [32] used to answer line segment intersection queries.

Lemma 4. *Let P be a decomposable search problem for which queries and updates can be performed in expected time $T(n)$. Then there is a dynamic data structure that can perform interval range queries of P, in expected time $O(T(n)\log n)$ per query and update.*

Proof: We partition the problem into a number of subproblems using a *skip list* [36], as follows. We sort the coordinates of the input points, giving a partition of the line into $n+1$ open intervals, and provide an (empty) subproblem for each such interval. Then for each point we flip a fair coin independently of all other points. If the coin is heads, the point is removed from the sorted list. The remaining points again partition the line into a sequence of open intervals; the expected number of intervals is $n/2+1$. For

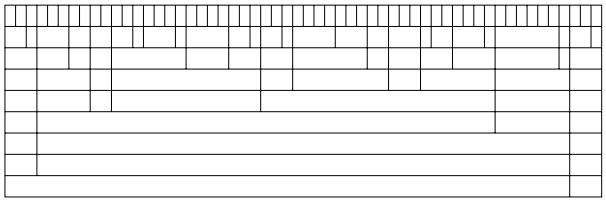

Figure 2. Skip list recursively partitions coordinates into open intervals.

each such interval we provide a subproblem for all input points contained in the interval. Figure 2 depicts the recursive partition of coordinates into open intervals formed by performing this process.

With high probability after $O(\log n)$ iterations all points will have flipped a head, and the single subproblem will include all points, so there are $O(\log n)$ levels of subproblems. With high probability any query interval can be composed of $O(\log n)$ subproblems (the expected number of subproblems at any level of the skip list is $O(1)$; some levels may use more but the overall expectation is $O(\log n)$). So any query can be answered in expected time $O(T(n) \log n)$.

When we insert a new point, we repeatedly flip a coin until a head is flipped, to determine the number of levels for which the new point is a partition boundary. At each such level the point is inserted and some subproblem is split to make two new subproblems. The data structure for each new subproblem is rebuilt by inserting its points in a random order. The expected size of a subproblem at level i is 2^i, so the expected time to rebuild the subproblem is $2^i T(n)$, but the probability of having to do so at level i is 2^{-i}. Therefore the expected total work in rebuilding the first $\log n$ levels of the data structure is $\sum_{i=1}^{\log n} T(n) = O(T(n) \log n)$. Higher levels can occur as well, but the expected time for rebuilding them is bounded by the total expected size of the data structure above the first $\log n$ levels, which is $O(1)$.

Each inserted point must then be inserted into one subproblem for each level higher than the one for which it flipped a head. Each such insertion is done using the data structure for that subproblem. After any insertion to a subproblem, given some particular set of points now existing in the subproblem, any permutation of those points is equally likely as the insertion order, so the expected-case nature of the input sequence holds for each subproblem and the expected time per subproblem insertion at level i is $T(n)$. Again the total expected time is $O(T(n) \log n)$.

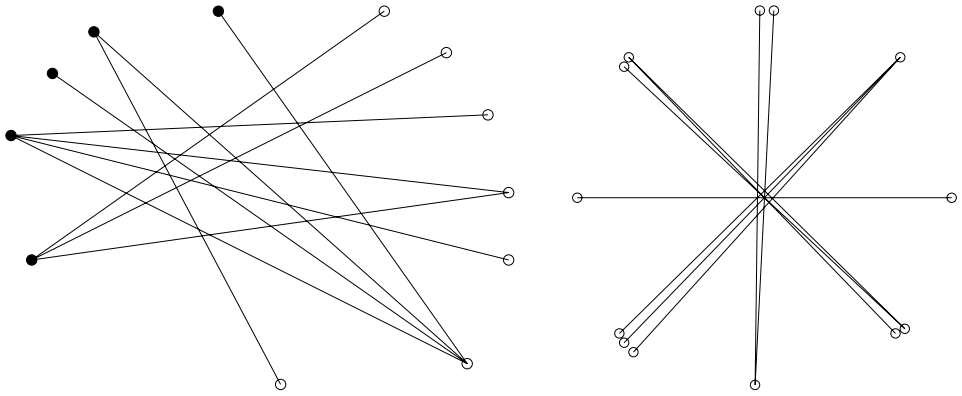

Figure 3. (a) Two-colored farthest neighbor tree forming two monochromatic intervals; (b) four trees of farthest neighbor forest in cyclic order.

Deletions are performed analogously to insertions, and the time for deletions can be shown to be $O(T(n) \log n)$ using a symmetric argument to that for insertions. □

3 Properties of the Maximum Spanning Tree

We now examine the edges that can occur in the maximum spanning tree. One might guess, by analogy to the fact that the minimum spanning tree is a subgraph of the Delaunay triangulation, that the maximum spanning tree is a subgraph of the farthest point Delaunay triangulation. Unfortunately this is far from being the case—the farthest point Delaunay triangulation can only connect convex hull vertices, and it is planar whereas the maximum spanning tree typically has many crossings. However we will make use of the farthest point Delaunay triangulation in maintaining the farthest neighbor forest.

Most of the material in this section is due to Monma *et al.* [28], and the proofs of the following facts can be found in that paper. The first fact we need is a standard property of graph minimum or maximum spanning trees.

Lemma 5. *The farthest neighbor forest is a subgraph of the maximum spanning tree.*

Lemma 6 (Monma *et al.* [28]). *Let the vertices each tree of the farthest neighbor forest be two-colored (so that no two adjacent vertices have the*

same color). Then for each such tree, the points of any one color form a contiguous nonempty interval of the convex hull vertices. The trees of the forest can be given a cyclic ordering such that adjacent intervals on the convex hull correspond to trees that are adjacent in the ordering.

Figure 3(a) depicts a farthest neighbor tree two-colored as described in Lemma 6. Figure 3(b) depicts a farthest neighbor forest with four trees in cyclic order; the trees form eight intervals, each of which is adjacent to two others determined by the cyclic order.

Lemma 7 (Monma *et al.* [28]). *Let $e = (x, y)$ be an edge in the maximum spanning tree but not in the farthest neighbor forest, with x in some farthest point neighbor tree T. Then x and y are both convex hull vertices, and y is in a tree adjacent to T in the cyclic ordering of Lemma 6.*

Lemma 8 (Monma *et al.* [28]). *The maximum spanning tree can be constructed by computing the farthest neighbor forest, determining the cyclic ordering of Lemma 6, finding the longest edge between each adjacent pair of trees in the cyclic ordering, and removing the shortest such edge.*

Monma *et al.* [28] show that each of these steps can be performed in time $O(n \log n)$, and hence that a static maximum spanning tree can be found in that time bound. Our algorithm performs a similar sequence of steps dynamically: we maintain a dynamic farthest neighbor forest, keep track of the intervals induced on the convex hull and of the cyclic ordering of the intervals, and recompute longest edges as necessary between adjacent intervals using a dynamic geometric graph defined using the *rotating caliper* algorithm for static diameter computation.

4 Maintaining the Farthest Neighbor Forest

As the first part of our dynamic maximum spanning tree algorithm, we show how to maintain the farthest neighbor forest. As shown in Lemma 1, the expected number of edges per update by which this graph changes is $O(1)$. We find the farthest neighbor to any point by determining the region in the farthest point Voronoi diagram containing that point.

Lemma 9. *We can maintain the farthest point Voronoi diagram in expected time $O(\log n)$ per update.*

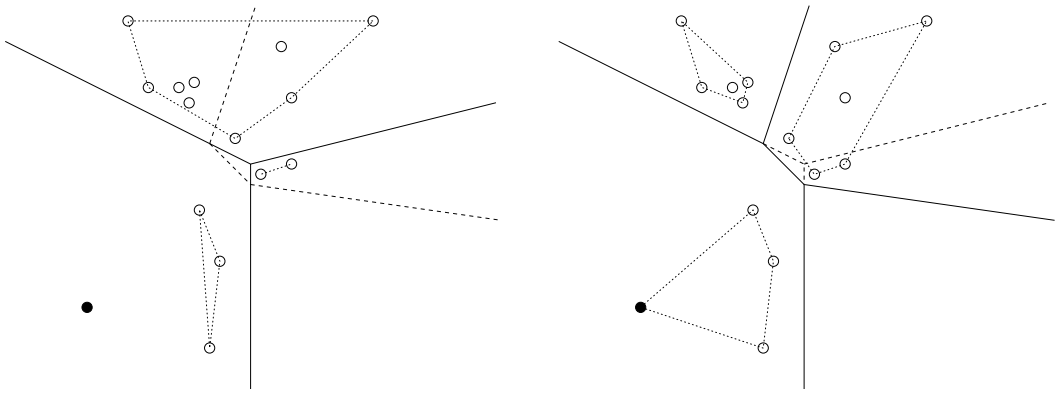

Figure 4. Inserting or deleting the solid point changes the farthest point Voronoi diagram (solid and dashed lines); we maintain the convex hulls of the points in each Voronoi region (dotted lines).

Proof: Since the farthest point Delaunay triangulation is the projection of a three-dimensional convex hull [10], we can maintain it using Mulmuley's dynamic convex hull algorithm [31]. The Voronoi diagram is dual to the Delaunay triangulation, so each change in the Voronoi diagram can be found from a corresponding change in the Delaunay triangulation. □

Along with the farthest point Voronoi diagram itself, we keep track of the set of input points within each diagram cell, using a convex hull data structure (Figure 4). When the diagram is updated, these sets need to be recomputed, and when a point is added to the input it must be added to the appropriate set. The latter operation can be performed using a point location data structure.

Lemma 10. *We can maintain a point location data structure in the farthest point Voronoi diagram in expected time $O(\log^2 n)$ per update or query.*

Proof: We can achieve these bounds per change and per query using any of a number of algorithms [8, 12, 13, 24, 35]. By Lemma 1, the expected amount of change per update is $O(1)$. □

Thus we are left with the problem of updating the potentially large sets of points in each diagram cell, after each change to the diagram. We no longer use the expected-case model for these updates, since our analysis does not indicate when such an update is likely to occur or how many points are likely

to be in the sets. However, we do know that few points are likely to change farthest neighbors as a result of the update.

There are two types of changes that may occur in a farthest point Voronoi diagram update. If a point is added to the input, a corresponding region may be added to the diagram, covering portions of the diagram that were previously parts of other regions. Symmetrically, if a point is removed from the input, its region is also removed, and split up among the remaining regions of the diagram.

In the first case, we must find the input points covered by the new region. For each of the old regions partially covered by the new region, we can find from the Voronoi diagram a line separating the old and new regions. We query the set of points corresponding to the old region, to find those points on the far side of this line from the new point. All such points will change their farthest neighbor to be the new point. We can perform the queries with an algorithm for maintaining the convex hull of the set of points in a region. We test whether the line crosses the convex hull; if not, all or none of the points are in the new region. If it does cross, we can find a convex hull vertex in the new region, remove it from the set of points in the old region, and repeat the process. In this way we perform a number of convex hull operations proportional to the number of points which change farthest neighbors. We can not use a fast expected-time convex hull algorithm, because we do not expect the behavior of the point set in a region to be random, but we can solve the planar dynamic convex hull problem in worst case time $O(\log^2 n)$ per update [33].

In the second case, we must recompute the farthest neighbors of all the points covered by the removed region. We compute the new farthest neighbors in $O(\log^2 n)$ time each, using the same point location structure used when a new point is inserted. The total expected time per farthest neighbor change is $O(\log^2 n)$. Each point is then inserted in the dynamic convex hull structure used for handling the first case, in time $O(\log^2 n)$.

Lemma 11. *We can maintain the farthest neighbor forest of a dynamically changing input in expected time $O(\log^2 n)$ per update.*

Proof: As explained above, this is the time for updating the data structures necessary to compute the farthest neighbor forest, measured in time units per change to the farthest neighbor forest. By Lemma 1 the expected change to the farthest neighbor forest is $O(1)$. □

It is conceivable that this can be improved to $O(\log n)$ per update using a point location technique based more directly on the dynamic farthest point Voronoi diagram, but such an improvement remains open. Mulmuley's dynamic convex hull algorithm uses implicitly an $O(\log n)$ time point location algorithm, but in the farthest point Delaunay triangulation rather than the corresponding Voronoi diagram, so this does not seem to help. Mulmuley [32] gives as an exercise a direct point location algorithm for nearest neighbor Voronoi diagrams with $O(\log n)$ update time, but the query time is still $O(\log^2 n)$. One would also have to replace the dynamic convex hull data structure used here, which contributes another $O(\log^2 n)$ term to the time bound.

5 Convex Hull Intervals from Farthest Neighbors

We saw in the previous section how to maintain the farthest neighbor forest of a point set. Lemma 6 shows that each tree in this forest gives rise to two intervals on the perimeter of the convex hull, one for each color of vertices if the tree is 2-colored. We wish to be able to find those intervals quickly, so that we can use the convex hull subinterval diameter algorithm of Section 7 to find the remaining maximum spanning tree edges not in the farthest neighbor forest. The difficulty is that, even though the farthest neighbor forest changes by a small amount per update, many points may be moved by that change from one tree in the forest to another.

Lemma 12. *We can determine the endpoints of the two intervals described in Lemma 6, for any tree T in the farthest neighbor forest specified by any vertex in that tree, in time $O(\log^2 n)$ per query and expected time $O(\log n)$ per update.*

Proof: We will use the following basic data structures: the dynamic tree data structure of Sleator and Tarjan [41] applied to the farthest neighbor tree; a dynamic planar convex hull data structure, for instance that of Mulmuley [31]; and a balanced binary tree in which the inorder traversal of nodes represents the ordered list of vertices on the convex hull (Figure 5). The dynamic tree can tell us to which color of which tree a point belongs, in time $O(\log n)$ per query. All of these data structures can be updated in $O(\log n)$ time per change, and incur $O(1)$ changes in expectation per update.

We then use the balanced binary tree to guide a binary search for the interval boundaries, at each step of the search using the Sleator-Tarjan dy-

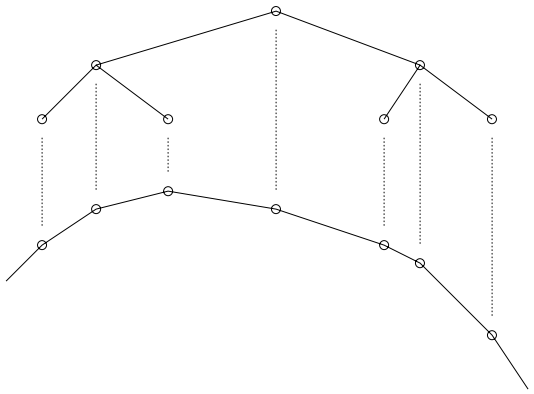

Figure 5. Balanced binary tree represents order of convex hull vertices.

namic tree to test which side of the boundary a given point is on.

In more detail, we can find a single point p_1 in one of T's two intervals simply by choosing the root of T. We can find a point p_2 in the other interval by taking the farthest neighbor of that root. There are two intervals on the convex hull with boundary p_1 and p_2, each of which can be represented as $O(\log n)$ disjoint subtrees of the balanced binary tree. We use the Sleator-Tarjan dynamic tree to test whether the root of each subtree is in T and if so whether it is in the interval corresponding to p_1 or p_2. If a subtree root is adjacent on both sides to other subtree roots of the same type (outside T, or in one of the two intervals inside T), it cannot contain the boundary between T and $S - T$, so we can restrict our attention to at most eight subtrees any one of which can contain one of the four boundaries we seek. (In some cases, there may be one subtree containing two boundaries.)

Within a subtree, we simply follow a path from the root of the tree, down to a leaf, at each step moving left or right depending on whether the node at the given step is in T and if so to which interval of T it belongs. Once our path reaches a leaf, we will know that our interval boundary is either immediately to the left of or to the right of that leaf.

The search described above performed $O(\log n)$ steps (either testing a subtree root or following paths in $O(1)$ subtrees), and at each step in the search we test whether a point belongs to either interval by querying the dynamic tree data structure in time $O(\log n)$, so the whole search takes $O(\log^2 n)$ time. □

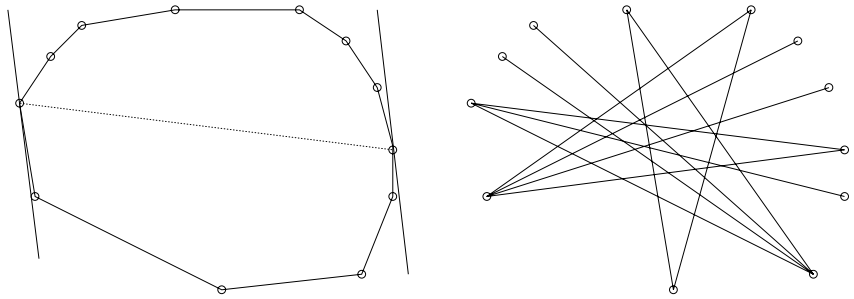

Figure 6. (a) Parallel tangent lines give rise to an edge in the rotating caliper graph; (b) the rotating caliper graph of a point set.

We use the same dynamic tree data structure later, to determine which parts of the farthest neighbor forest have been changed and need updating.

6 The Rotating Caliper Graph

In order to compute the longest edge between two trees of the farthest neighbor forest, we use another dynamic geometric graph, which we call the *rotating caliper graph* after its relation to the static algorithm for computing the width and diameter of planar point sets, known as the rotating caliper algorithm [34]. This *rotating caliper algorithm* considers pairs of parallel lines tangent to vertices of the convex hull of a point set (Figure 6(a)). If we continuously vary the slope of the parallel lines, they will pass through points at which they support convex hull edges, after which the point of tangency will be the other endpoints of the edges. As the slope varies around a circle, the tangent points move monotonically one vertex at a time around the convex hull perimeter.

The rotating caliper graph (Figure 6(b)) is then simply the collection of tangent point pairs considered by the rotating caliper algorithm. Equivalently an edge xy is in the rotating caliper graph exactly when all input points lie between the two parallel lines through x and y and perpendicular to xy. Like the farthest point Delaunay triangulation the rotating caliper graph only connects convex hull vertices.

Lemma 13. *The expected number of edges that change per update in the rotating caliper graph is* $O(1)$.

Proof: The proof is the same as for Lemma 3. That proof only depends

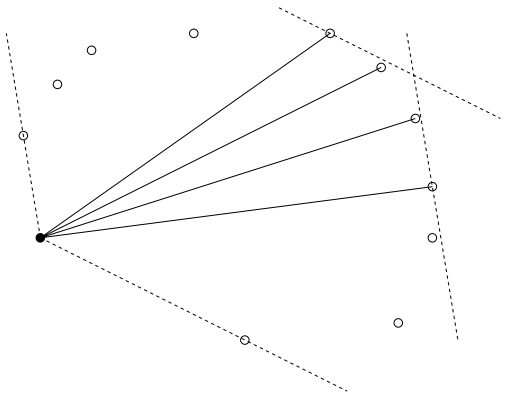

Figure 7. Rotating caliper neighbors of new point can be found from tangent lines parallel to convex hull edges.

on two facts which both hold for the rotating caliper graph. First, the only edges added in an insertion are adjacent to the inserted point. And second, the number of edges removed in an insertion is equal to the number added plus the difference in complexity in the convex hull, which follows for the rotating caliper graph from the fact that for $n \geq 3$, the number of edges is equal to the number of convex hull vertices. □

Lemma 14. *We can maintain the rotating caliper graph in expected time $O(\log n)$ per update.*

Proof: As in the previous section, we keep a binary search tree representing the order of vertices on the convex hull.

When a new point v is added to the convex hull perimeter, we examine the slopes of its two incident edges, and find for each slope the point opposite v on the convex hull having a tangent of the same slope. The rotating caliper graph neighbors of v can then be found as the interval of the convex hull boundary opposite v between these points of tangency (Figure 7). The two points of tangency can be found in $O(\log n)$ time by searching the balanced binary tree, after which the new neighbors of v can be found in constant time each. Each of these neighbors other than the endpoints of the interval lose the edges connecting them to any other vertices.

When a point is deleted, each of its neighbors in the rotating caliper graph may be reconnected to one of the two points on either side of the deleted point; we can test each such potential connection in constant time.

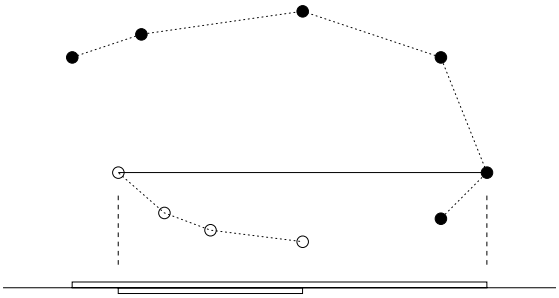

Figure 8. Farthest two points on two convex hull intervals form endpoints of interval projections.

The $O(\log n)$ bound follows from the fact that we perform a constant number of balanced tree operations per change to the rotating caliper graph. □

7 Connecting Two Farthest Neighbor Trees

We now describe a data structure to be used to find edges connecting disjoint trees of the farthest neighbor forest. Recall that each such edge connects two convex hull vertices, and that the convex hull vertices in each tree form two intervals in the cyclically ordered list of all convex hull vertices. The updates to the point set can be expected to be randomly distributed according to some signature in Mulmuley's expected-case model, but we can make no such assumption about the sequence of pairs of trees queried by the algorithm.

We use the following strategy. We partition an edge connecting the two trees into two sets: the *internal edges* connect a point interior to one of the two intervals of one tree with a point interior to an interval of the other tree, and the *external edges* connect one of the four interval endpoints of one tree with a point in the other tree. We show that if the longest edge connecting the trees is internal, it can be found using the rotating caliper graph. We then solve the remaining problem of finding the longest external edge using the technique of Lemma 4 for decomposable search problems.

First we deal with the internal edges.

Lemma 15. *Given two convex hull intervals X and Y, let $x \in X$ and $y \in Y$ be chosen to maximize the distance between x and y. Then x and y are both extrema within their own intervals with respect to their projected positions on some line ℓ.*

Proof: Take ℓ to be parallel to the line connecting x and y. Then if x and y were not extrema, we could replace them by other points and produce a farther pair. □

As depicted in Figure 8, x need not be an extremum of all points in both intervals; the lemma only claims that it is an extremum among points in its own interval. However, any point interior to its interval that is an extremum in the interval is also an extremum of the overall point set. Therefore we have the following result:

Corollary 1. *If the farthest pair (x, y) between two trees of the farthest neighbor forest forms an internal edge, it is an edge in the rotating caliper graph.*

Lemma 16. *Let (x, y) be the the longest internal edge connecting two given farthest neighbor forest trees that is also a rotating caliper graph edge (if such an edge exists). Then we can maintain a data structure in $O(\log n)$ expected time per point insertion or deletion that can find (x, y) in $O(\log n)$ time per query given the four convex hull intervals corresponding to the two trees.*

Proof: We solve the problem separately for each pair of intervals defined by the trees.

Recall that the rotating caliper graph connects points where lines of certain slopes are tangent to the convex hull. The rotating caliper graph edges internal to an interval of the convex hull boundary can be found by considering only those lines in a certain open interval of slopes, with the endpoints of the slope interval equal to the slopes of the end segments of the convex hull interval. We compute the two slope intervals corresponding to the two convex hull intervals, and intersect them to find a smaller interval of slopes which gives exactly the rotating caliper graph edges internal to the two intervals.

Thus we can find the longest internal rotating caliper edge by finding the longest edge coming from a certain interval of tangent slopes. If we maintain a balanced binary tree of rotating caliper graph edges, in order by the corresponding tangent slopes, and store at each internal node of the balanced tree the longest edge descending from that node, we can solve this problem in time $O(\log n)$ per query. The balanced binary tree requires an expected $O(\log n)$ time per update. □

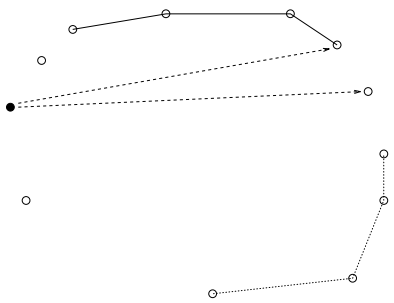

Figure 9. Included and excluded interval problem. Included interval I is marked with solid lines; excluded interval E is dotted. Either of the two dashed edges represents a correct solution.

We next deal with the external edges. To find these, we use a data structure to solve the *point-tree farthest neighbor problem*: find the farthest neighbor of one point v on the convex hull boundary among the vertices of a given tree in the farthest neighbor forest. In fact we find the neighbors of v separately among the two intervals determined by that tree. However our algorithm does not seem to generalize to finding the farthest neighbor of v in an arbitrary interval of the convex hull boundary.

As a subroutine for the point-tree farthest neighbor problem, we solve yet another farthest neighbor problem: the following *included and excluded interval problem* (Figure 9). We are given a point v on the convex hull of the input set, and two intervals I and E of the convex hull perimeter. I, E, and v are mutually disjoint. We must find a farthest neighbor among a set of convex hull vertices that includes all vertices of I but excludes all vertices of E. Other convex hull vertices may be either included or excluded arbitrarily. Points that are not convex hull vertices must not be included.

We now show how to solve the point-tree farthest neighbor problem by using the included and excluded interval problem.

Lemma 17. *Let a point v and a tree T of the farthest neighbor forest be given. Removing the intervals for T partitions the remaining convex hull boundary into two intervals; let E denote the interval not containing v. Then the farthest neighbor of v among those points on the convex hull boundary with E excluded is a point of T.*

Proof: Let H be the points of the convex hull boundary, and let w be the farthest neighbor of v in $H - E$. Each point in T has as its farthest

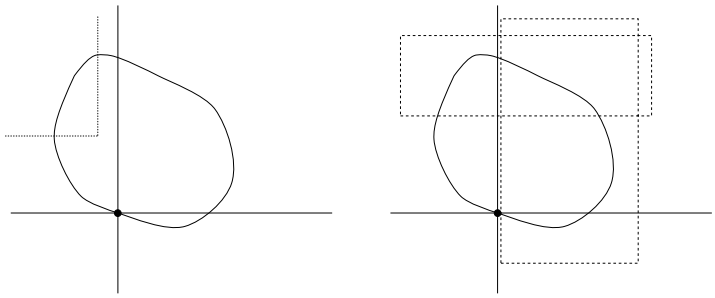

Figure 10. (a) In upper left quadrant, restrict included interval to subinterval with positive slope (inside the dotted lines). (b) Union of two ranges covers included interval and avoids excluded interval.

neighbor another point in T, which remains in $H - E$. If w were not in T, edge (v, w) would not cross any edge in T; in particular we could choose an edge (x, y) not crossed by (v, w). But then we could find two crossing edges (without loss of generality (v, x) and (w, y); by the quadrangle inequality, one of these edges would be longer than both (v, w) and (x, y), contradicting the assumption that these are edges in the farthest neighbor forest. □

Corollary 2. *We can solve the point-tree farthest neighbor problem by solving two included and excluded interval problems, with E as defined above and I successively the two intervals for T.*

Finally we show how to solve the included and excluded interval problem.

Lemma 18. *We can solve the included and excluded interval query problem in expected time $O(\log^3 n)$ per update or query.*

Proof: We show that each query can be solved by combining at most two orthogonal halfspace range queries that find the farthest input point in the given range. (Note that in general such queries can return points that are not on the convex hull, which would violate the requirements of the problem, so we must prove that the vertices we find are on the convex hull.) By Lemmas 4 and 10, we can perform these queries in expected time $O(\log^3 n)$ using a data structure with $O(\log^3 n)$ expected update time.

We assume without loss of generality that the query intervals occur in clockwise cyclic order vEI. Since v is a convex hull vertex, if we consider v the origin of a cartesian coordinate system then the input set is entirely

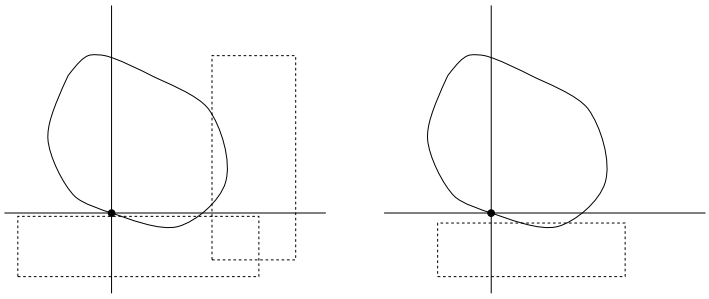

Figure 11. (a) Second case, I starts in upper right quadrant, and is covered by two ranges. (b) Third case, I is in bottom right quadrant, and can be covered by only one range.

contained in three quadrants of the plane, without loss of generality the upper left, upper right, and lower right.

First consider the case that I has some nonempty intersection I' with the upper left quadrant. We can assume that all convex hull boundary segments in I' have positive slope (Figure 10(a)). For if a segment uw occurs below the leftmost convex hull vertex, the portion of I' below that segment will be nearer v than u and w, and will never be the answer to the included and excluded segment problem. Similarly if a segment occurs after the topmost convex hull vertex, the portion of I' to the right but within the top left quadrant can be discarded. I' can be restricted to the portion with positive slope in $O(\log n)$ time by binary search. Let u be the point in I' with least y coordinate. We claim that E is entirely below the horizontal line through u. This follows since E is counterclockwise of I in the same quadrant and since u must be the point of I' closest to E in the cyclic order.

In this situation, we combine the results of one range query in the halfplane above a horizontal line through u, and a second range query in the halfplane right of a vertical line through v (Figure 10(b)). This finds the farthest point from v in a 270° wedge. These ranges both exclude E, which is entirely within the upper left quadrant. There are two portions of I that could be excluded from these ranges: there may possibly be such a portion below u, and another portion cut off by the corner of this wedge (when u has larger y-coordinate than the point where the convex hull crosses the y axis). But both portions are in the upper left quadrant and have negative slope, so do not contain the farthest point from v.

We now claim that the farthest point in the two ranges will be a convex

hull vertex. Note that the only other possibilities are the points where the lines delimiting these ranges cross the convex hull boundary, and that one or the other query could possibly return such a crossing point; we wish to show that in this case the other query returns a convex hull vertex farther from v. There are two possibilities. If the quarterplanar region of the input excluded from the two queries does not cross the convex hull boundary, the convex hull of all points in the two ranges is formed from the overall convex hull simply by cutting off line segment uv, and we know the farthest point from v in this smaller convex hull must itself be a convex hull vertex. Alternately, the convex hull boundary may be crossed by this quarterplane (a similar situation is depicted in Figure 11(a)). But in this situation both crossing points must be in the top left quadrant, and are nearer to v than is the vertex with maximum y coordinate.

In the second case, shown in Figure 11(a), I misses the upper left quadrant but intersects the upper right quadrant. This case can be treated similarly to the first case, by restricting I to segments with negative slope, and combining two range queries, one with a halfplane right of a vertical through the leftmost point in I', and another with a halfplane below the horizontal through v. Again, all of I is covered except for positive-slope portions in the top right quadrant, all of E is excluded, and the farthest point in the two query regions must be a vertex. In fact this case can be seen merely as a 90° rotation of the first case.

In the final case, shown in Figure 11(b), I is entirely contained in the lower right quadrant. As in the first case, we can restrict our attention to a portion of I having positive slope. We then perform a single halfspace farthest point range query, with the halfspace below a horizontal line through the uppermost point of I. This must be the point of I closest to E in the cyclic order, so E is excluded from the query. The query result is a convex hull vertex of the full input set since the range restriction doesn't change the portion of the convex hull boundary having positive slope. □

We summarize the results of this section.

Lemma 19. *We can compute the farthest pair of points connecting a pair of trees of the farthest neighbor forest in expected time $O(\log^3 n)$ per query or update.*

Proof: We find the longest internal rotating caliper edge by Lemma 16, and the longest external edge by using Corollary 2 and Lemma 18 once for

each of the eight interval boundary vertices. We then simply choose the longest among these $O(1)$ edges. □

8 Maintaining the Maximum Spanning Tree

Theorem 1. *The Euclidean maximum spanning tree can be maintained in expected time $O(\log^3 n)$ per update.*

Proof: We maintain the farthest neighbor forest in expected time $O(\log^2 n)$ per update as described in Lemma 11. We keep a list of the roots of the trees, and a priority queue of the edges connecting trees with adjacent intervals with pointers from the tree roots to the corresponding edges. For each of the $O(1)$ expected changes in the farthest neighbor forest, we find the corresponding tree root using the dynamic tree data structure of Sleator and Tarjan [41], remove the root of the old tree from the list of tree roots, and remove its edges from the priority queue. We then make a list of changed trees by again using the dynamic tree data structure and sorting the resulting list of tree roots to eliminate duplicates. For each changed tree, we recompute the two intervals described in Lemma 6, using the algorithm of Lemma 12. We determine the identities of the two adjacent trees in the cyclic order of Lemma 6 by looking up the points adjacent to the interval boundaries using again the dynamic tree data structure. We find the intervals for those trees (this can either be information stored with the tree roots, or recomputed as needed). We compute the edges connecting the changed tree with its two adjacent trees, using the interval query data structure described in the previous section, and add these edges to the priority queue.

We can now make a list of all edges removed from the tree (edges no longer in the farthest neighbor forest as well as edges connecting changed trees in the forest and the new smallest edge in the priority queue) as well as another list of newly added edges (edges added to the farthest neighbor forest, new edges connecting trees in the forest, and the old smallest edge in the priority queue). By sorting these lists together we can resolve conflicts occurring when an edge appears in both lists, and generate a list of all changes in the maximum spanning tree. □

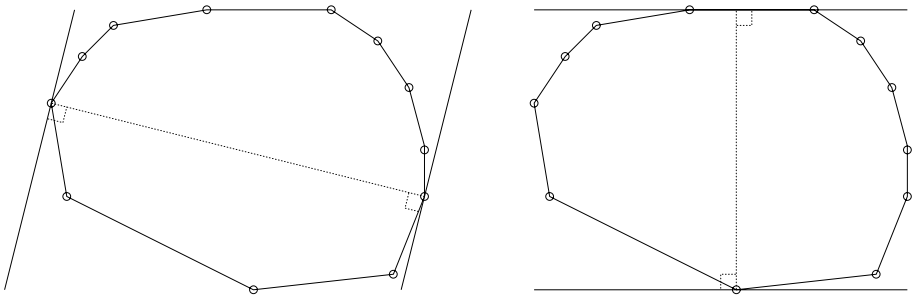

Figure 12. Diameter and width are determined by pairs of parallel tangent lines.

9 Width and Diameter

We now show how to maintain the solutions to several other geometric problems using the rotating caliper graph.

Recall that the *diameter* of a point set, the longest distance between any two points, is also the longest distance between any pair of parallel lines tangent to the convex hull (Figure 12(a)). Further, these lines must be tangent at vertices of the hull, rather than supported by edges. The *rotating caliper algorithm* for computing the diameter considers starting with any pair of parallel tangent lines, and rotating their slopes continuously through a circle maintaining a point of tangency to the convex hull at all times. As the slope of the lines changes, the tangent points move monotonically one vertex at a time around the convex hull perimeter, and one can find each successive event in constant time from the previous points of tangency. Thus given the convex hull of a point set, we can in linear time find its rotating caliper graph, from which the diameter can be found simply as the longest edge in the graph.

The *width* of a point set is similarly defined as the minimum distance between two parallel lines tangent to the convex hull (Figure 12(b)). At least one of the two lines must be supported by a convex hull edge. The width can then be computed by a similar rotating caliper process that keeps track of both the vertices and edges met by tangent pairs of lines.

Theorem 2. *We can maintain the width and diameter of a point set in expected time $O(\log n)$ per update.*

Proof: For the diameter, we simply maintain a priority queue of the lengths of all edges in the rotating caliper graph; the diameter can then be found as the maximum length in this priority queue.

For the width, first note that if a tangent line supports an edge xy on the convex hull perimeter then the point z of tangency for a parallel tangent line is exactly that convex hull vertex for which both xz and yz are edges in the rotating caliper graph. So for each pair of edges in the rotating caliper graph that are adjacent in terms of the slope ordering given by the rotating caliper algorithm, we maintain in a priority queue the distance between the common endpoint of the edges and the convex hull perimeter edge connecting the other endpoints of the edges. Each edge in the rotating caliper graph is associated with two such distances, so each graph update causes $O(1)$ changes in the set of distances. The width can be found by selecting the smallest of these distances in the priority queue. □

A similar technique can be used to maintain the minimum area or perimeter rectangle (not necessarily aligned with the coordinate axes) that encloses the point set. We maintain a hypergraph related to the rotating caliper graph, consisting of the four-tuples of vertices that can be formed by four lines tangent to the convex hull and at 90° angles to each other. There are exactly n such tuples, and as before the collection of tuples incurs an expected $O(1)$ changes per point insertion or deletion. When we insert a point, we can find the points involved in tuples with it by using binary search to find three intervals in the convex hull, after which we can find the actual tuples by performing a rotating caliper algorithm restricted to these intervals. When we delete a point, all the tuples associated with it are replaced by tuples involving the adjacent two points on the convex hull. The two minimization problems above are both typically solved by quadruples of tangent lines one of which is supported by a convex hull edge; these can be found as in our width algorithm by considering adjacent pairs of tuples in the slope ordering. This gives the following result.

Theorem 3. *We can maintain the minimum area or perimeter enclosing rectangle of a point set (having sides not necessarily aligned with the coordinate axes) in expected time $O(\log n)$ per update.*

10 Conclusions

We have seen how to maintain the maximum spanning tree of a planar point set in the expected case. Our algorithm is based on that of Monma et al. [28] and uses as subroutines algorithms for maintaining the farthest neighbor forest and for answering farthest pair queries between intervals on

the convex hull perimeter. We also solved the problem of maintaining the width and other related quantities in expected time $O(\log n)$ per update.

However some open problems remain. We mentioned earlier the problem of performing faster point location in average-case Voronoi diagrams. Also, can we say anything about higher dimensional maximum spanning trees? Our present algorithm depends strongly on planar properties such as the cyclic ordering of convex hull vertices. The higher dimensional problem can be solved by repeatedly merging pairs of trees using a bichromatic farthest pair algorithm [3, 28] but it is unclear whether such an algorithm could be dynamized efficiently.

References

[1] P. K. Agarwal, H. Edelsbrunner, O. Schwarzkopf, and E. Welzl. Euclidean minimum spanning trees and bichromatic closest pairs. *Discrete Comput. Geom.*, 6:407–422, 1991. See also *6th Symp. Comp. Geom.*, 1990, pp. 203–210.

[2] P. K. Agarwal, D. Eppstein, and J. Matoušek. Dynamic algorithms for half-space reporting, proximity problems, and geometric minimum spanning trees. In *Proc. 33rd IEEE Symp. Foundations of Computer Science*, pages 80–89, 1992.

[3] P. K. Agarwal, J. Matoušek, and S. Suri. Farthest neighbors, maximum spanning trees, and related problems in higher dimensions. *Comput. Geom.: Theory & Appl.*, 4:189–201, 1992. See also *2nd Worksh. Algorithms and Data Structures*, 1991, pp. 105–116.

[4] P. K. Agarwal and M. Sharir. Off-line dynamic maintenance of the width of a planar point set. *Comput. Geom.: Theory & Appl.*, 1:65–78, 1991.

[5] D. Alberts and M. Rauch Henzinger. Average case analysis of dynamic graph algorithms. In *Proc. 6th ACM-SIAM Symp. Discrete Algorithms*, pages 312–321, 1995.

[6] N. Amenta. Helly theorems and generalized linear programming. *Discrete Comput. Geom.*, 12:241–261, 1994. See also *9th Symp. Comp. Geom.*, 1993, pp. 63–72.

[7] T. Asano, B. Bhattacharya, M. Keil, and F. Yao. Clustering algorithms based on minimum and maximum spanning trees. In *Proc. 4th ACM Symp. Computational Geometry*, pages 252–257, 1988.

[8] H. Baumgarten, H. Jung, and K. Mehlhorn. Dynamic point location in general subdivisions. *J. Algorithms*, 17:342–380, 1994. See also *3rd Symp. Discrete Algorithms*, 1992, pp. 250–258.

[9] J. L. Bentley and J. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms*, 1:301–358, 1980.

[10] K. Q. Brown. Voronoi diagrams from convex hulls. *Inform. Proc. Lett.*, 9:223–226, 1979.

[11] G. Cattaneo and G. F. Italiano. Average case and empirical study of sparsification. Manuscript, 1995.

[12] S. W. Cheng and R. Janardan. New results on dynamic planar point location. *SIAM J. Comput.*, 21:972–999, 1992. See also *31st Symp. Found. Comp. Sci.*, 1990, pp. 96–105.

[13] Y.-J. Chiang and R. Tamassia. Dynamization of the trapezoid method for planar point location. *Int. J. Comput. Geom. & Appl.*, 2:311–333, 1992. See also *7th Symp. Comp. Geom.*, 1991, pp. 61–70.

[14] K. L. Clarkson and P. W. Shor. Algorithms for diametral pairs and convex hulls that are optimal, randomized, and incremental. In *Proc. 4th ACM Symp. Computational Geometry*, pages 12–17, 1988.

[15] O. Devillers, S. Meiser, and M. Teillaud. Fully dynamic Delaunay triangulation in logarithmic expected time per operation. *Comput. Geom.: Theory & Appl.*, 2:55–80, 1992. See also *2nd Worksh. Algorithms and Data Structures*, 1991, pp. 42–53.

[16] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. In *Proc. 8th ACM Symp. Computational Geometry*, pages 43–52, 1992.

[17] D. Eppstein. Dynamic three-dimensional linear programming. *ORSA J. Comput.*, 4:360–368, 1992. See also *32nd Symp. Found. Comp. Sci.*, 1991, pp. 488–494.

[18] D. Eppstein. Offline algorithms for dynamic minimum spanning tree problems. *J. Algorithms*, 17:237–250, 1994. See also *2nd Worksh. Algorithms and Data Structures*, 1991, pp. 392–399.

[19] D. Eppstein. Dynamic euclidean minimum spanning trees and extrema of binary functions. *Discrete Comput. Geom.*, 13:237–250, 1995.

[20] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. Sparsification – A technique for speeding up dynamic graph algorithms. In *Proc. 33rd IEEE Symp. Foundations of Computer Science*, pages 60–69, 1992.

[21] D. Eppstein, Z. Galil, and G.F. Italiano. Improved sparsification. Technical Report 93-20, Department of Information and Computer Science, University of California, Irvine, 1993.

[22] D. Eppstein, G. F. Italiano, R. Tamassia, R. E. Tarjan, J. Westbrook, and M. Yung. Maintenance of a minimum spanning forest in a dynamic plane graph. *J. Algorithms*, 13:33–54, 1992. See also *1st Symp. Discrete Algorithms*, 1990, pp. 1-11.

[23] G. N. Frederickson. Data structures for on-line updating of minimum spanning trees. *SIAM J. Comput.*, 14:781–798, 1985.

[24] M. T. Goodrich and R. Tamassia. Dynamic trees and dynamic point location. In *Proc. 23rd ACM Symp. Theory of Computing*, pages 523–533, 1991.

[25] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7:381–413, 1992. See also *17th Int. Coll. on Automata, Languages and Programming*, 1990, pp. 414–431.

[26] R. Janardan. On maintaining the width and diameter of a planar point-set online. *Int. J. Comput. Geom. & Appl.*, 3:331–344, 1993. See also *2nd Int. Symp. Algorithms*, 1991, pp. 137–149.

[27] K. Mehlhorn, S. Meiser, and C. O'Dunlaing. On the construction of abstract Voronoi diagrams. *Discrete Comput. Geom.*, 6:211–224, 1991. See also *Symp. Theor. Aspects of Comp. Sci.*, 1990, pp. 227–239.

[28] C. Monma, M. Paterson, S. Suri, and F. Yao. Computing Euclidean maximum spanning trees. *Algorithmica*, 5:407–419, 1990. See also *4th Symp. Comp. Geom.*, 1988, pp. 239–249.

[29] K. Mulmuley. A fast planar partition algorithm, I. *J. Symbolic Computation*, 10:253–280, 1990. See also *29th Symp. Found. Comp. Sci.*, 1988, pp. 580–589.

[30] K. Mulmuley. Randomized multidimensional search trees: dynamic sampling. In *Proc. 7th ACM Symp. Computational Geometry*, pages 121–131, 1991.

[31] K. Mulmuley. Randomized multidimensional search trees: lazy balancing and dynamic shuffling. In *Proc. 32nd IEEE Symp. Foundations of Computer Science*, pages 180–196, 1991.

[32] K. Mulmuley. *Computational Geometry, an Introduction through Randomized Algorithms*. Prentice Hall, 1993.

[33] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *J. Comp. Syst. Sci.*, 23:224–233, 1981.

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

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

[36] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Comm. Assoc. Comput. Mach.*, 33:668–676, 1990.

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

[38] O. Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *Proc. 32nd IEEE Symp. Foundations of Computer Science*, pages 197–206, 1991.

[39] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete Comput. Geom.*, 6:423–434, 1991. See also *6th Symp. Comp. Geom.*, 1990, pp. 211–215.

[40] M. I. Shamos and D. Hoey. Closest-point problems. In *Proc. 16th IEEE Symp. Foundations of Computer Science*, pages 151–162, 1975.

[41] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *J. Comp. Syst. Sci.*, 24:362–381, 1983.

[42] A. C. Yao. On constructing minimum spanning trees in k-dimensional space and related problems. *SIAM J. Comput.*, 11:721–736, 1982.