Average Case Analysis of Dynamic Geometric Optimization

David Eppstein[*]

Abstract

We maintain the maximum spanning tree of a planar point set, as points are inserted or deleted, in $O(\log^3 n)$ time per update in Mulmuley's expected-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 time $O(\log^2 n)$ per update and the latter in time $O(\log n)$ per update. We also use the rotating caliper graph to maintain the diameter, width, and minimum enclosing rectangle in time $O(\log n)$ per update.

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 [22, 23, 24] and Schwarzkopf [29] 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 recent text on randomized geometric algorithms [24] 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 [8]; we examine methods for performing this sort of reduction in a way that preserves the average case behavior of the update sequence.

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. The dynamic planar diameter problem can be reduced (with logarithmic overhead) to finding farthest neighbors to 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. Thus in the worst case we have time $O(n^\epsilon)$ per update [2]. But the diameter is also the longest edge in the farthest point De-

[*]Department of Information and Computer Science, University of California, Irvine, CA 92717. Work supported in part by NSF grant CCR-9258355.

launay triangulation, which can be maintained in $O(\log n)$ average time per update using techniques of Mulmuley [23]. We will see that an even simpler algorithm can also maintain the diameter in $O(\log n)$ time.

Linear programming. In a previous paper [13], 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. But in the average case, if the objective function is fixed, Seidel's algorithm [30] can easily be adapted to provide a constant time bound per update. The same result applies to the many related nonlinear optimization problems which can be solved by the same algorithm [5].

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. Using clustering techniques for graph minimum spanning trees [14, 15, 18], we were able to solve the minimum spanning tree problem in time $O(n^{1/2} \log^2 n)$ per update [2]. In the average case, the minimum spanning tree can be maintained much more easily in time $O(\log n)$ per update by combining a dynamic Delaunay triangulation algorithm [23] with a dynamic planar graph minimum spanning tree algorithm [16].

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 [6]. 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 [31] the maximum spanning tree is not contained in the farthest point Delaunay triangulation [21]. 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 expected case, the maximum spanning tree changes by $O(1)$ edges per update, so efficient algorithms may be possible. Monma *et al.* [21] 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 width appears to be much more difficult than that of maintaining 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 [20] maintains an approximation to the width in time $O(\log^2 n)$ per update. But neither 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 expected-case model, in 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 (non-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 to maintain the width.

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 [17] are found by computing nearest neighbors in certain directions from each point [33]; 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 [31]. In our maximum spanning tree algorithm, as in that of Monma *et al.* [21], no such strategy is used. Instead, the maximum spanning tree is computed directly from the geometry, with no need to use a graph minimum spanning tree algorithm.

2 The Model of Expected Case Input

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 time averaged over all possible update sequences determined by the signature, and over all random choices made by the algorithm. The expected time on inputs of size n is 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 expected case for randomized incremental algorithms is generated by signatures containing only the "+" character.

Lemma 2.1. *The expected number of edges that change per update in the maximum spanning tree, farthest point Delaunay triangulation, or farthest neighbor forest is $O(1)$.*

Proof. We first bound the change per insertion. Consider 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. Each of the three graphs has $O(n)$ edges. Each 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 $O(j) \cdot 2/j = O(1)$. The number of existing edges removed in the insertion is at most proportional to the number of edges added, and can possible be even smaller if the convex hull becomes less complex as a result of the insertion. Thus the total 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 also need the following special case of more general convex hull bounds [12].

Lemma 2.2. *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 apexes. This is true with probability $2/n$ so the expected number of vertices removed is no more than two. □

A similar bound on the change in a fifth geometric graph, the *rotating caliper graph*, is proved later in Lemma 5.1.

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* [8] 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 one-dimensional *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 partition the problems into subproblems in a way which destroys the expected case behavior of the subproblem updates. We add average case orthogonal range search capabilities to decomposable search problems by a generalization of a technique which Mulmuley [24] used to answer some interval range line segment intersection queries.

Lemma 2.3. *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, or better $O(T(n))$ if $T(n) = O(n)$ and $T(n) = \Omega(n^{\epsilon})$ for some fixed ϵ.*

Proof. We partition the problem into a number of subproblems using a *skip list* [28], 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 each such interval we provide a subproblem for all input points contained in the interval.

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 $2^i T(2^i)$, but the probability of having to do so at level i is 2^{-i} so the expected total work in rebuilding is $\sum_{i=1}^{\log n} T(2^i) = O(T(n) \log n)$.

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 $T(2^i)$. Again the total expected time is $O(T(n) \log n)$.

Deletions are similar to insertions, and the time for deletions can be shown to be $O(T(n) \log n)$ using a similar argument. The improved bounds for $T(n) = \Omega(n^{\epsilon})$ follow from the observation that in that case $\sum_{i=1}^{\log n} T(2^i) = O(T(n))$. $\square$

3 Maximum Spanning Tree Analysis

We now examine the edges that can occur in the maximum spanning tree. Most of the material in this section is due to Monma *et al.* [21], 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 3.1. *The farthest neighbor forest is a subgraph of the maximum spanning tree.*

Lemma 3.2. *Let each tree of the farthest neighbor forest be two-colored. 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 the intervals adjacent to any such interval come from adjacent trees in the ordering.*

Lemma 3.3. *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 3.2.*

Summarizing the above Lemmas, we have the following algorithm outline, again due to Monma et al. [21].

Lemma 3.4. *The maximum spanning tree can be constructed by computing the farthest neighbor forest, determining the cyclic ordering of Lemma 3.2, finding the longest edge between each adjacent pair of trees in the cyclic ordering, and removing the shortest such edge.*

Monma *et al.* 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 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 2.1, 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 4.1. *We can maintain the farthest point Voronoi diagram in expected time $O(\log n)$ per update.*

Proof. Since the farthest point Delaunay triangulation is the projection of a three-dimensional convex hull [9], we can maintain it using Mulmuley's dynamic convex hull algorithm [23]. 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. 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 the following point location data structure:

Lemma 4.2. *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 [7, 10, 11, 19, 27]. By Lemma 2.1, the amount of change per update is $O(1)$. □

We note that the $O(\log n)$ time dynamic Voronoi diagram algorithm maintains a point location data structure in the dual triangulation, however this does not provide the point location in the Voronoi diagram itself that is supplied by Lemma 4.2. It is open [24] whether Voronoi diagram point location can be performed more quickly than the above $O(\log^2 n)$ bound on average.

Lemma 4.3. *We can maintain the farthest neighbor forest of a dynamically changing input in expected time $O(\log^2 n)$ per update.*

Proof. We have seen how to maintain the farthest point Voronoi diagram, and a point location data structure for that diagram. We are then left with the problem of updating the 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.

Two types of changes may occur. 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. 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 [25].

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 3.2 shows that each tree in the farthest neighbor 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 the next section 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 4.4. *We can determine the endpoints of the two intervals described in Lemma 3.2, for any tree in the farthest neighbor forest specified by any vertex in that* *tree, in expected time $O(\log^2 n)$ per query and $O(\log n)$ per update.*

Proof. We maintain a balanced binary tree representation of the convex hull in $O(\log n)$ expected time per update. We find a single point in one interval by looking at the root of the given tree, and a point not in the interval by taking the other root of the same tree. We can then find the interval by binary search, using a dynamic tree data structure [32] to test whether each searched point is of the given color in the given tree. □

5 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 rotating caliper algorithm for computing width and diameter [26].

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. The rotating caliper algorithm considers the sequence of convex hull vertices tangent to lines of different slopes. As the slope varies, the tangent points move monotonically around the convex hull perimeter. The diameter can be computed by comparing lengths of segments formed by a tangent on one side of the convex hull and a tangent of the same slope on the other side. The *width*, or shortest distance between two parallel tangent lines, can be computed by a similar process that also considers lines tangent to convex hull edges.

The *rotating caliper graph* is the collection of tangent point pairs considered by the rotating caliper algorithm. Equivalently edge xy is in the rotating caliper graph exactly when all input points lie between two parallel lines through x and y. Like the farthest point Delaunay triangulation the rotating caliper graph only connects convex hull vertices.

Lemma 5.1. *The expected change per update in the rotating caliper graph is $O(1)$.*

Proof. The effect of an insertion is exactly the reverse of the effect of a deletion, so by symmetry we need only discuss insertions. The number of edges in the rotating caliper graph is equal to the cardinality of the convex hull, so by Lemma 2.2 the expected change in the number of edges is $O(1)$. Thus the expected total change in the graph is in expectation proportional to the number of new edges added to the graph. Each such new edge must be adjacent to the newly added point, and as in Lemma 2.1 the expected degree of that new point is $O(1)$. □

Lemma 5.2. *We can maintain the rotating caliper graph in expected time $O(\log n)$ per update.*

Proof. We keep a search tree of convex hull vertices. When a new point is added to the convex hull perimeter, it forms a certain angle with its two neighbors. Its new neighbors are then the k points on the other side of the convex hull with angles of tangency in the same range, which can be found in $O(k + \log n)$ time by binary search. When a point is deleted, each neighbor is reconnected to either side of the deleted point. □

Corollary 5.1. *We can maintain the width and diameter of a point set in expected time $O(\log n)$ per update.*

Proof. The diameter is the longest edge in the rotating caliper graph. For the width, 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 adjacent pair of edges in the rotating caliper graph we maintain 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 among these distances using a priority queue. □

A similar technique using a hypergraph defined by rotating calipers of four lines at right angles to each other can be used to maintain the minimum area or perimeter rectangle (not necessarily aligned with the coordinate axes) that encloses the point set.

6 Diameter of Convex Hull Intervals

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.

We solve the following abstract generalization of the problem. We are given a dynamically changing point set. We wish to answer queries of the form: given two intervals on the convex hull of the point set (specified by their endpoints) find the longest edge from one interval to the other. 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 queries.

With such a data structure, we can answer our original problem by determining the two intervals for each tree and pairing them up in two queries to the data structure. As a subroutine for these interval farthest pair problems, we would need a subroutine that could answer *interval farthest neighbor queries* (this is simply the special case of the two interval farthest pair problem in which one interval is a single point). This problem can be solved in time $O(n^\epsilon)$ by combining a weight-balanced tree of the convex hull vertices with a farthest neighbor data structure of Agarwal and Matoušek [2]. However such a bound is too large for our algorithm.

Instead we show certain properties of the intervals determined by the farthest neighbor forest, that allow us to answer the desired interval farthest pair problem using a faster data subroutine for the simpler problem of orthogonal halfspace farthest neighbor range searching.

Lemma 6.1. *Let v be a convex hull vertex, in a given tree T of the farthest neighbor forest. Then the farthest neighbor of v outside T is in a tree adjacent to T in the cyclic order of Lemma 3.2.*

Proof. Remove all points from T but v. The only change to the farthest neighbor forest will be that v is added as a leaf to some other tree. By Lemma 3.2, it must be added to an adjacent tree. □

In light of this lemma, we can solve our desired interval queries using a dynamic algorithm for the following *included and excluded interval query problem*: 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.

Lemma 6.2. *We can solve the dynamic 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. By Lemmas 2.3 and 4.2, we can perform these queries in update and query time $O(\log^3 n)$.

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 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. 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.

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. These ranges both exclude E, which is entirely within the upper left quadrant. The only portion of I that can be excluded from both ranges is in the upper left quadrant and has negative slope, so cannot contain the desired answer. We claim that the farthest point in the two ranges will be a convex hull vertex (even though this is not necessarily true just of the first range). There are two possibilities. First, 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. Second, if the convex hull boundary is crossed, insert for sake of argument two artificial points at the crossings. Then with these new points, the two range queries cover disjoint point sets with convex hulls exactly equal to the intersection of the original convex hull with the range query halfplanes, so each returns a convex hull vertex. The uppermost convex hull vertex is farther than either artificial point, so neither would be returned if it were part of the input and instead a true convex hull vertex would result. But then that vertex must also be the result of the actual queries that are performed.

In the second case, I misses the upper left quadrant but intersects the upper right quadrant. This case can be treated exactly the same as 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.

In the final case, 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 next need the following fact which justifies the correctness of the "rotating caliper" algorithm.

Lemma 6.3. *Let (x, y) be the farthest pair of points drawn from two convex hull intervals. 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 xy. Then if x and y were not extrema, we could replace them by other points and produce a farther pair. □

Note that e.g. x may not necessarily be an extremum among 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 the interval that is an extremum in the interval is also an extremum of the overall point set.

Lemma 6.4. *With the aid of the included and excluded interval data structure described above, we can compute the farthest pair of points in a pair of farthest neighbor forest intervals in expected time $O(\log^3 n)$.*

Proof. We conceptually rotate line ℓ through 360° of motion, tracking the pairs of points that arise as extrema on ℓ. As ℓ rotates, the extrema in each interval pass monotonically along the perimeter of the convex hull, including each convex hull vertex in turn. The pairs involved are thus edges in the rotating caliper graph defined in the previous section, except for those pairs involving one or two endpoints of intervals. We keep a balanced binary tree of all edges in the rotating caliper graph, sorted by slope; for each node in the tree we track the longest rotating caliper graph edge among all descendants of that node. With this structure we can find the longest edge connecting internal vertices of the two intervals, in time $O(\log n)$. The longest edge involving interval boundary vertices can be found with the data structure of Lemma 6.2. □

7 The Maximum Spanning Tree

Theorem 7.1. *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 4.3. 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 [32], 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 3.2, using the algorithm of Lemma 4.4. We determine the identities of the two adjacent trees in the cyclic order of Lemma 3.2 by looking up the points adjancent 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. □

8 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. [21] 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 in expected time $O(\log n)$ per update.

However some open problems remain. In particular, can we say anything about higher dimensional minimum or maximum spanning trees? Our present algorithm for maximum spanning trees depends strongly on planar properties such as the cyclic ordering of convex hull vertices, and the algorithm for minimum spanning trees depends on the planarity and sparsity of the Delaunay triangulation. The higher dimensional static problem can be solved by repeatedly merging pairs of trees using a bichromatic farthest pair algorithm [3, 21] but it is unclear whether such an algorithm could be dynamized efficiently. It is also open whether width and diameter can be maintained quickly in higher dimensions. Another open problem in average case geometric optimization is whether we can handle dynamic linear programming queries with variable objective functions.

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] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] David Eppstein. Offline algorithms for dynamic minimum spanning tree problems. In *Proc. 2nd Worksh. Algorithms and Data Structures*, volume 519 of *Lecture Notes in Computer Science*, pages 392–399. Springer-Verlag, August 1991.

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

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

[20] 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.

[21] 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.

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

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

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

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

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

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

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

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

[30] 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.

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

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

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