10-1-1992

Algorithms for Closest Point Problems: Practice and Theory

Peter Su
Dartmouth College

ALGORITHMS FOR CLOSEST POINT PROBLEMS: PRACTICE AND THEORY

Peter Su

Technical Report PCS-TR92-185

October,1992

Algorithms for Closest Point Problems: Practice and Theory.

Peter Su

Dartmouth College

ABSTRACT

This paper describes and evaluates known sequential algorithms for constructing planar Voronoi diagrams and Delaunay triangulations. In addition, it describes a new incremental algorithm which is simple to understand and implement, but whose performance is competitive with all known methods. The experiments in this paper are more than just simple benchmarks, they evaluate the expected performance of the algorithms in a precise and machine independent fashion. Thus, the paper also illustrates how to use experimental tools to both understand the behavior of different algorithms and to guide the algorithm design process.

1. Introduction

This paper discusses the following basic problem in computational geometry:

Voronoi diagram

Let S be a set of n points (or **sites**) in space. For each $s \in S$ define $V(s)$ to be a a collection of points such that for all $x \in V(s)$ and $y \in S - \{s\}$, $\text{dist}(x, s) \leq \text{dist}(x, y)$. The Voronoi diagram of S is the collection $\{V(s) \mid s \in S\}$.

The straight-line dual of the Voronoi diagram is called the Delaunay triangulation. Figure 1 shows the Voronoi diagram and the Delaunay triangulation of 16 points in the plane.

In our algorithms, we assume that no four points in S are co-spherical. In this case, the Delaunay triangulation partitions the convex hull of S into simplices, and the circumsphere of each simplex contains no site. For example, in the planar case, the convex hull is divided into triangles, and each triangle has an empty circumcircle.

Researchers in computational geometry have been very successful in designing algorithms for this problem that are elegant, theoretically efficient and practical. Bentley, Weide and Yao [1] describe "cell" algorithms for nearest-neighbor search and Voronoi diagram construction that use buckets as their main data structure and run in linear expected time for uniformly distributed input. These algorithms use a technique called "spiral search" which they show to be both simple and efficient.

Guibas and Stolfi [11] present the first conceptually simple algorithms for this problem. They also describe the ideal data structure for representing the diagram and its dual. Their

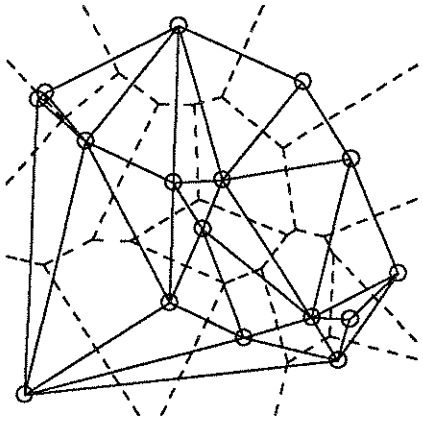


Figure 1: The Voronoi diagram and Delaunay triangulation.

algorithms then construct Delaunay triangulations rather than Voronoi diagrams. I also will use this approach because the resulting algorithms are simpler and more elegant.

Guibas and Stolfi show how to implement two algorithms for constructing the Delaunay triangulation. The first is a divide and conquer algorithm, and the second constructs the diagram incrementally. These algorithms run in $O(n \log n)$ and $O(n^2)$ worst-case time respectively. Fortune [8] examines an optimal sweepline algorithm for constructing the Voronoi diagram. The algorithm uses a transformation of the plane to allow a straightforward sweepline algorithm to correctly compute either the Voronoi diagram or the Delaunay triangulation of the input.

Many proposed algorithms have efficient expected-case runtimes. These algorithms come in two flavors. First, there are algorithms whose expected run-time depends on the input distribution. These algorithms tend to use some sort of bucketing scheme to take advantage of smooth input distributions. Bentley, Weide and Yao's cell algorithms fall into this class [1]. Maus [14] presents a simple algorithm that uses bucketing to speed up an algorithm for constructing the Delaunay triangulation that is similar to gift-wrapping algorithms for convex hulls [17]. The algorithm runs in linear expected time. Dwyer's thesis [6] shows that a generalization of this algorithm to higher dimensions also runs in linear expected time on uniform inputs. Dwyer [7] also shows that a modified version of the divide-and-conquer algorithm runs in $O(n \log \log n)$ expected time. Finally, Ohya, Murota and Iri [16] describe an incremental algorithm that uses a combination of bucketing and quad-trees to achieve linear expected time.

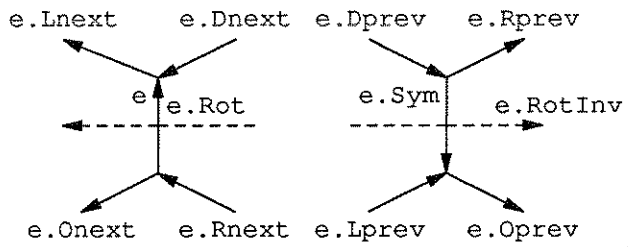
A second class of fast expected-time algorithms use randomization, and achieve their expected performance on any input. Clarkson and Shor [3] describe an incremental algorithm that inserts the points in a random order and runs in $O(n \log n)$ expected time. Knuth, Guibas and Sharir [12] give a refined analysis of a modified randomized incremental algorithm, and Shair and Yaniv [18] refine the analysis further and discuss an implementation of this algorithm. These algorithm use a persistent tree structure to perform point location in the current diagram.

Devillers, Meiser and Teillaud [5] use similar data structures in their dynamic algorithms. Finally, Clarkson, Mehlhorn and Seidel [4] show how to extend Clarkson's original results to develop dynamic algorithms for higher-dimensional convex hulls, which can be used to construct the Voronoi diagram.

Many of these algorithms are simple enough to be implemented. In the rest of the paper, I will describe the incremental, divide-and-conquer and sweepline algorithms in more detail and present detailed a performance analysis of their implementations. In addition, I will show how to use Bentley's cell technique to speed up the incremental algorithm so that on most distributions of points it displays performance that is competitive with all known methods.

2. Divide-and-Conquer

Sequential algorithms for constructing the Delaunay triangulation come in three basic flavors: divide-and-conquer, sweepline, and incremental. Divide-and-conquer algorithms work by dividing the original input set into smaller subsets and solving the sub-problems recursively. Then, answers for the sub-problems must be merged together to form the final answer for the entire problem. Guibas and Stolfi [11] present a conceptually simple method for implementing this idea, and an elegant data structure for representing both the Delaunay triangulation and the Voronoi diagram at the same time. The "quad-edge" data structure represents a subdivision of the plane in terms of the edges of the subvision. Each edge is a collection of pointers to neighboring edges in the subdivision or its dual:



The solid edges in the picture are from the subdivision, while the dotted edges show the dual of `e`. Each edge has an origin, `e.Org`, a destination, `e.Dest`, and a left and right face. The edge `e.Sym` is the same as `e` but directed the opposite way. The edge `e.Rot` is the edge dual to `e`. The edges `e.Onext` and `e.Lnext` are the next edges counterclockwise around the same origin and left face respectively. The other edges in the picture are defined analogously for the destination of `e` and the right face of `e`. Guibas and Stolfi show that each of these edges is computable from suitable combinations of `e.Sym` and `e.Rot`. The proof of this, and the other algebraic properties of these abstractions is beyond the scope of this paper.

Edges are represented using a record of four pointers and four data fields. This data represents the edges `e`, `e.Rot` `e.Sym` and `e.Rot.Sym`, and the pointers represent the `Onext` edge of each of these edges. In general, if e is an edge record, then $e[i]$ represents the edge $e.Rot^i$. As an example, the following picture shows a triangle and its quad-edge representation. In the picture, bold lines connect nodes in a vertex ring while thin lines connect nodes in a face ring (see figure 2).

Faces and vertices in the subdivision are represented implicitly by circular linked lists of edges. To add and delete edges from the subdivision, we use an operator called `splice(a,b)`

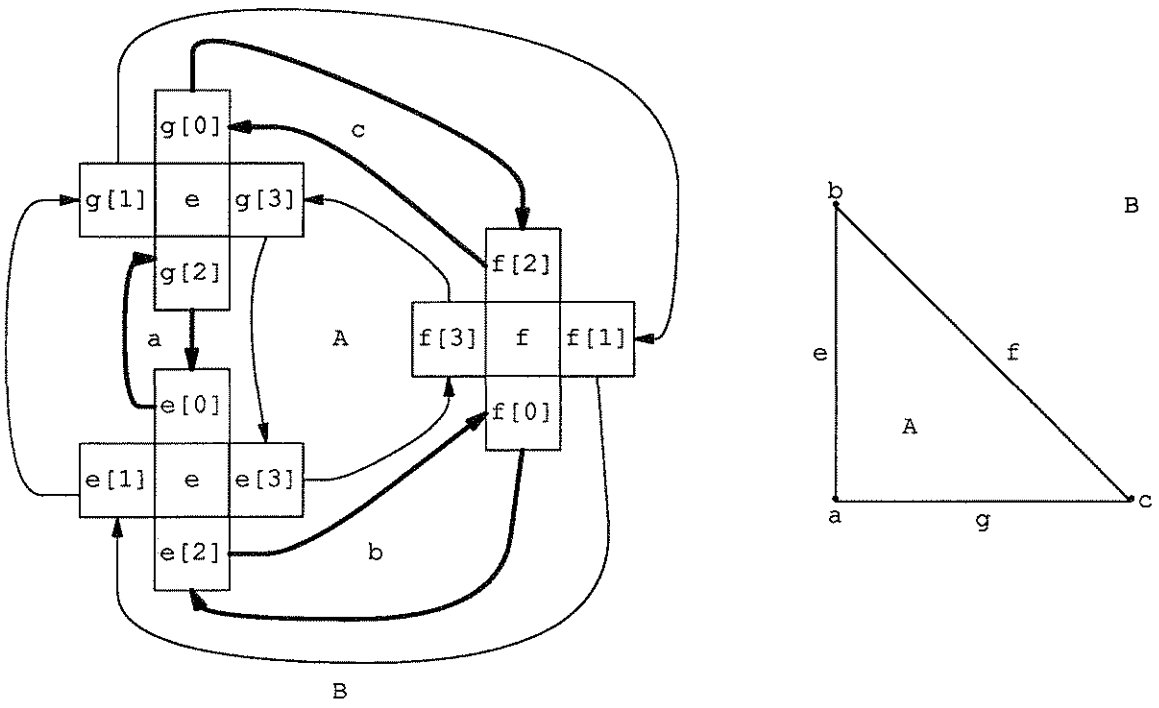


Figure 2: Quad-edge representation of a simple subdivision.

which is similar to linked list insertion. A call to `splice(a,b)` has the effect of splicing the ring of the edge `b` together with the ring of the edge `a`. If `a` and `b` are part of the same ring, then `splice` splits that ring into two pieces. Guibas and Stolfi give a more detailed description of how this works, and also discuss the topological and algebraic foundations of this representation in great detail. To understand the algorithms in this paper, it should be sufficient just to keep the above picture in mind when reading my pseudocode.

Guibas and Stolfi's algorithm uses two additional geometric primitives. First, it uses a two-dimensional orientation test, `CCW(a,b,c)` that returns whether the points `a`, `b` and `c` form a counter-clockwise turn. The second is `in-circle(a,b,c,d)` which determines whether `d` lies within the circle formed by `a`, `b` and `c`. These primitives are defined in terms of two and three dimensional determinants. Fortune shows how to compute these accurately with finite precision [9,10].

With these primitives in place, Guibas and Stolfi's algorithm proceeds using a standard divide-and-conquer scheme. The points are sorted by x-coordinate, and split until three or fewer points remain in each sub-problem. Then, the sub-problems are merged together using the following routine:

```
Merge(L, R) {
  Find the lower common tangent between R and L;
```

```
bl := lower tangent;
Walking from the lower tangent to the upper {
  lcand := bl.Sym.Onext; lvalid := CCW(lcand.Dest, bl.dest, bl.org);
  if (lvalid) {   /* remove bad edges from L */
    while (in-circle(bl.Dest, bl.Org, lcand.Dest,lcand.Onext.Dest)) {
        t := lcand.Onext; Delete lcand ; lcand := t;
      } }
  rcand := bl.OPrev; rvalid := CCW(rcand.Dest, bl.dest, bl.org);
  if (rvalid) {   /* Remove bad edges from R */
    while (in-circle(bl.Dest, bl.Org, lcand.Dest,lcand.Oprev.Dest)) {
        t := rcand.Oprev; Delete rcand ; rcand := t;
      } }
  if ((!lvalid) && (!rvalid)) return;
  /* Now connect the new cross-edge. */
  if (!lvalid || (rvalid &&
      in-circle(lcand.Dest, lcand.org, rcand.Org, rcand.Dest)))
    bl := connect(rcand, bl.Sym);
  } else {
    bl := connect(bl.Sym, lcand.Sym); bl := bl.Sym; } } }
```

The first line of code finds the edge that connects the convex hulls of the left and right diagrams together. Then, the `in-circle` tests are used to find edges in `L` and `R` that are not valid, and to determine how to connect new edges between `L` and `R`. The movie in figure 3 shows the operation of the `Merge` loop, starting from the lower common tangent of the left and right diagrams. In the frames, bold circles show successful `in-circle` tests, dotted edges are being tested for validity, the current base edge is bold, and cross-hairs mark candidate points for connecting the new cross-edge. Reading the frames left to right and top to bottom, we see that the first loop deletes invalid edges in the left diagram, starting with the first edge incident on `bl` and then moving counterclockwise into `L`. The second loop does the same for the right diagram in a clockwise fashion. Then, one final circle test determines a new cross edge between `L` and `R`. The routine `connect(a,b)` makes a new edge connecting `a.Dest` to `b.Org` in such a way that `a`, `b` and the new edge share the same left face. Guibas and Stolfi provide a proof that this scheme only deletes invalid edges and only adds valid ones, so the final diagram is the Delaunay triangulation of the whole set. They also show that the merge step takes $O(n)$ steps in the worst case, so the whole algorithm runs in $O(n \log n)$ worst-case time. Dwyer [7] showed that a simple modification to this algorithm runs in $O(n \log\log n)$ expected time on uniformly distributed points. Dwyer's algorithm splits the point set into vertical strips of width $\sqrt{n/\log n}$, constructs the DT of each strip by merging along horizontal lines and then merges the strips together along vertical lines. Experiments indicate that in practice this algorithm runs in linear expected time.

Another version of this algorithm, due to Katajainen and Koppinen [13] merges square buckets together in a "quad-tree" order. They show that this algorithm runs in linear expected time for uniform points. In fact, the performance of this algorithm is nearly identical to Dwyer's.

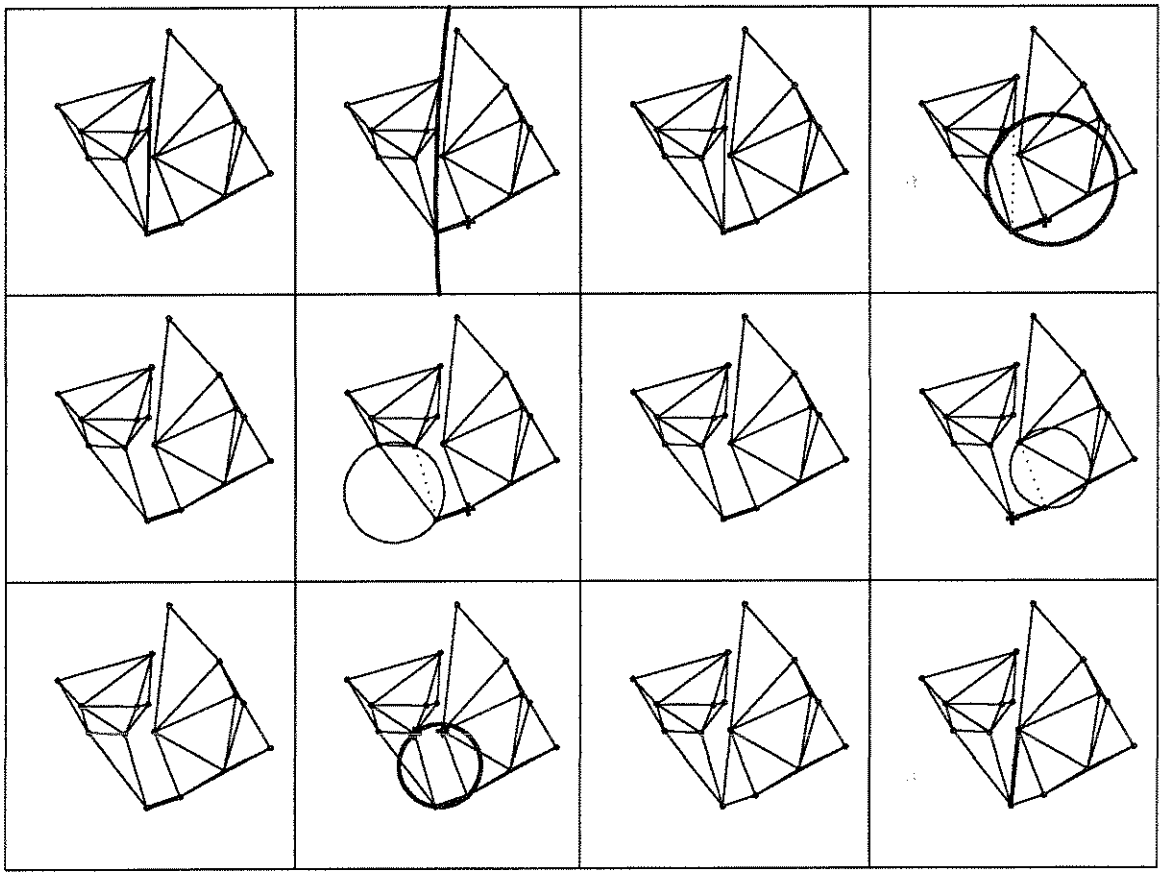


Figure 3: The merge loop.

3. Sweepline Algorithms

Fortune [8] discovered another optimal scheme for constructing the Delaunay triangulation using a sweepline algorithm. The algorithm keeps track of two sets of state. The first is a list of edges on the *frontier* of the diagram. These edges form a tour around the outside of the incomplete triangulation, and correspond to triangles that have not yet been completed. The algorithm also keeps track of a queue of events containing *site* events and *circle* events. Site events happen when the sweepline reaches a site, and circle events happen when it reaches the top of a circle formed by three vertices on the frontier. Events are ordered in the y direction, so the next event at any time is event in the queue with minimum y coordinate. The algorithm sweeps a line up in the y direction, and performs the following steps for each event:

```
Do_Event(e)
{
  if e is a site {
```

```
      Find the site T on the frontier such that the circle
        inscribed at e and T is empty;
      Add the edge (e,T) to the frontier;
      Update the event queue with any new circle events;
      Delete invalid circle events from the event queue;
    } else {
      Let (a,b,c) be the sites lying on the circle;
      Remove the edges (a,b) and (b,c) from the frontier;
      Add the edge (a,c) to the frontier;
      Update the event queue as above;
    }
  }
```

Frontier edges are stored as left/right half-edge pairs, so they get deleted once for each triangle they appear in. Circle events are added to the event queue when a new frontier edge makes an externally convex angle with either of its neighboring edges. Finally, new sites and edges may invalidate existing circle event. When a new site is swept, any future circle event incident on the left neighbor of the new edge becomes invalid, since the new site is in the circle represented by the future event. More formally, if (s, T) is the new edge, and the old circle passes through the points (a, T, c) in counter-clockwise order, the new circle passes through the points (a, T, s).

When the algorithm encounters a new circle event, it may delete an edge which is associated with a future circle event from the frontier. In this case, the future event is invalidated, and a new one is added that is incident on the new frontier edge (a, c) in the above code.

In his paper, Fortune shows that this scheme only adds valid Delaunay edges to the diagram, since the edges added when processing site events are valid, and circle events which are reached by the sweepline represent empty circles. Using standard tree-based data structures for the priority queue and the frontier, each new event can be processed in $O(\log n)$ time, so the whole algorithm uses $O(n \log n)$ time in the worst case.

In Fortune's implementation, the frontier is a simple linked list of half-edges, and point location is performed using a bucketing scheme. The x-coordinate of a point to be located is used as a hash value to get close to the correct frontier edge, and then the algorithm walks to the left or right until it reaches the correct edge. This edge is placed in the bucket that the point landed in, so future points that are nearby can be located quickly. This method works well as long as the query points and the edges are well distributed in the buckets. A bucketing scheme is also used to represent the priority queue. Members of the queue are bucketed according to their priorities, so finding the minimum involves searching for the first non-empty bucket and pulling out the minimum element. Again, this works well as long as the priorities are well distributed.

Figure 4 shows the operation of the algorithm on a small point set. Reading the frames from left to right and up to down, the first few frames show how a new site is processed. The sweepline is the dotted horizontal line, sites are small circles and circle events are marked with "+" symbols. Thick lines mark edges that still have at least one half edge on the frontier. Thin lines mark edges that have been totally removed from the frontier. Dotted edges are the frontier edges being searched. The new edge is added in the fifth frame, and in the sixth and eighth frames new circle events are added on either side of the new frontier edge. The movie flashes a

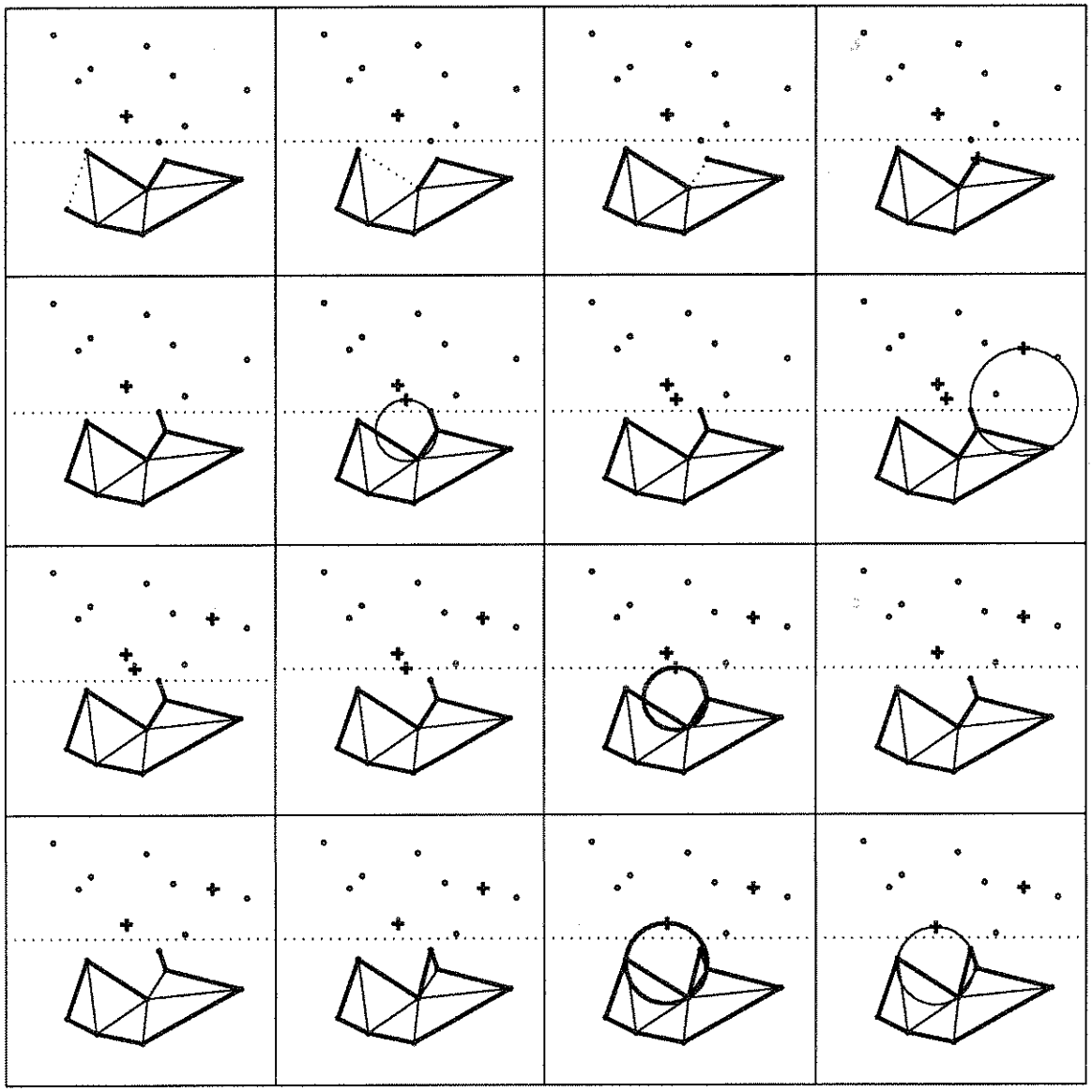


Figure 4: Fortune's algorithm.

thin circle and then leaves the "+" symbol behind as a marker. In the tenth frame, we move to a circle event. The circle being deleted is shown in bold in the eleventh frame. The thirteenth frames removes one edge from the frontier, the fourteenth frame closes the triangle with a new frontier edge. The last two frames show the invalidation of a circle event because the circle was incident on the edge removed from the frontier.

4. Randomized Incremental Construction

The third, and perhaps simplest class of algorithms for constructing the Delaunay triangulation is incremental algorithms. Incremental algorithms add sites to the diagram one by one and update the diagram after each site is added. Guibas and Stolfi [11] present a basic incremental algorithm at the end of their paper. The algorithm consists of two main subroutines: `Locate` locates a new site in the current diagram by finding the triangle that the point lies in, and `Insert` inserts a new site into the diagram by calling `Locate` and then iteratively updating the diagram until all edges invalidated by the new site have been removed:

```
Locate(start, p)
{
  e := start;
  while(1) {
    if (p == e.Dest || p == e.Org) return e;
    if (right_of(p,e.Org, e.Dest)) e := e.Sym;
    else {
      t1 := e.Onext;
      if (CCW(p,t1.Org,t1.Dest)) e := t1;
      else {
        t1 := e.Dprev;
        if (CCW(p,t1.Org, t1.Dest)) e := t1;
        else { return e; } } } }
}
Insert(p)
{
  e := Locate(last,p);
  if (p == e.Org || p == e.Dest) return;
  if (is_on(p,e.Org, e.Dest)) { t = e.Oprev; delete(e); e = t; }
  base = make_edge(e.Org, p); first = e.Org; splice(base,e);
  do {  /* Connect new point */
    t = base.Sym;
    base = connect(e, t);
    e = base.Oprev;
  } while (e.Dest != first);
  e = base.Oprev;
  for(;;) {   /* Flip invalid edges until done */
    t = e.Oprev;
    if (not CCW(t.Dest, e.Org, e.Dest) &&
        in-circle(e.Org,t.Dest,e.Dest, p)) {
      swapedge(e); e = e.Oprev;
    } else if (e.Org == first) { last = e; return; }
    else { e = e.Onext.Lprev; } }
}
```

The `Locate` routine works by starting at a random edge in the current diagram and walking

along a line in the direction of the new site until the right triangle is found. The algorithm is made simpler by assuming that the points are enclosed within large triangle. The `Insert` routine works by calling `Locate` and connecting the new point into the diagram with one edge to each vertex of the triangle found by `Locate`. Then, `Insert` updates the diagram by finding invalid triangles around the outside of the polygon containing the new site. These triangles are found using the `in-circle` test. Let ABC be such a triangle with the point A opposite the new site p. The edge flipping procedure replaces the edge BC with an edge from A to p, creating two new triangles ApC and ABp which are guaranteed to be Delaunay. The loop then adds the edges AC and AB to a queue of edges to be checked. It keeps track of this queue implicitly through links in the quad-edge data structure. When the loop comes back to the original starting edge, this queue is empty and the routine stops. Figure 5 shows the operation of the `Insert` routine. In the first frame, we are inserting the site p marked by the "+" symbol. The bold edges show the path that `Locate` takes through the diagram to find the new site. The second frame shows the three new edges that connect p into the diagram. Then, we start to test the validity of the edges surrounding p. The dotted edges are the ones being tested. The first two circles are empty and drawn with thin lines. The next circle contains p and is drawn in bold. The following frame shows the flipped edge as dotted and the corresponding new circle, which is now empty. The algorithm now moves on to the edges neighboring the edge that it just flipped. The next two frames show another edge flip. The next three edges are all valid. In the last frame, the algorithm has reached the starting edge, so the insertion is complete.

The worst case runtime of this algorithm is $O(n^2)$ since it is possible to cook up a point set and insertion order where each inserting the k^{th} point into the diagram causes $O(k)$ updates. However, if the points are inserted in a random order, Clarkson and Shor's analysis [3] shows that one can design an algorithm with an expected run-time of $O(n \log n)$ operations. The algorithm works by maintaining the current diagram along with an auxiliary data structure called a *conflict graph*. The conflict graph is a bipartite graph with edges connecting sites to triangles in the current diagram. If an edge (s, t) exists, it means that the site s lies within the circumcircle of the triangle t. When a new site s is added to the diagram, the algorithm deletes all the triangles that s conflicts with, adds new triangles in their place, and updates the conflict graph. The standard incremental algorithm could be modified to do this by adding code which updated the conflict graph after each edge flip. Clarkson and Shor use random sampling results to show that the total expected runtime of the algorithm is $O(n \log n)$ steps.

Guibas, Knuth, and Sharir [12] propose a similar algorithm that does not use a conflict graph, and provide a simpler analysis of its runtime. In particular, they show when we use a random insertion order, the expected number of edge flips that the standard incremental algorithm performs is linear. The bottleneck in the algorithm then becomes the `Locate` routine. Guibas, Knuth, and Sharir propose a tree-based data structure where internal nodes are triangles that have been deleted and the current triangulation is stored at the leaves. Whenever a triangle is deleted, it marks itself as such and then creates two pointers to the triangles that replaced it in the diagram. We can then modify `Locate` to search this tree, rather than doing the standard edge walk. It is then not hard to show that the expected cost of `Locate` will be $O(n \log n)$ time. Sharir and Yaniv [18] prove a bound of about $12nH_n + O(n)$.

Modifying the existing incremental algorithm to maintain this point location structure is not difficult. The easiest way to do this would be to keep a separate tree structure with links back

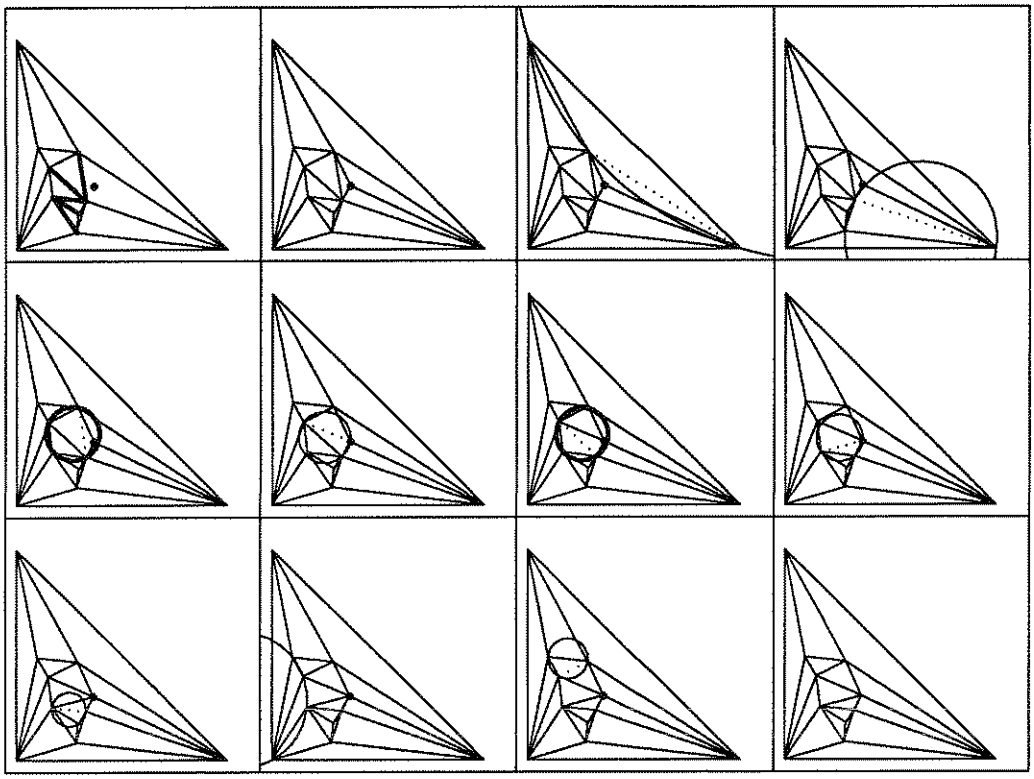


Figure 5: The incremental algorithm.

to the current quad-edge representation of the Delaunay triangulation. However, the resulting algorithm is somewhat unsatisfactory, since it uses $O(n \log n)$ expected time on point location when actually maintaining the Delaunay triangulation only costs $O(n)$ expected time.

We can remedy this situation if we assume that the input consists of points from some probability distribution. For example, if the distribution is the uniform distribution over the unit square, many known algorithms run in linear expected time [1,13,14,16]. The motivation for studying this case comes from the fact that in many applications involve data which is fairly uniform. In the next section, I will show how to modify the randomized incremental algorithm to run in linear expected time on such data.

5. A Faster Incremental Algorithm

The goal is to speed up point location in the incremental algorithm when the input is uniformly distributed. To do this, I use a simple bucketing algorithm similar to the one that Bentley, Wiede and Yao used for finding the nearest neighbor of a query point [1]. The idea is that point location in the triangulation is equivalent to nearest neighbor searching. Since the bucketing algorithm finds the nearest neighbor of a point in constant expected time for certain distributions of points,

I can reduce the total expected cost of the point location steps from $O(n \log n)$ time to $O(n)$ time while maintaining the relative simplicity of the incremental algorithm.

The bucketing scheme places the sites into a uniform grid as it adds them to the diagram. If two or more points fall in the same bucket, the last one inserted is kept and the others are discarded. To find a near neighbor, the point location algorithm uses finds the bucket that the query point lies in and searches in an outward spiral for a non-empty bucket. It then uses any edge incident on the point in this bucket as a starting edge in the normal point location routine. If the spiral search fails to find a point after a certain number of layers, the point location routine continues from an arbitrary edge in the current triangulation.

One key difference between this scheme and the one that Bentley, et al. propose is that in their scheme, all the sites are known in advance and are placed in the bucket structure as a preprocessing step while in my scheme, the sites must be added to the buckets one at a time as they are inserted into the triangulation. I use an adaptive grid to resolve this problem.

The other difference between my scheme and the one described by Bentley, et al. is that my scheme does not find the true nearest neighbor of the query point. My algorithm just finds a point that should be close to the actual nearest neighbor. In particular, it doesn't bother to resolve bucket collisions, and it also doesn't bother to perform the second phase of the spiral search after finding the initial near neighbor. I reasoned that eliminating these complexities would make the algorithm simpler, and would not seriously affect its runtime. Let $C > 0$ be some small constant that we can choose later. The details of the method are as follows:

```
Build_DT()
{
  grid_size := 4;
  maxl := 1;
  N := 0;
  C := C;
  for each site S {
    insert S into the diagram;
    bucket S in the current grid;
    if (++N > 4*C*grid_size) {
      make a new grid of size 4*grid_size;
      rebucket all points;
      grid_size *= 4;
      maxl++;
    }
  }
}
```

Now, we modify `Locate` to first search the bucket structure for a near neighbor, and then use some edge incident on that neighbor as a starting edge in a standard edge walk. The search procedure starts in the bucket that the new site lies in and searches out in a spiral pattern until has searched more than `maxl` layers (see figure 6). If it still hasn't found a non-empty bucket at this point, it gives up and `Locate` proceeds from an arbitrary edge. Assuming that each point `P` keeps track of an incident edge in `P.edge`, the modified point location routine looks like this:

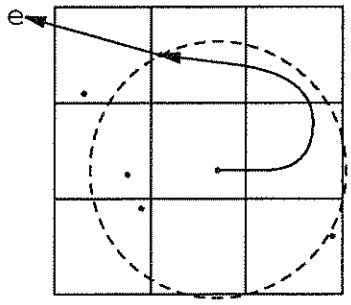


Figure 6: Spiral search for a near neighbor.

```
Spiral(p)
{
  bucket := compute_bucket(p);
  layer := 0;
  while (layer < maxl) {
    for each bucket in layer {
      if P is a point in this bucket return P.edge;
    }
    layer++;
  }
  return NULL;
}
New_Locate(p)
{
  edge := Spiral(p);
  if (edge == NULL) {
    edge = some arbitrary edge;
  }
  Locate(edge, p);
}
```

The bucketing procedure makes sure that the expected number of points per bucket is always at least C and at most $4C$. The variable `maxl` makes sure that the routine never looks at more than $\log_4 n$ layers of buckets. Thus, it is not hard to show that the expected cost of one point location is constant when the points are uniformly distributed. The probability that a point falls into a particular bucket is $q = C/n$. The probability that a particular bucket is empty is then

$$(1-q)^n = (1-C/n)^n \leq e^{-C}$$

The cost of the point location algorithm for a given query point is proportional to the number of

buckets examined in the spiral search and the number of edges examined in the locate step following the spiral search. There are two cases to consider. First, the spiral search may succeed before the cutoff point, after examining i buckets in the first $\log_4 n$ layers. Referring to the picture, the analysis of Bentley, Weide and Yao says that number of points in the dashed circle is bounded by $8C(11i+7)$. The expected number of edges in the area is bounded by six times the expected number of points. Thus, the expected number of edges that locate examines is less than $48C(11i+7)$. The total cost of the procedure in this case is then bounded by

$$O(1) \sum_{i \le 4\log^2 n} e^{-C(i-1)} O(i) = O(1).$$

If the spiral search fails, then the cost of the locate procedure is no more than $O(n)$ operations. But, since the algorithm must look at $O(\log^2 n)$ buckets before the spiral search fails, the probability of this happening is bounded by $e^{-C\log^2 n} = O(1/n)$, so the expected cost is also $O(1)$ operations.

The cost of maintaining the buckets is $O(n)$ if the floor function is assumed to be constant time. Since we insert the points in a random order, the expected number of edge flips that the algorithm does is $O(n)$. Thus, the total expected run time of this algorithm is $O(n)$ time when the points are uniformly distributed in the unit square.

The two important principles at work here are that the bucket structure takes advantage of the local nature of nearest neighbor search, and that the cost of adapting the bucket grid as more points are added can be amortized over the whole construction process. The resulting algorithm is only slightly more complicated than the original incremental algorithm, but as we will see, in most practical situations, it is nearly as fast as all of the methods I have described until now.

6. Empirical Results

In order to evaluate the effectiveness of the algorithms that I have described, I studied C implementations of each algorithm. Rex Dwyer and Steve Fortune were kind enough to provide code for their algorithms, and I implemented the incremental algorithms myself. None of the implementations are tuned in any machine dependent way, and all were compiled using the GNU C compiler and timed using the standard UNIX® timers. In addition, I added code to each algorithm to keep track of abstract costs that are important to performance. A good understanding of each algorithm, and profiling information from test runs determined what was important to monitor. Each algorithm was tested for set sizes of between 1024 and 131072 sites. Ten trials were run for each size, and the graphs either show the median of all the sample runs or a "box plot" summary of all ten samples at each size. In the box plots, a dot indicates the median value of the trials and vertical lines connect the 25^{th} to the minimum and the 75^{th} percentile to the maximum.

Finally, the programs include code to generate simple animations of their operation. This code outputs simple graphics commands which can be interpreted either by an interactive tool for X windows or a Perl program that generates input for various typesetting systems. These "movies" were very helpful in gaining intuition about every detail of the operation of the various algorithms, and also made debugging much easier. The movies in this paper were all generated automatically by the animation scripts.

6.1. Performance of the Incremental Algorithm

The performance of the incremental algorithm is determined by the cost of point location and the number of circle tests the algorithm performs. While the standard incremental algorithm spends almost all of its time doing point location, the bucket-based point location routine effectively removes this bottleneck. Figure 7 compares the performance of the two algorithms on uniformly distributed sites.

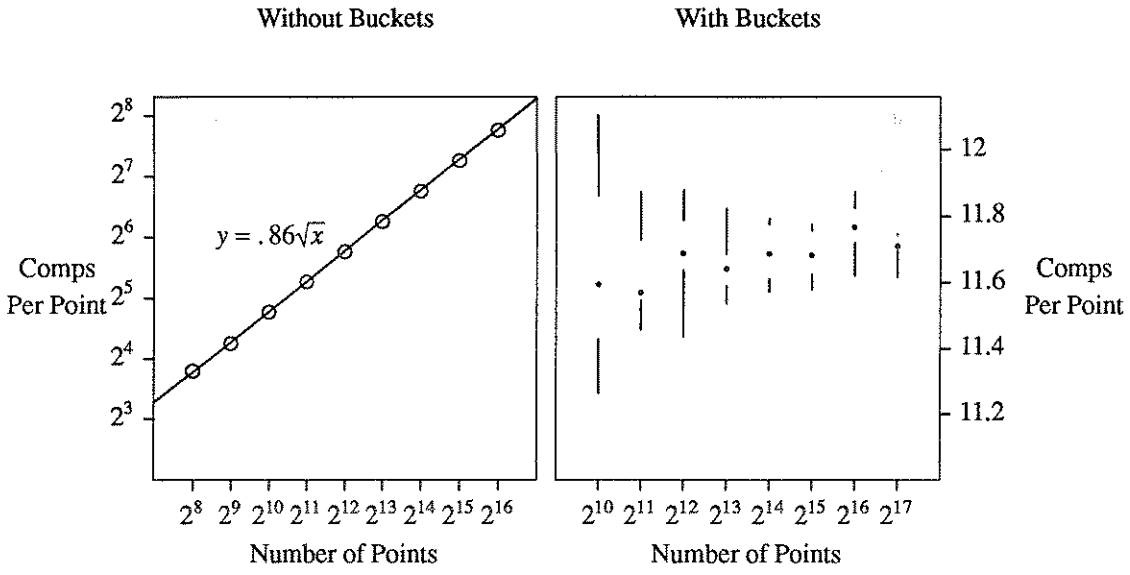


Figure 7: Comparison of point location costs.

The plots show that the standard incremental algorithm uses an average of $O(\sqrt{n})$ comparisons per point location step. Simple regression analysis indicates that the number of comparisons per point grows as $0.86n^{.49}$. Therefore, I have plotted the curve $.86\sqrt{n}$ in with the data. The curve is a close fit to the experimental data.

Adding the point location heuristic improves the performance of the incremental algorithm substantially. It is apparent that the number of comparisons per site is bounded by a constant near 12. Almost all of these comparisons are tests done during second phase of the algorithm, after the spiral search. These tests take much longer than the simple comparisons needed in spiral search. Therefore, although the spiral search accounts for about 10% of the total number of comparisons, its contribution to the runtime of the point location algorithm is much less significant.

The cost of point location depends on whether $\log_2 n$ is even or odd. This is easy to explain when you remember that the algorithm rebuckets the sites at each power of four. Because of this, at each power of four, the average search time dips, since one extra rebucket step reduces the average bucket density in the later phases of the construction process. As the input size moves towards the next power of four, the bucket density increases steadily. Thus, the cost of point

location see-saws up and down.

The number of comparisons needed for point the point location steps also depends on the average density of the sites in the bucket grid. If the density of points in buckets is too high, then the point location routine will waste time examining useless edges. On the other hand, if it is too low, the algorithm will waste time examining empty buckets. Figure 8 shows the dependence of the density on the cost of point location:

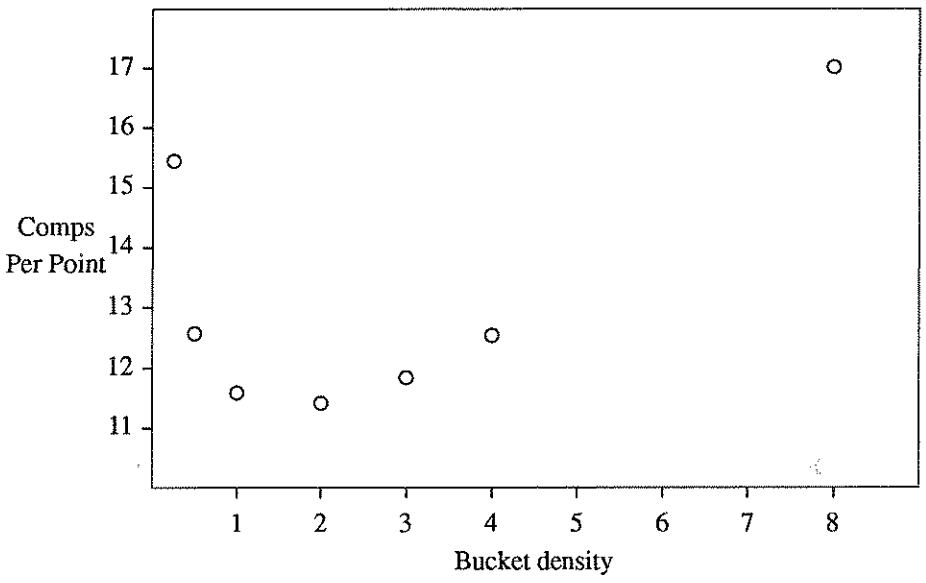


Figure 8: Effect of bucket density on point location speed.

The graph shows the average cost of point location over ten trials with $n = 8192$ and C ranging from 0.25 to 8. Based on this data, I set $C = 2$ for my timing runs. Although the graph shows a large variation in the number comparisons needed per site, the actual effect on the runtime of the algorithm was less than 10%.

The runtime of the incremental algorithm now depends on how many circle tests it performs. Since the algorithm inserts the points in a random order, the analysis of Guibas, Knuth and Sharir [12] shows that the total number of circle tests is asymptotically $O(n)$. Sharir and Yaniv [18] tightened this bound to about $9n$. Figure 9 shows that this analysis is remarkably accurate. Also shown on the plot is the cost of Dwyer's divide and conquer algorithm. Since this algorithm is also based primarily on circle testing, it makes sense to compare them in this way. The plot shows that Dwyer's algorithm performs about 25% fewer circle tests than the incremental algorithm. Profiling both programs shows that circle testing makes up roughly half the runtime of each, so Dwyer's algorithm should run roughly 10 to 15 percent faster than mine.

There are still many aspects of this algorithm's performance which I have not studied. In particular, I would like to tighten the analysis of the cost of the point location routine. However, this is likely to be difficult since it involves estimating the average number of edges in the

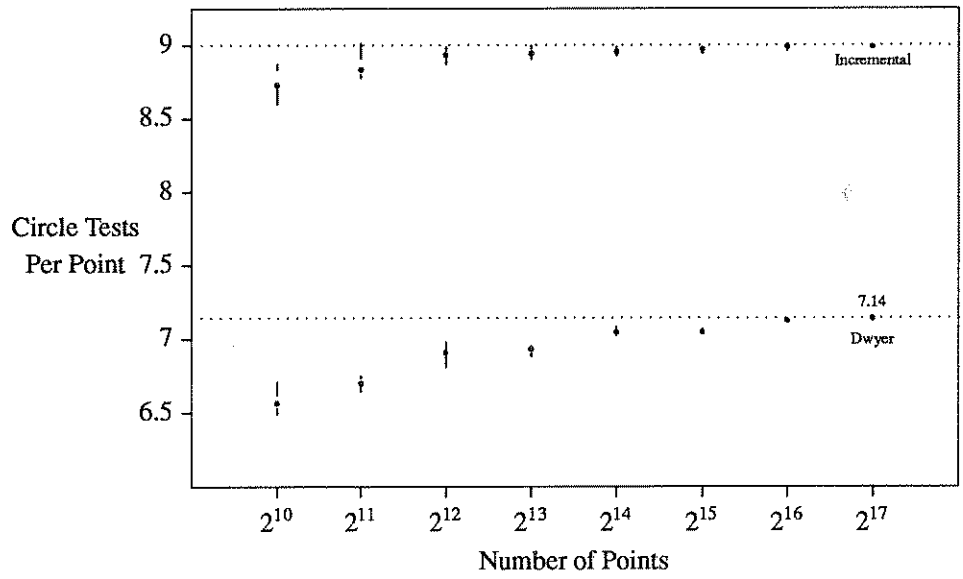


Figure 9: Circle tests per site for two algorithms.

triangulation that cross a particular area of the unit square. Also, the behavior of the algorithm on non-uniform point sets is suspect. Luckily, such input should only slow down the point location algorithm. Since this is now relatively small percentage of the total runtime, using more adaptive bucketing methods should be sufficient to keep those costs from dominating the rest of the algorithm. Also, it should be possible to extend the algorithm to handle more exotic distance functions, but that would involve working out many tricky details.

6.2. The Incremental Algorithm with a Quad-Tree

Ohya, Iri and Murota [16] describe a modification to the incremental algorithm which buckets the points like my algorithm does but then inserts the points using a breadth-first traversal of a quad-tree. In their paper, the authors claim that their algorithm runs in linear time on average, and they provide experimental evidence for this fact. Their analytical results are suspect, however. In order to show that my algorithm was competitive with, or better than other similar methods, I implemented this algorithm and compared its performance with mine.

Like before, a profile of this program showed that the algorithm spends most of its time either testing circles or locating points. Therefore, I monitored these two operations in more detail. My experiments showed that the quad-tree algorithm performs almost identically to my algorithm, except that it does more edge tests in its point location phase. As figure 10 shows, the quad-tree algorithm performs about 20% more edge tests than my algorithm. For the uniform case, we can conclude that my algorithm performs slightly better than this more complicated alternative.

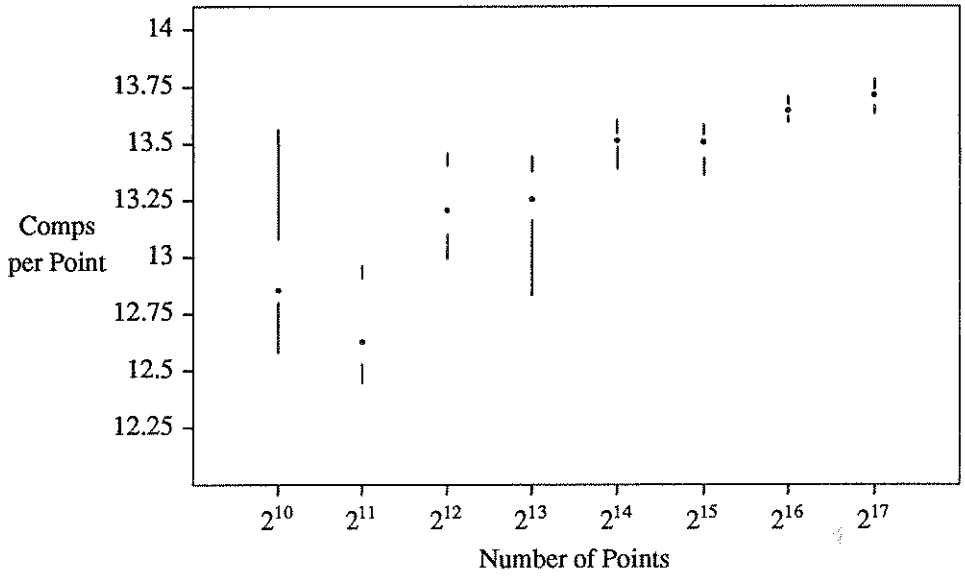


Figure 10: Point location cost for the quad-tree algorithm.

6.3. Fortune's Algorithm

The runtime of Fortune's algorithm is proportional to the cost of searching and updating the data structures representing the event queue and the state of the sweepline. Fortune's implementation uses hash tables for this purpose. We would expect that these data structures would perform well on random inputs. In fact, for small input sets, the algorithm seems to run in linear time.

Figure 11 shows the performance of the sweepline and priority queue data structures in Fortune's implementation. With sites that are uniformly distributed in the x direction, the bucket structure representing the frontier performs exactly as we would expect. Figure 11a indicates that the search procedure performs around 12 comparisons per point, on average.

The main bottleneck in Fortune's algorithm ends up being the maintenance of the priority queue. The priority queue is represented using a uniform array of buckets in the y direction. Events are hashed according to their y-coordinate and placed in the appropriate bucket. The problem here is that while the sites are uniformly distributed, the resulting priorities are not. Circle events tend to cluster in the upper part of the bucket array and in the buckets that are close to the current position of the sweepline. This clustering increases the cost of searches in the priority queue. Regression analysis shows that the number of comparisons per point grows as $9.95 + .25\sqrt{n}$ (see figure 11b).

Given the behavior of the bucket data structure, it is natural to speculate as to whether a tree-based data structure would provide better performance for larger problems. To investigate this question, I instrumented the implementation of Fortune's algorithm to give more detailed information about the behavior of the queue data structure.

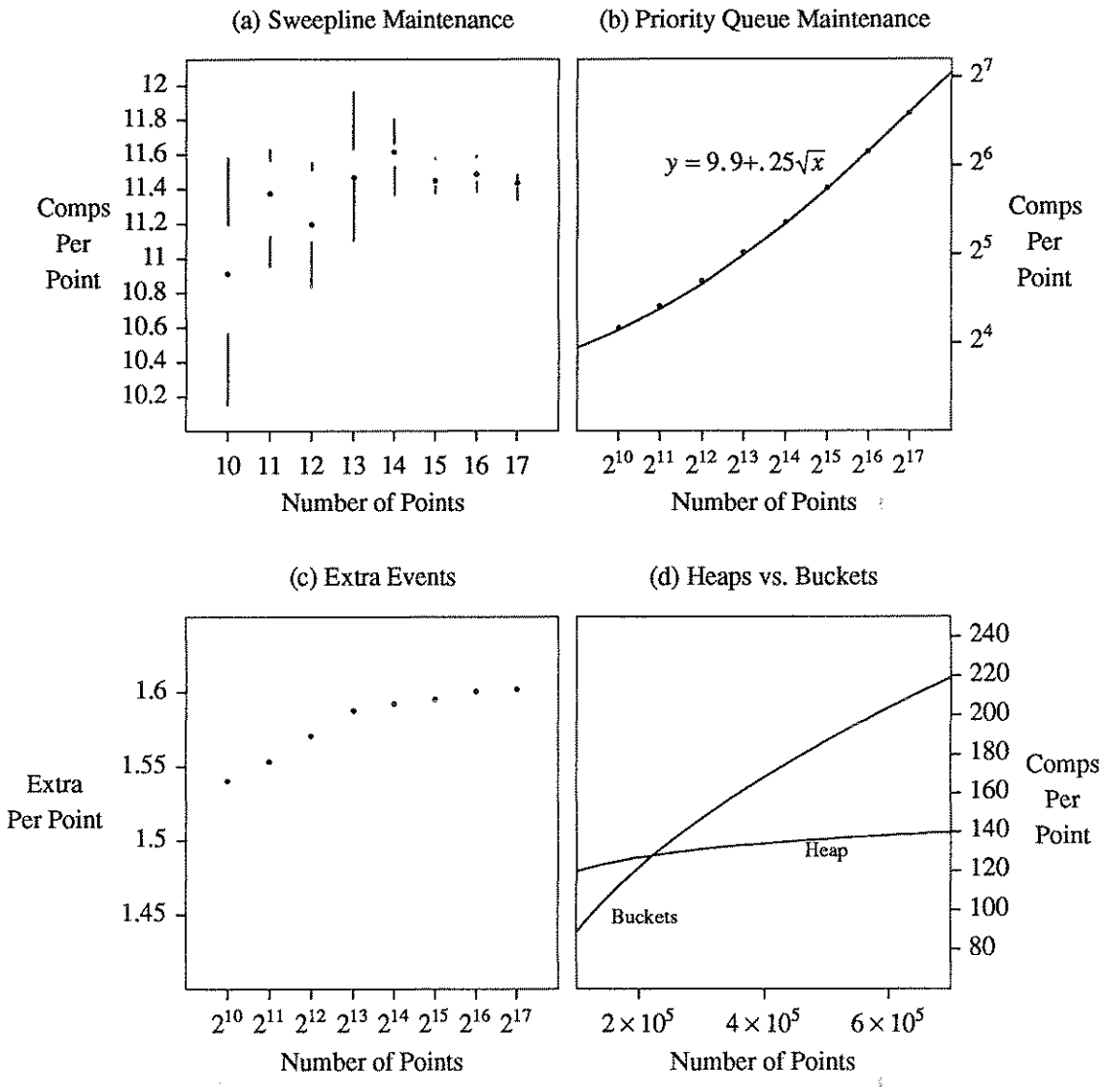


Figure 11: Cost of Fortune's algorithm.

Fortune's algorithm performs two queue operations for each vertex in the Voronoi diagram. Since the number of vertices is roughly $2n$, the total number of queue operations is at least $4n$ on average. However, Fortune's algorithm inserts some extra events into the priority queue which later become invalid. The number of extra events that get inserted is difficult to analyze. But, it is easy to make the program count such events because the algorithm invalidates them as they occur. Monitoring Fortune's program gave the results shown in figure 11c, which plots surplus event insertions as a function of n.

Based on the experimental data, we can conjecture that the average number of extra events is bounded by $1.6n$. Thus, the total number of queue operations should be around $7.2n$. If this is the true constant, then we can plot $7.2n \log_2 n$ against $.25\sqrt{n}$ to find out when switching to a heap structure would pay off. Figure 11d shows the results of this comparison. We see that in terms of total comparisons, the heap structure will begin to pay off at around 2×10^5 points. This could be an overestimate, since we can use a simple, low overhead array implementation of the heap. But, given this data, we can conclude that for all but the largest problems, using a heap won't save a significant amount of time.

6.4. The Bottom Line

The point of all of this is, of course to develop an algorithm that has the fastest overall run time. I gathered run times on a Sparcstation 2 using the `getrusage()` mechanism in UNIX I measured user time, not real time, and I did not include system time in the measurements. I only ran tests that did could fit in main memory, and I generated the test sets in main memory so I/O and paging would not affect the results.

Figure 12 shows the results of my timing experiments. The graph shows that Dwyer's algorithm gives the best performance overall. Fortune's algorithm and the improved incremental algorithm are about the same, but for large problems the incremental algorithm does better. Fortune's algorithm does the worst on the largest problems due to the problems with its priority queue data structure.

Dwyer's algorithm is faster because it does almost 1/3 fewer circle tests than the incremental algorithm. Profiling information shows that each algorithm spends about half of it's total time in circle testing, so this accounts for the fact that Dwyer's program is about ten to fifteen percent faster than the incremental. However, although the incremental algorithm is marginally slower than Dwyer's, it is much simpler to code. It would be interesting to see if the number of circle tests that the incremental algorithm does could be reduced, though this is likely to be difficult. In the meantime, I would claim that the simplicity and "on-line" nature of the incremental algorithm outweighs its small performance disadvantage.

Finally, the quad-tree order incremental algorithm is slower than my algorithm and Dwyer's algorithm, but is somewhat faster than the others. Again, the main difference between my algorithm and the quad-tree algorithm is that my point location routine is faster. This accounts for the fact that my algorithm is roughly 15% faster than the quad-tree algorithm.

7. Pathological Point Sets

Each of the algorithms that we have studied uses a uniform distribution of points to its advantage in a slightly different way. The incremental algorithm uses the fact that nearest neighbor search is fast on uniform point sets to speed up point location. Dwyer's algorithm uses the fact that Delaunay edges tend to be short to speed up its merge steps. Fortune's implementation uses bucketing to search the sweepline and to maintain its priority queue.

From the analysis and experimental experience we gained previously, we would expect that since it can do no more than $O(n \log n)$ operations, Dwyer's algorithm would be the least sensitive to pathological input. Next in line would be the incremental algorithm which would perform $O(n^{3/2})$ operations on average if the distribution of the sites totally defeated the

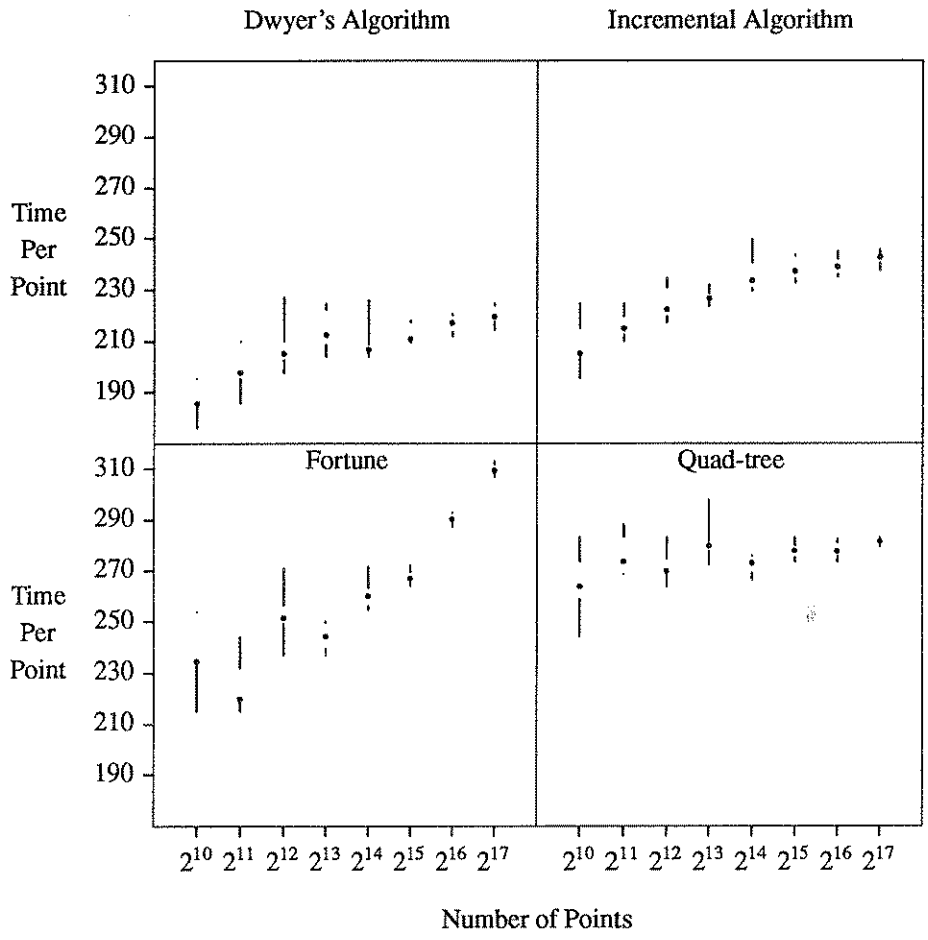


Figure 12: Comparison of different algorithms. Times are in microseconds.

bucketing strategy

Finally, we have already seen that Fortune's heap implementation is not efficient for large problems even when the input is uniform. If the input set were extremely irregular, so that most of the priorities fell into just a few buckets, we would expect the performance of the implementation to be even worse than the incremental algorithm.

In order to investigate the sensitivity of each algorithm to non-uniform point distributions, I ran further tests on some extremely bad point sets. Two distributions generated particularly interesting results. The first distribution was generated using the formulas $x = 0.5 + o_x$ and $y = 0.5 + o_y$ where o_x and o_y are chosen from a uniform distribution in the interval $[-t/2, t/2]$ for some $t > 0$. The algorithm generates x and y, then flips a coin and returns either $(0.5, y)$ or $(x, 0.5)$.

The second has the points distributed in one tight cluster. These point sets were generated by choosing a center point p and then choosing the other points using the formula $x = p_x + o_x$

and $y = p_y + o_y$ where o_x and o_y have an exponential distribution $f(s) = (1/t)\, e^{-s/t}$. Figure 13 shows examples of each of these point distributions for $t = .01$.

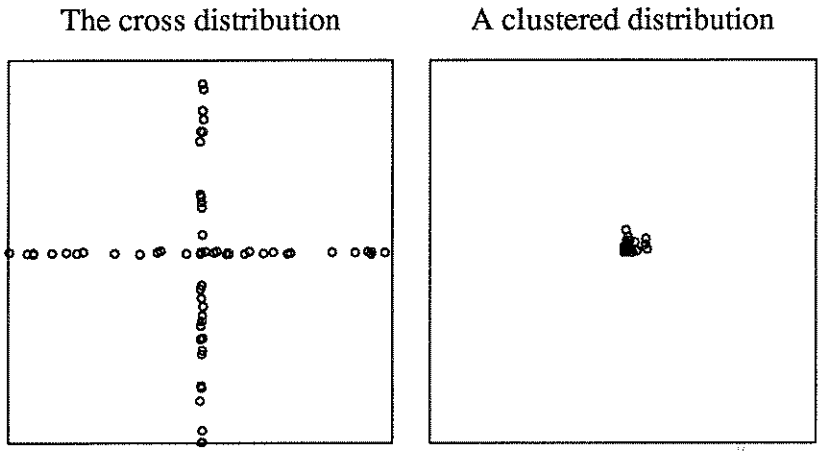


Figure 13: Bad point distributions

As before, I performed ten runs of each algorithm for $n = 1024, 2048, \cdots, 65536$. I only tested the incremental algorithm at powers of four to smooth out the response. This is because the saw-tooth behavior that is apparent in figure 7 was even more pronounced. In the text below, the "incremental algorithm" may refer to either my algorithm or the quad-tree algorithm. As before, the performance of the two algorithms is almost the same, with the bucketing algorithm being a small amount faster.

The incremental algorithm is fairly robust as long as there is some spread in the point distribution. The algorithm slows down on point sets with heavy clustering because this makes the point location algorithm perform badly. Figure 14 shows the performance of the point location routine on the two point sets described above. The incremental algorithm does surprisingly well on the "cross" distribution. The random input order keeps it from doing too many circle tests, and the bucketing data structure is still effective here because the points are spread out along the axes. Since there are $\sqrt{n}$ buckets along each axis, we would expect that the cost of the point location algorithm would be $O(n^{\varepsilon})$ per site where $\varepsilon \approx 1/4$. Figure 14a shows that the algorithm actually does exhibit this behavior. The curve shown in the figure is $y = 4.33 + 1.59n^{.25}$, which is a reasonable fit to the data. The residuals in this case are a bit erratic, especially for small point sets.

Although the performance of the point location algorithm is somewhat worse in this case than in the uniform case, we should remember that the tests performed in a point location steps are relatively cheap compared to an `in-circle` test, thus the effect on the actual run-time of the algorithm isn't as great as it might seem. In fact, for the range of input sizes that I studied, the incremental algorithm was actually faster than Dwyer's on this kind of input.

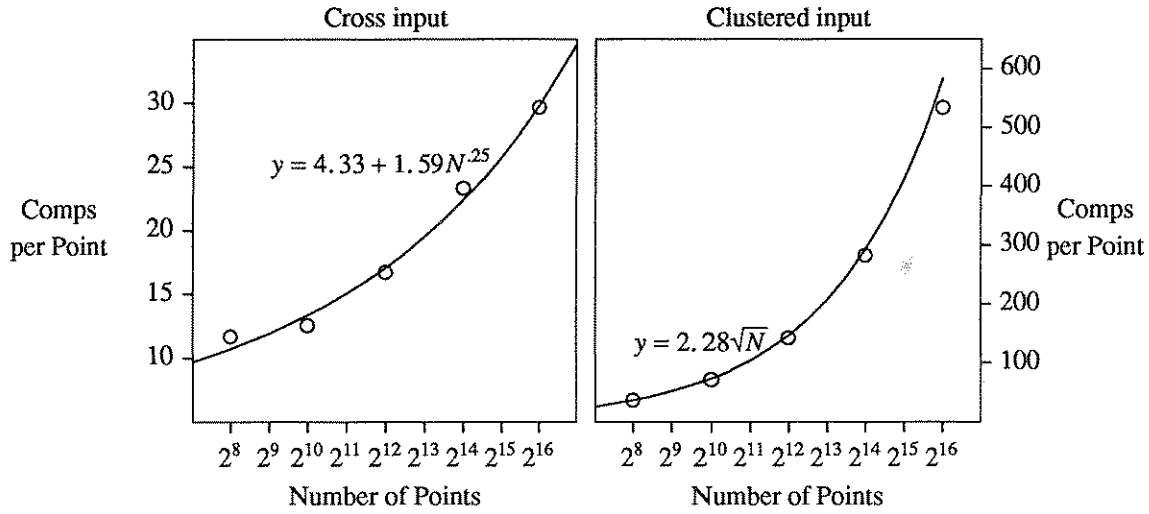


Figure 14: Performance of point location on bad inputs.

On the cluster distribution, the incremental algorithm does $O(\sqrt{n})$ comparisons per point. The curve shown in figure 14b is $2.28\sqrt{n}$, and is a fairly good fit to the data. The residuals in the fit were not as well behaved as we might hope, but it seems safe to assume that the number of comparisons per site is $O(\sqrt{n})$.

Finally, although we have seen that the incremental algorithm can do very badly on extremely non-uniform input, we haven't examined a large range of "badness." To do this, I ran a final set of tests on data made up of clusters of sites generated using the formulas $x = p_x^i + o_x$ and $y = p_y^i + o_y$ where o_x and o_y are normally distributed and p_x^i and p_y^i are chosen at random for $i = 1, 2, \ldots, 10$. I ran ten tests each for n between 1024 and 65536. To vary the degree of clustering, I ran three sets of tests with the normal distributions

Figure 15b shows the cost per site (in microseconds) of the algorithm throughout this range of inputs. These scales represent a large range of "badness," and figure 15b shows that the run time of the incremental algorithm is still close to its average throughout the whole range.

The other two algorithms do most poorly on the cross distribution. In particular, this distribution of points makes the priority queue structure in Fortune's implementation perform extremely badly. What happens is that when the sweepline reaches the sites along the x axis, the event queue fills up with events that cluster heavily in the y direction. Most of these events fall into a few buckets in the priority queue structure and causes the searches in the structure cost close to $O(n)$ operations each. Figure 16a shows the behavior of the priority queue structure on this input when $t = .001$. This behavior is another possible reason to replace the Fortune's bucket structure with a heap.

The cross distribution is also close to the worst-case input for Dwyer's algorithm. Of course, Dwyer's algorithm has a worst case runtime of $O(n \log n)$ operations, so it should still do relatively well. Figure 16b shows how many circle tests the algorithm performs as a function of

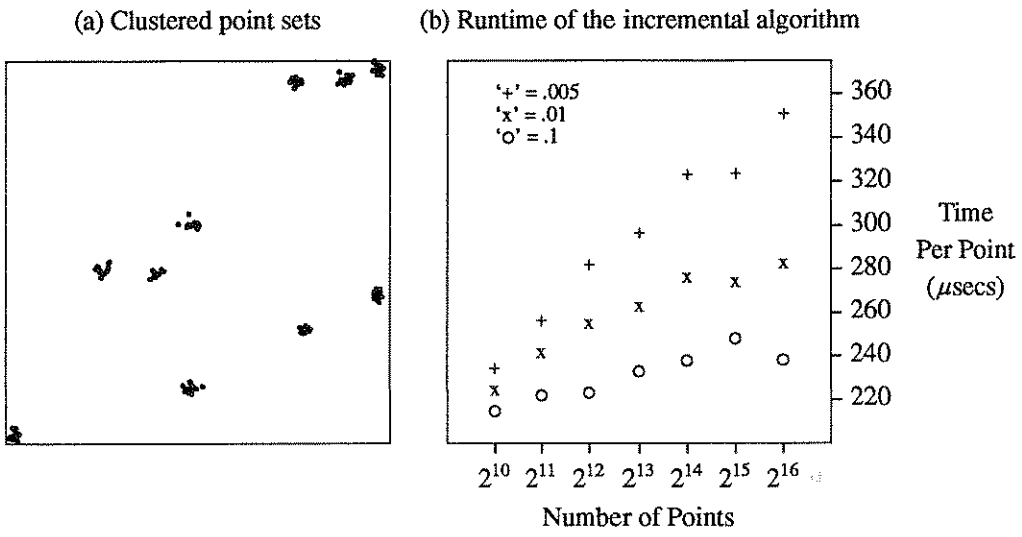


Figure 15: Performance of the incremental algorithm on clusters.

the input size. The tests were run with $t = .001$. Circle testing is still the dominant cost in the algorithm, and figure shows that the algorithm is doing close to $O(\log n)$ tests per site. The reason is because while the algorithm has the same linear runtime inside the horizontal and vertical strips, it has no advantage over the normal divide-and-conquer algorithm when building the edges which cross between the regions. Thus, as the strips become smaller and smaller, the performance of the algorithm converges on the worst case.

8. Notes and Discussion

We have seen that that a simple modification to the incremental algorithm for constructing the Delaunay triangulation yields a method that runs in linear expected time for uniformly distributed points. The modification uses a simple bucket-based data structure to perform near neighbor searches. This speeds up the point location phase of the incremental algorithm without the complex tree-based data structures that other algorithms use. While substantially simpler than most other algorithms for this problem, it still performs as well or better than the competition. Experimental evidence indicates that the algorithm will perform poorly only on extremely pathological (and contrived) inputs.

There are many published algorithms that I did not examine. I did not consider the divide and conquer algorithm discussed by Guibas and Stolfi because other empirical studies have already shown it to be substantially slower than the algorithms presented in this paper. I also did not implement the algorithm of Guibas, Knuth and Sharir [12], because it is likely that their point location scheme is substantially slower than mine on the inputs that we considered. I also did not look at some of the known linear expected-time algorithms, because the papers which described them contain empirical results or because no implementation of the algorithms were available for

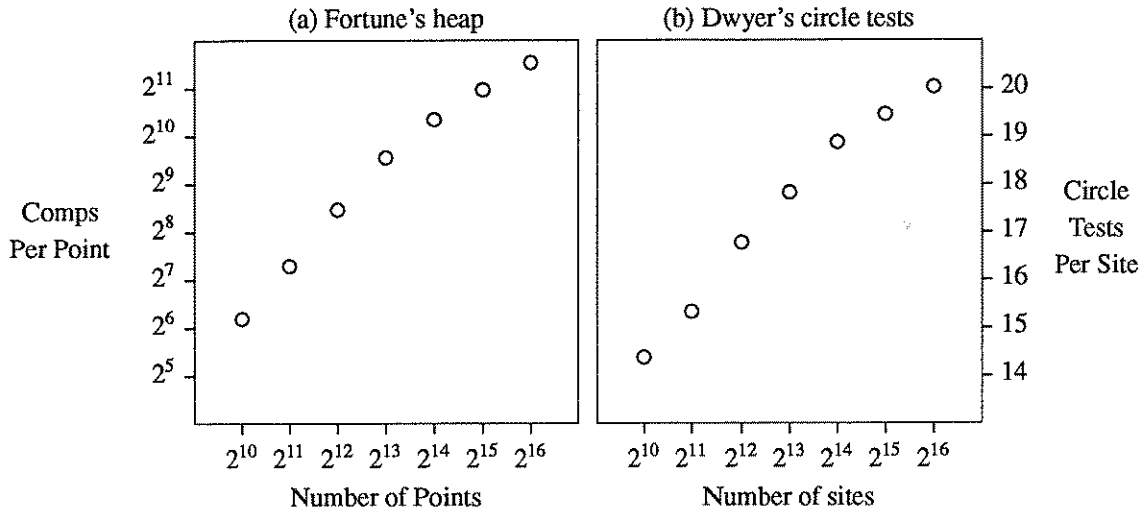


Figure 16: The performance of Fortune's algorithm and Dwyer's algorithm on the cross input.

study [13,14].

The real purpose of this paper, however, is not to advertise a new and superior algorithm, but to illustrate some principles in the design and analysis of algorithms. First, a variety of tools are available for doing in-depth experimental analysis of algorithms and programs. For example, animation can be extremely useful for understanding algorithms and debugging code. Concentrating on simple displays that are easily generated keeps the programmer from becoming overly distracted, while still producing useful information. Bentley [2] and McGeoch [15] have discussed other tools useful for performance analysis. These include data analysis programs, profilers, data processing tools like Awk or Perl, and custom programs based on "little languages" for many tasks including generating experiments, graphing data, or displaying animations.

Using these tools, there is much more to experimental analysis than just runing benchmarks. Only one of the graphs that I showed in this paper has anything to do with the actual run-time of these algorithms on a real machine. For the most part my experiments have concentrated on abstract costs which dominate the execution time of the algorithm in a way which is machine independent.

Using these experiments, and some knowledge as to how much critical primitive operations (such as distance computations, or in-circle tests) will cost on a given machine, we can model the real execution time of the program with high accuracy. In addition, we can do this without the burden of esoteric computational models that attempt to cover every possible variable in the design of a computer system. Thus, I take a very pragmatic view of the world. High level abstract analysis should guide the use of experiments and empirical models to accurately predict performance. Neither theory nor experiments are more important to me, both have equal weight.

References

1. J. L. BENTLEY, B. W. WEIDE, AND A. C. YAO, "Optimal Expected Time Algorithms for Closest Point Problems," *ACM Transactions on Mathematical Software,* 6, 4, pp. 563-580 (1980).

2. J. L. BENTLEY, "Experiments on Travel Salesman Heuristics," *ACM-SIAM Symposium on Discrete Algorithms,* pp. 91-99 (1989).

3. K. L. CLARKSON AND P. W. SHOR, "Applications of Random Sampling in Computational Geometry, II," *Discrete and Computational Geometry,* 4, pp. 387-421 (1989).

4. K. L. CLARKSON, K. MEHLHORN, AND R. SEIDEL, "Four Results on Randomized Incremental Construction," *Annual Symposium on Theoretical Aspects of Computer Science* (1992).

5. O. DEVILLERS, S. MEIDER, AND M. TELLAUD, *Fully Dynamic Delaunay Triangulation in Logarithmic Expected Time per Operation,* Tech. report 1349, INRIA (1990).

6. R. A. DWYER, *Average-case Analysis of Algorithms for Convex Hulls and Voronoi Diagrams,* Ph.D. Thesis, CMU, Pittsburgh, PA (1988).

7. R. A. DWYER, "A Faster Divide-and-Conquer Algorithm for Constructing Delaunay Triangulations," *Algorithmica,* 2, pp. 137-151 (1987).

8. S. FORTUNE, "A Sweepline Algorithm for Voronoi Diagrams," *Algorithmica,* 2, pp. 153-174 (1987).

9. S. FORTUNE, "Numerical Stability of Algorithms for Delaunay Triangulations and Voronoi Diagrams," *Annual ACM Symposium on Computational Geometry* (1992).

10. S. FORTUNE, "Stable Maintenance of Point-Set Triangulations in Two Dimensions," *IEEE Symposium on Foundations of Computer Science,* pp. 494-499 (1989).

11. L. GUIBAS AND J. STOLFI, "Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams," *ACM Transactions on Graphics,* 4, 2, pp. 75-123 (1985).

12. L. GUIBAS, D. KNUTH, AND M. SHARIR, "Randomized Incremental Construction of Delaunay and Voronoi Diagrams," *ICALP,* pp. 414-431 (1990).

13. J. KATAJAINEN AND M. KOPPINEN, *Constructing Delaunay Triangulations by Merging Buckets in Quad-tree Order,* p. Unpublished manuscript (1987).

14. A. MAUS, "Delaunay Triangulation and the Convex Hull of n points in expected linear time," *BIT,* 24, pp. 151-163 (1984).

15. C. MCGEOCH, *Experimental Analysis of Algorithms,* PhD Thesis, CMU (1986).

16. T. OHYA, M. IRI, AND K. MUROTA, "Improvements of the Incremental Method for the Voronoi Diagram with Computational Comparison of Various Algorithms," *Journal fo the Operations Research Society of Japan,* 27, pp. 306-337 (1984).

17. F. PREPERATA AND M. I. SHAMOS, *Computational Geometry: An Introduction,* Springer-Verlag, New York (1985).

18. M. SHARIR AND E. YANIV, "Randomized Incremental Construction of Delaunay Diagrams: Theory and Practice," *Annual ACM Symposium on Computational Geometry* (1991). Submitted.