Sorting Helps for Voronoi Diagrams

L. Paul Chew*
Steve Fortune**

TR 93-1347
May 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*Author's address: Department of Computer Science, Cornell University, Ithaca, NY 14853. This work was supported by the Advanced Research Projects Agency of the Department of Defense under ONR Contract N00014-92-J-1989, and by ONR Contract N00014-92-J-1839, NSF Contract IRI-9006137, and AFOSR Contract AFOSR-91-0328.
**Author's address: AT&T Bell Laboratories, Murray Hill, NJ 07974.

Sorting Helps for Voronoi Diagrams

L. Paul Chew * Steve Fortune †

May 11, 1993

Abstract

It is well known that, using standard models of computation, it requires $\Omega(n \log n)$ time to build a Voronoi diagram for n data points. This follows from the fact that a Voronoi diagram algorithm can be used to sort. But if the data points are sorted before we start, can the Voronoi diagram be built any faster? We show that for certain interesting, although nonstandard types of Voronoi diagrams, sorting helps. These nonstandard types Voronoi diagrams use a convex distance function instead of the standard Euclidean distance. A convex distance function exists for any convex shape, but the distance functions based on polygons (especially triangles) lead to particularly efficient Voronoi diagram algorithms - fast algorithms using simple data structures. Specifically, a Voronoi diagram using a convex distance function based on a triangle can be built in $O(n \log \log n)$ time after initially sorting the n data points twice. Convex distance functions based on other polygons require more initial sorting.

1 Introduction

We examine the relationship between sorting and Voronoi diagrams. It is well known that using standard models of computation (e.g., the algebraic decision tree model used in Preparata and Shamos [PS85]), it requires $\Omega(n \log n)$ time to build a Voronoi diagram (n is the number of data points). This follows from the fact that a Voronoi diagram algorithm can be used to sort. But if the data points are sorted before we start, can the Voronoi diagram be built any faster? In this paper, we present an affirmative answer to this question for certain interesting, although nonstandard, types of Voronoi diagrams. These nonstandard Voronoi diagrams use a convex distance function instead of the standard Euclidean distance.

We show that for certain convex distance functions, the shape of a Voronoi diagram using that distance function is so constrained that the Voronoi diagram can be built with surprising efficiency. In fact, for one such distance function (see Section 4), after presorting by x-coordinate and y-coordinate, the Voronoi diagram can be built in time $O(n \log \log n)$,

*Author's address: Department of Computer Science, Cornell University, Ithaca, NY 14853. This work was supported by the Advanced Research Projects Agency of the Department of Defense under ONR Contract N00014-92-J-1989, and by ONR Contract N00014-92-J-1839, NSF Contract IRI-9006137, and AFOSR Contract AFOSR-91-0328.

†Author's address: AT&T Bell Laboratories, Murray Hill, NJ 07974

where n is the number of data points. In Section 5, we show how this result can be extended to produce efficient Voronoi diagram algorithms for any polygon-based convex distance function.

These results have a number of applications. Some of the Voronoi diagrams that we can build efficiently are immediately useful (e.g., the L_1 metric Voronoi diagram [Lee80, LW80]). In other cases, although our efficiently-built Voronoi diagrams are nonstandard, they are close enough to standard Voronoi diagrams to be valuable. In other words, for some problems we can exchange accuracy for running time, building a nonstandard Voronoi diagram quickly to determine an approximate answer to the given problem. This approximate answer can often be shown to be within a small factor of the optimal answer. A number of such applications are described in Section 6.

Bern, Karloff, Raghavan, and Schieber [BKRS89] have used some of our results and applications (originally available in 1988 via conference presentation [CF88] and via a rough-draft manuscript), combining their work on fast approximate geometric sorting with our fast Delaunay triangulation algorithms to produce fast approximation algorithms for a number of geometric applications.

The algorithms presented here are most likely of theoretical rather than practical interest, since the time improvement (going from $O(n \log n)$ to $O(n \log\log n)$) is unlikely to be of great practical benefit except for very large data sets that happen to be already sorted. One conclusion that can be drawn from our work that is of particular theoretical interest is that the existing lower bounds on building convex-polygon-based Voronoi diagrams are due entirely to lower bounds on sorting. It is an open problem as to whether this is also true for the standard Voronoi diagram.

2 Background

In this section, we present some of the definitions and properties of convex distance functions, Voronoi diagrams, and Delaunay triangulations that are needed for the remainder of the paper. The text by Preparata and Shamos [PS85] is a good source for additional relevant background material.

Convex Distance Functions. Convex distance functions, also called Minkowski distance functions, were first used by Minkowski in 1911. For such a function, distance can be defined in terms of a unit *circle*. Circle, here, is printed in italics because this circle can be any oriented convex shape. To find the distance from point p to point q, we center the unit circle at point p, without altering the circle's orientation, and expand (or contract) the circle until its boundary intersects q. The distance from p to q is defined to be the factor by which the circle changed. If the circle is a true unit circle, with its center in the usual place, then we get the usual Euclidean distance. If the circle is an arbitrary convex shape with a center anywhere in its interior then we get a convex distance function. Note that, although the Triangle Inequality holds, the distance defined in this manner is not necessarily a metric, since the Symmetry Property (the distance from p to q is the same as the distance from q to p) holds only if the given circle is symmetric about its center.

Voronoi Diagrams. Let S be a set of n points in the plane (called *data points* or *sources*). The Voronoi diagram for S divides the plane into regions, one region for each point in S, such that for each region R and its corresponding point p, every point within R is closer to p than to any other point of S. The boundaries of these regions form a planar graph. The Voronoi diagram and its dual, the Delaunay triangulation, have been found to be among the most useful data structures in computational geometry. (See [PS85] for a large number of Voronoi diagram applications.)

Delaunay Triangulations. Let S be a set of n points in the plane. The Delaunay triangulation of S is the straight-line dual of the Voronoi diagram for S; that is, we connect a pair of points in S if they share a Voronoi boundary. This standard definition of Delaunay triangulation can lead to some minor problems if there are 4 points in S that are on the same circle; this is often resolved by outlawing such points. (Note that *circle*, here refers to the distance defining convex shape – see below.) An equivalent definition of Delaunay triangulation can be developed using the Empty Circle Property. A planar graph G on S has the Empty Circle Property if, for any edge of G, there exists a circumscribed circle that contains no points of S in its interior. Using this property, a Delaunay triangulation of S is defined as a planar graph G that has the Empty Circle Property and is maximal. (Maximal, here, means that any additional edge would make G nonplanar or violate the Empty Circle Property.) The equivalence of this definition and the standard Voronoi diagram definition is straightforward and is not presented here.

For any convex distance function, there is a corresponding version of Voronoi diagram and Delaunay triangulation. Such a Voronoi diagram is defined just like the standard (Euclidean) Voronoi diagram except the convex distance function is used to calculate distances. A useful intuition is to think of circular waves (i.e., waves in the shape of the distance defining circle) expanding from each data point; where the waves collide, we have a Voronoi boundary. Chew and Drysdale [CD85] have shown that, like the Euclidean Voronoi diagram, such a Voronoi diagram can be constructed in $O(n \log n)$ time where n is the number of data points. A version of Fortune's sweepline algorithm [For87] can also be used to construct convex-distance-function-based Voronoi diagrams in $O(n \log n)$ time. See [CD85, For85] or the survey by Aurenhammer [Aur91] for more information on convex distance functions and their relation to Voronoi diagrams and Delaunay triangulations.

There is one subtlety about the orientation of the distance-defining convex shape (the circle) that needs to be mentioned. A Delaunay-triangulation empty circle is always a reflection (about its center) of the distance-defining circle. Alternately, one can think of the Delaunay-triangulation empty circles as the distance-defining shape; in this case the circles used for the wave description of Voronoi diagrams are the reflections. In any case Voronoi-diagram wave-circles and Delaunay-triangulation empty-circles are reflections of each other. This subtlety can be ignored, of course, for a convex distance function that is a metric (i.e., a distance function for which symmetry holds).

3 The Center is Arbitrary

The results of this paper are based on the following observation: the Empty Circle Property is independent of the center chosen for our convex distance function. In other words, the Delaunay triangulation is unaffected by where we decide to put the center of our distance-defining convex shape. Note that, although the Delaunay triangulation remains unchanged, the shape of the Voronoi diagram changes as the center is moved.

It takes just linear time to go from a Voronoi diagram to a Delaunay triangulation and vice versa; thus, given a convex distance function (an oriented convex shape and its center) we have a nice way to compute a Voronoi diagram for the given distance function. We start by moving the center of our distance-defining convex shape to whatever position makes the computation easiest. Then, we compute the Voronoi diagram using the new center, and we use this Voronoi diagram to compute the Delaunay triangulation. The Delaunay triangulation is the same regardless of where the center is; thus, we can use it to quickly compute the Voronoi diagram based on the original convex distance function. In practice, we do not actually need to construct all of these diagrams and triangulations; any one of them can be used to efficiently compute one of the others whenever it is needed.

For our purposes, it is particularly useful to move the center of the given convex shape all the way to the boundary of the shape. In particular, for a convex polygon we want to move the center into a corner of the polygon. It is not immediately clear that this results in a sensible distance function. The problem is, that for such a distance function, whenever the distance from p to q is defined, the distance from q to p is undefined (i.e., infinite). The Empty Circle Property is unaffected by this difficulty, but it plays havoc with the idea of a Voronoi boundary as a subset of a bisector (bisector: the set of points equidistant from two data points). It also leads to Voronoi diagrams in which portions of the plane are not part of any Voronoi region. Fortunately, there are several equivalent ways of defining Voronoi diagrams that remain valid. Two such ways are (1) the Voronoi region for a data point p of S is the set of points in the plane that are closer to p (starting from p) than to any of other data points of S, and (2) simultaneously start a circular wave at each data point, where the waves collide we have Voronoi boundaries.

4 Triangle Distance Voronoi Diagrams

In this section, we use a convex distance function with properties that lead to an especially efficient Voronoi diagram algorithm. This convex distance function, called RT (right triangle) distance, is based on a right triangle with vertices at (0,0), (1,0), and (0,1), and with center at (0,0).

Given a set of n points in the plane, a nice, intuitive way to construct the RT Voronoi diagram uses the idea of expanding waves (see Figure 1). We start by placing a very small copy of the distance-defining triangle at each data point. We then create our waves by simultaneously expanding all of these triangles. The Voronoi boundaries are determined by where these waves collide. A completed RT Voronoi diagram appears in Figure 2. Note that one portion of the plane is not part of any data point's Voronoi region. We consider this region as belonging to a special data point sitting at $(-\infty, -\infty)$.

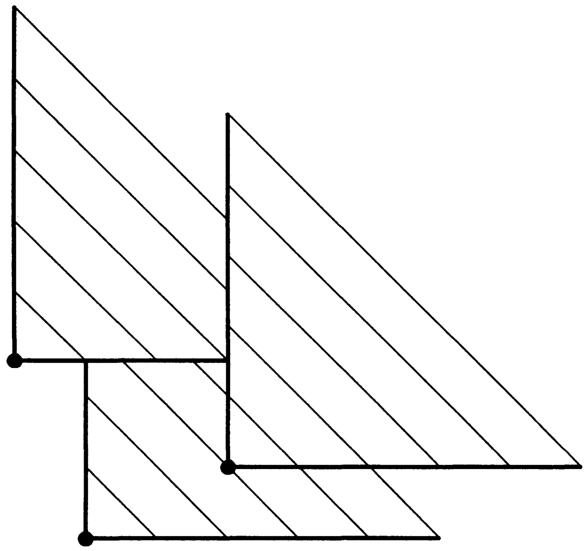

Figure 1: Expanding RT waves and the resulting Voronoi boundaries.

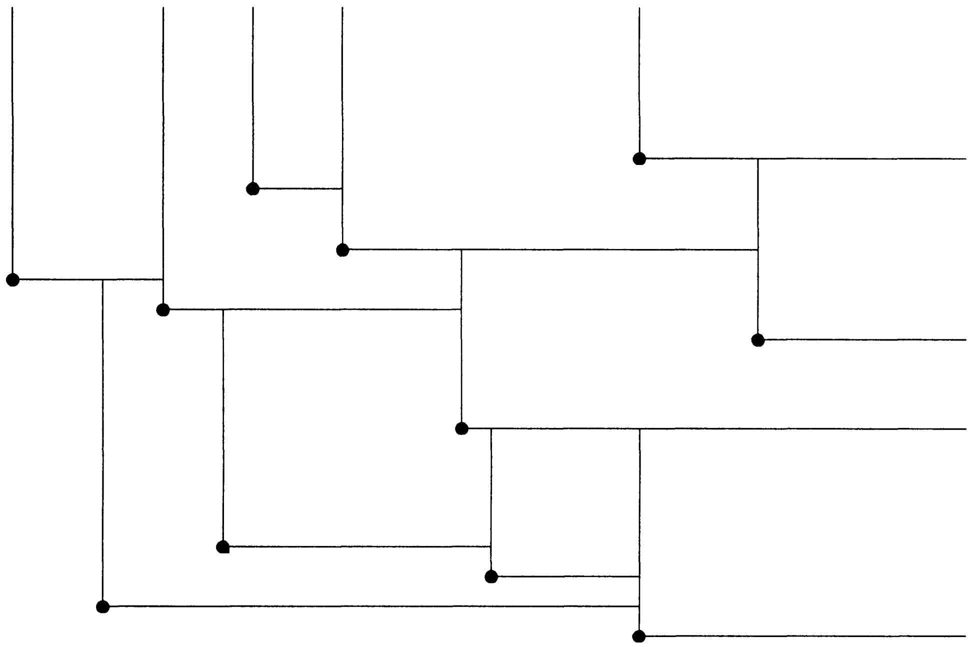

Figure 2: An RT Voronoi diagram.

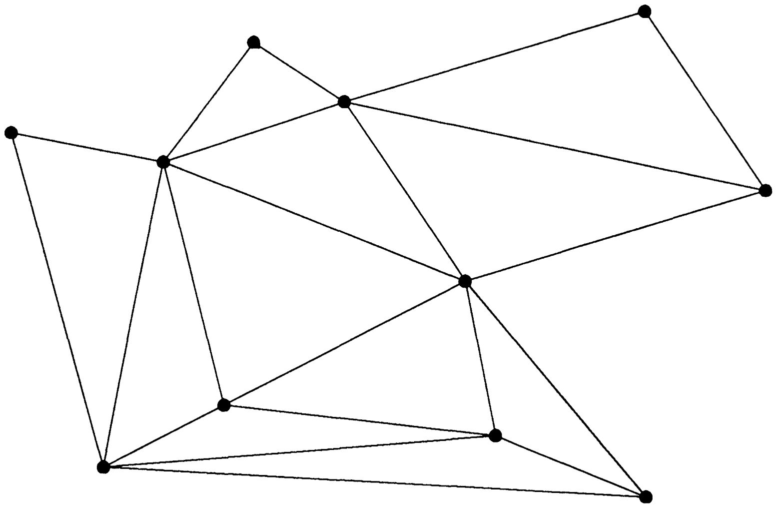

Figure 3: The corresponding RT Delaunay triangulation.

Although the resulting Voronoi diagram is a bit unusual, it retains many of the important properties of a standard Voronoi diagram. There are Voronoi regions, where each location in the region is "closer" to that region's source point than to any other source point. There are Voronoi boundaries; a circle (here a circle is a reflected version of the initial right triangle) placed with its center on the boundary can be grown until it just touches two source points with no source points in its interior. Voronoi boundaries meet to form Voronoi vertices; a circle can be placed on a Voronoi vertex and grown so that it just touches three source points with no source points in its interior. If we place a segment between each pair of source points whose regions share a portion of a Voronoi boundary then the result is the RT Delaunay triangulation of the source points (see Figure 3).

We produce an efficient algorithm for the RT Voronoi diagram by taking advantage of some of its less-standard features. In the RT Voronoi diagram all the Voronoi boundaries are either horizontal or vertical line segments. Not only that, each Voronoi boundary starts at one of the data points. This observation leads to a particularly efficient sweepline algorithm. Following the standard sweepline paradigm, a sweepline moves in discrete steps stopping at each interesting event. For this Voronoi diagram, each interesting event is closely associated with two of the initial n data points: each event occurs to the right of one data point and above another data point. Thus, once the data points are sorted, by x-coordinate and by y-coordinate, we know where all sweepline events occur.

The remainder of the algorithm is based on the following intuitive ideas. Each data point sends out two rays, one toward the right and one toward the top. Whenever two rays collide, only one, the one with the closest source, continues; the proof that this is the right thing to do is an immediate consequence of the expanding wave view of the Voronoi diagram. The Voronoi diagram is built using a horizontal sweepline sweeping upward. As this sweepline moves, we keep track of which vertical rays are still active by halting at each horizontal ray to determine which of the active vertical rays are blocked.

To do this efficiently, we need to use a priority queue with the following operations: Insert, Delete, and Successor (i.e., given some value i in the priority queue, determine j, the next larger value that is currently in the priority queue). The integer priority queue originally due to van Emde Boas [VKZ77, Joh82] has just the properties we need. Each data point can be assigned one of the integers 1 through n based on its order when sorted by x-coordinate (i.e., the data point with least x-coordinate is 1, the one with next smallest x-coordinate is 2, etc.). These integer values can be used in an integer priority queue with operations Insert, Delete, and Successor. Insert and Delete each take time $O(\log\log n)$; the time for Successor is $O(1)$.

Initially, no vertical rays are active and the horizontal sweepline starts at the data point p with least y-coordinate. Suppose p has been assigned the integer value i based on its ordering by x-coordinate. Then i is inserted in the priority queue and we have a vertical ray (from p) that is part of the Voronoi diagram.

The sweepline move upward through the data points, stopping at each one until all have been processed. Each time the sweepline stops at a data point q we insert the integer value associated with q into the priority queue. Then we check q's successor in the priority queue to see if the horizontal ray from q blocks the vertical ray from q's successor (the ray with the closest source continues, the other is blocked). If q's successor is blocked then we delete it from the priority queue and check q's new successor, continuing until q's horizontal ray is blocked or we run out of successors.

To show that this algorithm takes $O(n\log\log n)$ time we need to show that the operations Insert, Successor, and Delete are each done at most $O(n)$ time. This is obvious for operations Insert and Delete, since each data point is inserted at most once when the sweepline crosses it. To see that operation Successor is done at most $O(n)$ times, note that for each Successor operation one ray, either a horizontal ray or a vertical ray, is blocked. Each data point sends out exactly one horizontal ray and one vertical ray; thus, thus there are at most $2n$ rays that can possibly be blocked. This gives us the following theorem.

Theorem 1 *Given a set S of n points in the plane where the points have been presorted by x coordinate and presorted by y-coordinate, the RT Voronoi diagram of S can be constructed in time $O(n\log\log n)$.*

This theorem has a number of straightforward consequences. Assume we are given an arbitrary triangle Δ with one distinguished vertex. Then for a set S of n points in the plane, after two presorts — in directions parallel to the sides adjacent to the distinguished vertex of the triangle — we can construct any of the following in $O(n\log\log n)$ time:

- The Voronoi diagram of S using a convex distance function based on Δ where the distinguished vertex of Δ is used as its center. This is a consequence of combining Theorem 1 with some simple geometric transformations.

- The Delaunay triangulation of S using a Δ-based convex distance function. This holds since it takes just linear time to go from a Voronoi diagram to the corresponding Delaunay triangulation.

- The Voronoi diagram of S using a convex distance function based on Δ where Δ's *center* is arbitrary. This holds since it takes just linear time to go from a Delaunay triangulation to a Voronoi diagram and the Delaunay triangulation is unaffected by the location of the triangle's *center.*

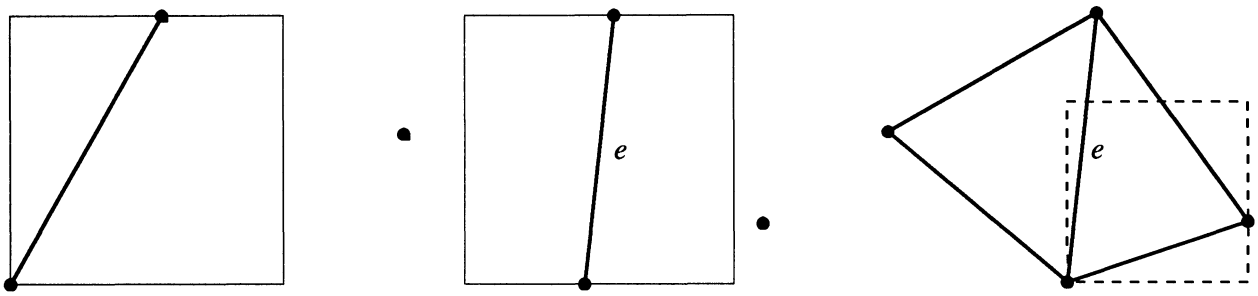

Figure 4: A corner edge, a noncorner edge e with its blocking points, and the corner edges that surround e.

5 Squares and Other Polygons

In this section, we show that once we have Delaunay triangulations for a number of appropriately chosen triangle-based convex distance functions, the Delaunay triangulation for an arbitrary polygon-based convex distance function can be built in linear time. Consequently, computing the Voronoi diagram of a polygonal convex distance function takes time $O(n \log\log n)$, given that the distance-defining polygon has a constant number of sides and that points are presorted in a constant number of directions.

For clarity we illustrate the reduction in a simple case using a convex distance function based on a square (i.e., the L_∞ metric). The general case of arbitrary polygonal convex distance function is discussed at the end of the section. In the square case, we need four triangle-based Delaunay triangulations. The triangles used are the four isosceles right triangles with short sides parallel to the axes. These Delaunay triangulations can be computed in time $O(n \log\log n)$ given that points are presorted in the x and y directions.

We distinguish two types of edges in the L_∞ Delaunay triangulation: *corner edges* and *noncorner edges* (see Figure 4). A Delaunay edge is called a corner edge if there exists an empty square through the endpoints of the edge with a corner of the square on one of the endpoints. The remaining Delaunay edges are called noncorner edges. Note that since an empty square placed on an edge can slide while remaining in contact with the edge's endpoints, a Delaunay edge is a noncorner edge only if the sliding is blocked by other source points — otherwise the square could slide far enough to show that the edge is really a corner edge. Each noncorner edge has a pair of these *blocking points.* Further, each noncorner edge is surrounded by corner edges. To see this, consider sliding the empty square against one of the blocking points. It is easy to see that the square can be shrunk to confirm the existence of two corner edges for each blocking point (see Figure 4).

We claim that (1) corner edges can be determined by examining the four triangle-based Delaunay triangulations and (2) noncorner edges are easy to determine once we have the corner edges. The first part of the claim is proved in the following paragraphs. The second part follows from the observation above that each noncorner edge exists in a cell of four corner edges; we simply construct the planar graph consisting of all corner edges, then examine each 4-cell to choose the appropriate diagonal.

Note that there are four types of corner edges corresponding to the four corners of the square. Without loss of generality we show how to find all those corner edges that use the

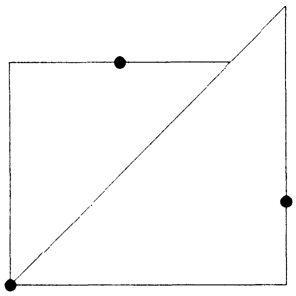

Figure 5: The closer hit corresponds to a corner edge.

lower left corner of the square.

We can find the lower-left corner edge for source point p by placing a very small square with its lower left corner on p; we then grow the square until it hits some other source point. This first point that we hit (assuming we hit something at all) determines the corner edge for p. We recast this view of corner edges to use triangles instead of a square. We split the square into two triangles (see Figure 5); call the triangles T_1 and T_2. Think of placing both triangles on our source point p, then expanding each of them independently until they each hit a source point. The closer of the (at most) two hits determines the corner edge for p.

We build the corner edges for the L_∞ Delaunay triangulation by examining the corner edges for the four triangle-distance Delaunay triangulations. Observe that each edge of a triangle-distance Delaunay triangulation is a type of corner edge. For each Delaunay edge there is an empty circle (actually a triangle) that has the endpoints of the edge on its boundary. Any such triangle can be shrunk to show that the Delaunay edge is actually a corner edge that uses one of the triangle's three corners. Thus, once we have a triangle-distance Delaunay triangulation, we can, in linear time, label each edge of the triangulation to show which corner of the triangle it uses. Also in linear time, we can label each vertex v to show those corner edges (at most three) that have v at a corner of the empty triangle.

Now we have enough information to find the lower-left corner edges of the L_∞ Delaunay triangulation. For each data point p, find the lower-left corner edge of p in the T_1 Delaunay triangulation and the lower-left corner edge of p in the T_2 Delaunay triangulation. The shorter of these two edges is clearly a lower-left corner edge of the L_∞ Delaunay triangulation (refer back to Figure 5). One or both of the two triangle-based corner edges may not exist; in this case, we either choose the one that does exist, or conclude that p does not have an L_∞ lower-left corner edge, as appropriate. Note that when choosing the *shorter* edge, it is important to measure lengths in the L_∞ metric. This complete the proof of the following theorem.

Theorem 2 *Given a point set S of size n and given the four Delaunay triangulations based on the four possible isosceles triangles with short sides parallel to the axes, the L_∞ Delaunay triangulation for S can be found in time $O(n)$.*

Corollary 3 *Given a set S of n points in the plane where the points have been presorted by x coordinate and presorted by y-coordinate, the L_∞ Voronoi diagram of S can be constructed in time $O(n \log\log n)$.*

Proof: Immediate consequence of Theorems 1 and 2. □

If the only goal is to compute the L_∞ Delaunay triangulation then the triangle-based Delaunay triangulations have too much information and need not be stored in their entirety. In particular, corner edges that use a triangle's right-angle corner are unused.

We now state the general theorem for arbitrary polygonal convex distance functions. Note that competitive algorithms in the literature [CD85, For85] have running time $O(kn \log kn)$ where n is the number of data points and k is the number of sides for the distance-defining convex polygon.

Theorem 4 *Let P be a convex polygon with k sides and let S be a set of n points in the plane. The Delaunay triangulation of S using a convex distance function based on P can be computed in time $O(k^2 n \log\log n)$ after presorting S in each of the $O(k^2)$ directions parallel to a side or a diagonal of P.*

Proof: (Sketch.) Just as for the square, the P-based Delaunay triangulation has *corner edges* and *noncorner edges.* By placing a copy of P on a noncorner edge and by sliding and growing/shrinking P one can show that each noncorner edge is within a cell consisting of k corner edges.

We claim that all the corner edges of the P-based Delaunay triangulation can be found by examining appropriate triangle-based Delaunay triangulations. To find all corner edges that use corner c of P, we need the Delaunay triangulations for each of $O(k)$ triangles — those triangles that use c and some other edge of P. So to find all corner edges, we need $O(k^2)$ triangle-based Delaunay triangulations. By Theorem 1, the necessary triangle-based Delaunay triangulations can be found in $O(k^2 n \log\log n)$ time, since we have assumed the necessary presorting is already done. Given these triangle-based triangulations, all the corner edges of the P-based Delaunay triangulation can be found in time $O(k^2 n)$ by using techniques similar to those used for the L_∞ case.

The corner edges alone form a planar graph containing $O(n/k)$ k-cells. To see that there are $O(n/k)$ such cells, observe that each noncorner edge is part of a group of $k-3$ noncorner edges: those that share the same k-cell. There are $O(n)$ noncorner edges divided into groups of size $O(k)$; so there are $O(n/k)$ groups, each of which corresponds to a single k-cell.

We complete the triangulation by triangulating within k-cells using existing techniques from the literature. For a single k-cell the appropriate diagonals can be found by forming the P-based Delaunay triangulation of the cell's vertices, taking time $O(k^2 \log k)$ [CD85, For85]. Since there are $O(n/k)$ such cells, the total time spent triangulating the cells is $O(nk \log k)$. Thus, the total running time for forming the P-based Delaunay triangulation is dominated by the time to form the necessary triangle-based Delaunay triangulations: $O(k^2 n \log\log n)$. □

In many cases, especially for regular polygons, the number of sorts is not as bad as implied by Theorem 4. We took advantage of this in the case of the L_∞ metric, where we did not actually need to sort along the diagonals of the square.

It may be possible to improve the dependence of the running time on k somewhat. Ideally, the exponent of k should be 1, although we have been unable to bring it below 2.

6 Applications

The term "applications" used here is perhaps a bit of misnomer, since the improvements in running times are of theoretical rather than practical interest. The algorithms are unlikely to be useful except for very large data sets that happen to be already sorted in two or more directions. Additional information on each of these applications can be found in [PS85]. Work combining some of these applications with fast approximate sorting can be found in [BKRS89].

Largest Empty Circle. Use a convex distance function based on an equilateral triangle to build a Voronoi diagram in $O(n \log\log n)$ time after two initial sorts. Transform it in linear time to get the Voronoi diagram in which the center of the triangle is at the true center. Find the largest empty equilateral triangle in linear time by checking all the Voronoi vertices (the vertices where three Voronoi boundaries intersect). The circle inscribed within this triangle has radius within a factor of two of the true largest empty circle. Better approximations can be found by using polygons with more sides than the triangle used here.

Euclidean Minimum Spanning Tree (EMST). Use a convex distance function based on an equilateral triangle to build a Voronoi diagram in $O(n \log\log n)$ time after two initial sorts. Transform it in linear time to a triangle-distance Delaunay triangulation. This triangulation has the property that, for any two vertices A and B, the distance from A to B traveling along edges of the graph is at most twice the straight-line distance between A and B [Che89]. Consider the true EMST for the initial set of data points. For each edge in this EMST there is a set of edges in the Delaunay triangulation that connects the same two vertices and is at most twice the length of the EMST edge. Take the set of all Delaunay edges that correspond in this way to some EMST edge. This is a spanning graph for the initial set of data points and its total length is less than twice the length of the EMST. Thus, since the EMST is the smallest spanning graph, the minimum spanning tree for the Delaunay triangulation has total length at most twice the length of the EMST. We can find the minimum spanning tree for the Delaunay triangulation in linear time since it is a planar graph [CT76]. Thus, after two initial sorts, we can find a spanning tree that is within a factor of two of the EMST in $O(n \log\log n)$ time.

Even Better EMST. Several triangle-based Delaunay triangulations can be combined into a single (nonplanar) graph with the property that, for any two vertices A and B, the distance from A to B traveling along edges of the graph is at most $1+\varepsilon$ times the straight-line distance between A and B. This follows from a straightforward extension to the results in [Che89]. Intuitively, the unit circle is divided into m isosceles triangles, all congruent and radiating from the center. Combining Delaunay triangulations based on these triangles leads to the $1+\varepsilon$ bound with $\varepsilon = 2\sin\frac{\pi}{m}$. Thus, for a fixed ε (and m), after a constant number of initial sorts, we can find a spanning tree that is within a factor of $1+\varepsilon$ of the EMST in $O(n \log\log n)$ time.

Traveling Salesman Problem. The Euclidean Minimum Spanning Tree can be used almost directly as an approximation to the Traveling Salesman Problem that is off by a factor of at most two [PS85]. Thus the previous results on EMSTs indicate that (1) two presorts can be used to do a very fast approximation to the Traveling Salesman Problem that is within a factor of four and (2) more presorts can be done to get a fast approximation that is within a factor of $2 + \varepsilon$.

7 Comments

This idea of moving-the-center to build Voronoi diagrams can also be applied to standard Voronoi diagrams. The standard Euclidean distance is just a convex distance function based on a standard circle with its center in the usual place. If we move the center to the edge of the circle then we get a distance function equivalent to the one used by Fortune for his sweepline Voronoi diagram algorithm [For87].

The RT (right-triangle-distance) Voronoi diagram and the corresponding Delaunay triangulation can be built with less round-off error than other types of Voronoi diagrams. If the initial data points are correct then the RT Voronoi diagram can be built using only addition and subtraction — no multiplication or division! Thus, the RT Voronoi diagram can easily be calculated exactly. Unless extraordinary care is taken with the arithmetic, the algorithms for the standard Voronoi diagram will usually produce only an approximation to the true Voronoi diagram.

An interesting open problem is whether sorting helps for the standard Voronoi diagram as it does for convex-polygon-based Voronoi diagrams.

References

[Aur91] F. Aurenhammer, Voronoi Diagrams – a Survey of a Fundamental Geometric Data Structure, *ACM Computing Surveys*, 23 (1991), 345–405.

[BKRS89] M. W. Bern, H. J. Karloff, P. Raghavan, and B. Schieber, Fast Geometric Approximation Techniques and Geometric Embedding Problems, *Proceeding of the Fifth Annual Symposium on Computational Geometry* (1989), ACM Press, 292–301.

[CD85] L. P. Chew and R. L. Drysdale, Voronoi Diagrams Based on Convex Distance Functions, *Proceedings of the First Annual Symposium on Computational Geometry* (1985), ACM Press, 235–244.

[CF88] L. P. Chew and S. Fortune, Sorting Helps for Voronoi Diagrams, *13th Symposium on Mathematical Programming* (1988).

[Che89] L. P. Chew, There are Planar Graphs Almost as Good as the Complete Graph, *Journal of Computer and System Sciences*, 39 (1989), 205–219.

[CT76] D. Cheriton and R. E. Tarjan, Finding Minimum Spanning Trees, *SIAM J. Comput.*, 5 (1976), 724–742.

[For85] S. Fortune, Fast Algorithms for Polygon Containment, *Proc. 12th International Colloquium on Automata, Language and Programming*, Lecture Notes in Comp. Science 194, Springer-Verlag, 1985, 189–198.

[For87] S. Fortune, A Sweepline Algorithm for Voronoi Diagrams, *Algorithmica*, 2 (1987), 153–174.

[Joh82] D. B. Johnson, A Priority Queue in Which Initialization and Queue Operations Take $O(\log\log D)$ Time, *Mathematical Systems Theory*, 15 (1982), 295–309.

[Lee80] D. T. Lee, Two-Dimensional Voronoi Diagrams in the L_p Metric, *JACM*, 27 (1980), 604–618.

[LW80] D. T. Lee and C. K. Wong, Voronoi Diagrams in the L_1 (L_∞) Metrics with 2-Dimensional Storage Applications, *SIAM J. Comput.*, 9 (1980), 200–211.

[PS85] F. P. Preparata and M. I. Shamos, *Computational Geometry*, Springer-Verlag, 1985.

[VKZ77] P. van Emde Boas, R. Kaas, and E. Zijlstra, Design and Implementation of an Efficient Priority Queue, *Math. Systems Theory*, 10 (1977), 99–127.