A Comparison of Sequential Delaunay Triangulation Algorithms

Peter Su
Imperative Internet Technologies
305 S. Craig Steet
Pittsburgh, PA 15213
psu@tig.com

Robert L. Scot Drysdale*
Department of Computer Science
6211 Sudikoff Laboratory
Dartmouth College, Hanover NH 03755-3510, USA
scot.drysdale@dartmouth.edu

April 1, 1996

Abstract

This paper presents an experimental comparison of a number of different algorithms for computing the Deluanay triangulation. The algorithms examined are: Dwyer's divide and conquer algorithm, Fortune's sweepline algorithm, several versions of the incremental algorithm (including one by Ohya, Iri, and Murota, a new bucketing-based algorithm described in this paper, and Devillers's version of a Delaunay-tree based algorithm that appears in LEDA), an algorithm that incrementally adds a correct Delaunay triangle adjacent to a current triangle in a manner similar to gift wrapping algorithms for convex hulls, and Barber's convex hull based algorithm.

Most of the algorithms examined are designed for good performance on uniformly distributed sites. However, we also test implementations of these algorithms on a number of non-uniform distibutions. The experiments go beyond measuring total running time, which tends to be machine-dependent. We also analyze the major high-level primitives that algorithms use and do an experimental analysis of how often implementations of these algorithms perform each operation.

1. Introduction

Sequential algorithms for constructing the Delaunay triangulation come in five basic flavors: divide-and-conquer [8, 17], sweepline [11], incremental [7, 15, 17, 16, 20], growing a triangle at a time in a manner similar to gift wrapping algorithms for convex hulls [9, 19, 25], and lifting the sites into three dimensions and computing their convex hull [2]. Which approach is best in practice? This paper presents an experimental comparison of a number of these algorithms. Many of these algorithms were designed for good performance on uniformly distibuted sites rather than good worst-case performance, but implementations of these algorithms are also tested on a number of highly non-uniform distributions.

In addition, we describe a new version of the incremental algorithm that is simple to understand and implement, but is still competitive with the other, more sophisticated methods on a wide range of problems. The algorithm uses a combination of dynamic bucketing and randomization to achieve both simplicity and good performance.

The experiments in this paper go beyond measuring total run time, which is highly dependent on the computer system. We also analyze the major high-level primitives that algorithms use and do an experimental analysis of how often implementations of these algorithms perform each operation.

The rest of this section briefly describes the various algorithmic approaches. More detailed descriptions of the algorithms, including pseudocode, can be found in Chapter 2 of the first author's Ph.D. thesis [24].

*This study was supported in part by the funds of the National Science Foundation, DDM-9015851, and by a Fulbright Foundation fellowship.

1.1. Divide-and-Conquer

Guibas and Stolfi [17] gave an $O(n \log n)$ Delaunay triangulation algorithm that is asymptotically optimal in the worst case. The algorithm uses the quad-edge data structure and only two geometric primitives, a CCW orientation test and an in-circle test. These primitives are defined in terms of 3 by 3 and 4 by 4 determinants, respectively. Fortune [12, 13] shows how to compute these accurately with finite precision.

Dwyer [8] showed that a simple modification of this algorithm runs in $O(n \log\log n)$ expected time on uniformly distributed sites. Dwyer's algorithm splits the set of sites into vertical strips with $\sqrt{n / \log n}$ sites per strip, constructs the DT of each strip by merging along horizontal lines, and then merges the strips together along vertical lines. His experiments indicate that in practice this algorithm runs in linear expected time. Another version of this algorithm, due to Katajainen and Koppinen [18], merges square buckets together in a "quad-tree" order. They show that this algorithm runs in linear expected time for uniformly distributed sites. In fact, their experiments show that the performance of this algorithm is nearly identical to Dwyer's.

1.2. Sweepline Algorithms

Fortune [11] invented another $O(n \log n)$ 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 called the *frontier* of the diagram. These edges are a subset of the Delaunay diagram, and form a tour around the outside of the incomplete triangulation. In addition, the algorithm uses a priority queue, called the *event queue* to keep track of places where the sweep line should stop. This queue stores two types of events called *circle* events and *site* 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 adjacent vertices on the frontier. The algorithm sweeps a line up in the y direction, processing each event that it encounters.

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 bucketed 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.

1.3. Incremental Algorithms

The third, and perhaps simplest class of algorithms for constructing the Delaunay triangulation consists of incremental algorithms. These algorithms add sites to the diagram one by one and update the diagram after each site is added. They have two basic steps. The first, `Locate`, finds the triangle containing the new site. (The algorithms are made simpler by assuming that the sites are enclosed within large triangle.) The second, `Update`, updates the diagram.

All of the algorithms perform `Update` using an approach similar to that in Guibas and Stolfi [17], flipping edges until all edges invalidated by the new site have been removed. Each edge is tested for validity via an in-circle test. In the worst case we must do $O(n^2)$ in-circle tests and edge flips, because it is possible to construct a set of sites and insertion order where inserting the k^{th} site into the diagram causes $\Theta(k)$ updates. However, if the sites are inserted in a random order, Guibas, Knuth, and Sharir [16] show that the expected number of edge flips is linear no matter how they are distributed.

Therefore the bottleneck in the algorithm is the `Locate` routine. Guibas and Stolfi start at a random edge in the current diagram and walk across the diagram in the direction of the new site until the correct triangle is found. The basic step is to perform a CCW orientation step against an edge of a triangle to see if the site lies on the correct side of that edge. If not, the algorithm crosses to the other triangle that shares that edge.

If so, it steps to the next edge around the triangle. When the site is on the correct side of all three edges in a triangle it has been located. This search is expected to perform $O(\sqrt{n})$ CCW orientation tests per `Locate`.

Ohya, Iri and Murota [20] bucket the sites and process the buckets in the order of a breadth-first traversal of a quad-tree. (They do not actually build such a tree, but simply insert sites in that order.) They start a Guibas and Stolfii-like `Locate` at an edge created by the previous insertion. The quad-tree traversal order means that this edge is likely to be close to the correct triangle. They claim that their algorithm runs in expected linear time on sites that are uniformly distributed, and they provide experimental evidence for this fact.

Guibas, Knuth, and Sharir propose a tree-based data structure where internal nodes are triangles that have been deleted or subdivided at some point in the construction, and the current triangulation is stored at the leaves. A step in the locate algorithm consists of going from a triangle containing the site at one level to one of a constant number of triangles that might contain the site at the next level. It is not hard to show that the total expected cost of `Locate` will be $O(n \log n)$ time. Sharir and Yaniv [23] prove a bound of about $12nH_n + O(n)$. This structure is similar to the Delaunay tree described by Boissonnat and Teillaud [5].

1.4. A Faster Incremental Construction Algorithm

We present a `Locate` variant that leads to an easily implemented incremental algorithm that seems to perform better than those mentioned above when the input is uniformly distributed. We use a simple bucketing algorithm similar to the one that Bentley, Weide and Yao used for finding the nearest neighbor of a query point [4]. This leads to an $O(n)$ time algorithm 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. To find a near neighbor, the point location algorithm first finds the bucket that contains the query site and searches in an outward spiral for a non-empty bucket. It then uses any edge incident on the site in this bucket as a starting edge in the Guibas and Stolfi `Locate` routine. If the spiral search fails to find a site after a certain number of layers, the point location routine starts the Guibas and Stolfi `Locate` routine from an arbitrary edge in the current triangulation.

The bucketing scheme uses a dynamic table to deal with the fact that sites are processed in an on-line fashion. The scheme does not bother to store all the sites that fall in a particular bucket, but just stores the last site seen. This is because the bucket structure does not have to provide the insertion algorithm with the true nearest neighbor of a query. It only has to find a site that is likely to be close to the query. Therefore, it makes sense not to use the extra space on information that we do not need. Let $c > 0$ be some small constant that we can choose later. The point location maintains the bucket table so that on average between c and $4c$ sites fall into each bucket. It does this on-line by quadrupling the size of the bucket grid and re-bucketing all sites whenever the average goes above $4c$.

It is not hard to show that the expected cost of a `Locate` is $O(1)$ and the total cost of maintaining the buckets is $O(n)$ if the floor function is assumed to be constant time. Thus, the total expected run time of this algorithm is $O(n)$ time when the sites are uniformly distributed in the unit square.

1.5. Gift Wrapping Algorithms

Another class of Delaunay triangulation algorithms constructs the Delaunay triangulation by starting with a single Delaunay triangle and then incrementally discovering valid Delaunay triangles, one at a time. Each new triangle is grown from an edge of a previously discovered triangle by finding the site that joins with the endpoints of that edge to form a new triangle whose circumcircle is empty of sites. Several algorithms in the literature, including ones due to Dwyer [9], Maus [19], and Tanemura et. al. [25] are based on this basic idea. They all differ in the details, and only Dwyer's algorithm has been formally analyzed. Dwyer's analysis assumed that the input was uniformly distributed in the unit d-ball, and extending this analysis to the unit cube appears to be non-trivial. However, in the plane the difference is not great, and the experiments

in the next section will show that the empirical run time of the algorithm appears to be $O(n)$.

The basic approach is to take an edge (a, b) of a triangle on the border of the part of the diagram constructed so far and to choose a candidate site c for the third vertex of this triangle. Other sites are tested to see if they fall within the circumcircle of triangle abc. If one does, it becomes the new candidate site c. When a c is found such that the circumcircle of abc contains no other sites then abc is added to the Delaunay triangulation. Note that sites lying far enough from the center of the circumcircle of abc need not be explicitly tested, and "far enough" is just the radius of this circumcircle. Because of this it can be useful to compute the center and radius of the current circumcircle. Given these it is faster to perform the in-circle test by computing the distance from a site to this center and comparing it to the radius than using the determinant method described above. (Actually, the square of the distance is computed to save computing a square root.)

The trick is to examine sites in an order that allows the correct candidate can be found and verified quickly, ideally after only a constant number of sites have been considered. Therefore the data structure that is critical to the performance of this algorithm is the one that supports site searches. Dwyer's algorithm [9] uses a relatively sophisticated algorithm to implement site searching. First, the sites are placed in a bucket grid covering the unit circle. The search begins with the bucket containing the midpoint of (a, b), where (a, b) is an edge of a current triangle that does not yet have a triangle in its other side. As each bucket is searched, its neighbors are placed into a priority queue that controls the order in which buckets are examined. Buckets are ordered in the queue according to a measure of their distance from the initial bucket. Buckets are only placed in the queue if they intersect the half-plane to the right of (a, b) and if they intersect the unit disk from which the sites are drawn. Dwyer's analysis shows that the total number of buckets that his algorithm will consider is $O(n)$.

The site search algorithm in our gift wrapping algorithm is a simple variant on "spiral search" [4]. It differs from Dwyer's in several ways. The main difference is the lack of a priority queue to control the action of the search routine. Our spiral search approximates this order, but it is not exactly the same. Dwyer's algorithm is also careful not to inspect any buckets that are either to the left of (a, b) or outside of the unit circle. Our algorithm is somewhat sloppier about looking at extra buckets. The advantages of our scheme are that it is well tuned to the case when sites are distributed in the unit *square* and it avoids the extra overhead of managing a priority queue, especially avoiding duplicate insertions.

1.6. Convex Hull Based Algorithms

Brown [6] was the first to establish a connection between Voronoi diagrams in dimension d and convex hulls in dimension $d + 1$. Edelsbrunner and Seidel [10] later found a correspondence between Delaunay triangles of a set of sites in dimension 2 and downward-facing faces of the convex hull of those sites lifted onto a paraboloid of rotation in dimension 3. If each site (x_i, y_i) is mapped to the point $(x_i, y_i, x_i^2 + y_i^2)$, the convex hull of these lifted points is computed, and the the upward-facing convex hull faces are discarded, what is left is the Delaunay triangulation of the original set of sites. (This correspondence is true for arbitrary dimension d.)

Guibas and Stolfi [17] note that their divide and conquer algorithm for computing the Deluanay triangulation can be viewed as a variant of Preparata and Hong's algorithm for computing three dimensional convex hulls [21]. Others have also used this approach. Recently Barber [2] has developed a Delaunay Triangulation algorithm based on a convex hull algorithm that he calls Quickhull. He combines the 2-dimensional divide-and-conquer Quickhull algorithm with the general dimension Beneath-beyond algorithm to obtain an algorithm that works in any dimension and is provably robust.

2. Empirical Results

In order to evaluate the effectiveness of the algorithms described above, we studied C implementations of each algorithm. Rex Dwyer provided code for his divide and conquer algorithm, and Steve Fortune

Name	Description
Dwyer	Dwyer's divide and conquer algorithm.
Fortune	Fortune's sweepline algorithm.
Inc	Guibas and Stolfi's naive incremental algorithm.
BucketInc	The bucketed incremental algorithm presented in this paper.
QtreeInc	Ohya, Iri, and Murota's incremental using quad-tree insertion order.
Giftwrapping	The gift wrapping algorithm described in this paper.
Qhull	Barber's algorithm that uses the Quickhull convex hull algorithm.
Dtree	Devillers's implementation of Delaunay Tree code.
LEDA	The version of Delaunay tree code implemented in LEDA.

Table 1: List of the algorithms examined.

provided code for his sweepline algorithm. We implemented the incremental algorithms and the gift wrap algorithm. Olivier Devillers provided a version of his Delaunay tree-based code, and another version of his code was available in the programming package LEDA developed at the Max-Plank-Institut für Informatik in Saarbrücken. Because the LEDA version could have more overhead we tested both. The LEDA code is available at `ftp.mpi-sb.mpg.de`. (We have been told that there are plans to replace this algorithm in the LEDA code sometime in the future.) For the convex-hull based algorithm we used Brad Barber's code, which is available via internet at `ftp://geom.umn.edu/pub/software/qhull.tar.Z`. None of the implementations are tuned in any machine dependent way, and all were compiled using the GNU C or C++ compiler and timed using the standard UNIX^{tm} timers. (The Delaunay tree and LEDA algorithms are in C++.) See Table 1 for a list of the algorithms considered with short names that will be used to refer to them in the rest of this paper.

Some notes on implementations are in order. First, all numerical primitives use floating point computations. Using integer or exact arithmetic could greatly change the relative run times. Beyond using Fortune's stable in-circle test we did little to handle degeneracies and numerical instabilities.

We tried to share primitives and data structures when this was possible. Dwyer was modified to use Fortune's stable in-circle test, which is somewhat more expensive than the standard test. All of the incremental algorithms use this same stable in-circle test. All incremental algorithms also use a version of the quad-edge data structure like the one that Dwyer implemented for his divide-and-conquer algorithm. This array-based structure proved to be substantially faster than a pointer-based structure that we used earlier. Fortune was not modified, but does not use either in-circle tests or the quad-edge data structure. Giftwrapping does not use the quad-edge data structure. As was mentioned above it uses an in-circle test using the radius of the circumcircle rather than computing a determinant. All of these algorithms use the same CCW orientation test.

Qhull, Dtree, and LEDA were added after the initial series of experiments. The run times of the unmodified programs were not fast enough to make re-writing them to share data structures and primitives worth the effort. These run times will be discussed later.

The performance data presented in this section was gathered by instrumenting the programs to count certain abstract costs. A good understanding of each algorithm, and profiling information from test runs determined what was important to monitor. Each algorithm was tested for point set sizes of between 1,024 and 131,072 sites. Ten trials with sites uniformly distributed in the unit square 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.

2.1. Performance of the Incremental Algorithms and Dwyer

The cost of an incremental algorithm is the sum of the cost of point location plus the cost of updating the diagram once the correct triangle is found. The update portion is dominated by the in-circle test, with the rest of the time going to a CCW orientation test, quad-edge data structure manipulation and loop maintainance. The loop structure is such that the amount of other work is directly proportional to the number of in-circle tests, so by counting in-circle tests we can get a number that lets us compare the amount of update work performed by the various algorithms. (Because of shortcut evaluation it is possible for the CCW evaluation to be done without the in-circle test being performed, but this happens very rarely. In our tests this happened a bit less than 0.3% of the time for 1024 sites, but had fallen off to 0.003% of the time for 131,072 sites.)

For all of the incremental algorithms except Dtree the main cost of point location is the walk across the diagram once an edge has been selected. A CCW orientation step is performed each time through the loop that performs this walk, and the number of calls to quad-edge manipulation routines and other overhead in the walk across the diagram is proportional to the number of CCW orientation tests. Therefore we use the number of CCW orientation calls to stand in for the work done in this loop. (We count these separately from the CCW orientation tests done in the update portion.)

Finally, Dwyer also is based on the same two geometric primitives. Each quad-edge operation in the merge loop can be charged to a CCW orientation test or to an in-circle test, so these numbers give a good way to quantify the work in Dwyer as well. The specific additional operations associated to one of these primitives in an incremental algorithm is different than in Dwyer, but because profiling shows that the in-circle routine accounts for more time than any other routine (about half the time for both Dwyer and incremental algorithms running on the Sparcstation 2 described below) and the CCW orientation test is a significant part of what remains, looking at these numbers seems reasonable. We also note that this implementation of Dwyer uses Quicksort to sort the points so is asymptotically $\Theta(n \log n)$, but that for the number of sites that we considered sorting was a very small part of the overall work.

We first compare the point location strategies for Inc, BucketInc, and QtreeInc. While Inc spends almost all of its time doing point location, the point location routines in BucketInc and QtreeInc effectively remove this bottleneck.

Figure 1 compares the performance of the point location for Inc and BucketInc on uniformly distributed sites. The plots show that Inc uses an average of $O(\sqrt{n})$ tests per point location step. Simple regression analysis indicates that the number of tests per site grows as $0.86n^{.49}$. Therefore, we have plotted the curve $0.86\sqrt{n}$ along with the data. The curve is a close fit to the experimental data.

The number of tests needed for point location in BucketInc depends on the average density of the sites in the bucket grid. If the density of sites 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 and will use more space for its buckets. Figure 1d shows the dependence of the density on the cost of point location. (Here we just added together the numbers of CCW tests, bucket insertions, and spiral search bucket examinations per site. The three are not equivalent-time operations, but because the vast majority of these tests are CCW tests this gives a first approximation to the total cost.) 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 and the actual run times, we used $c = 2$ for the uniform distribution timing runs. Although the graph shows a large variation in the number of tests needed per site, the actual effect on the run time of the algorithm was less than 10%.

Adding the point location heuristic improves the performance of BucketInc substantially. The number of CCW orientation tests performed per site appears to be bounded by a constant near 10.5. The additional costs involved in spiral search and bucketing are insignificant. The algorithm almost always finds a site in the first bucket in the spiral search, and the average number of times that a site gets bucketed is 1.33 for even powers of two and 1.66 for odd powers. These last two operations are substantially faster than the CCW orientation test, so they have a very small effect on the run time.

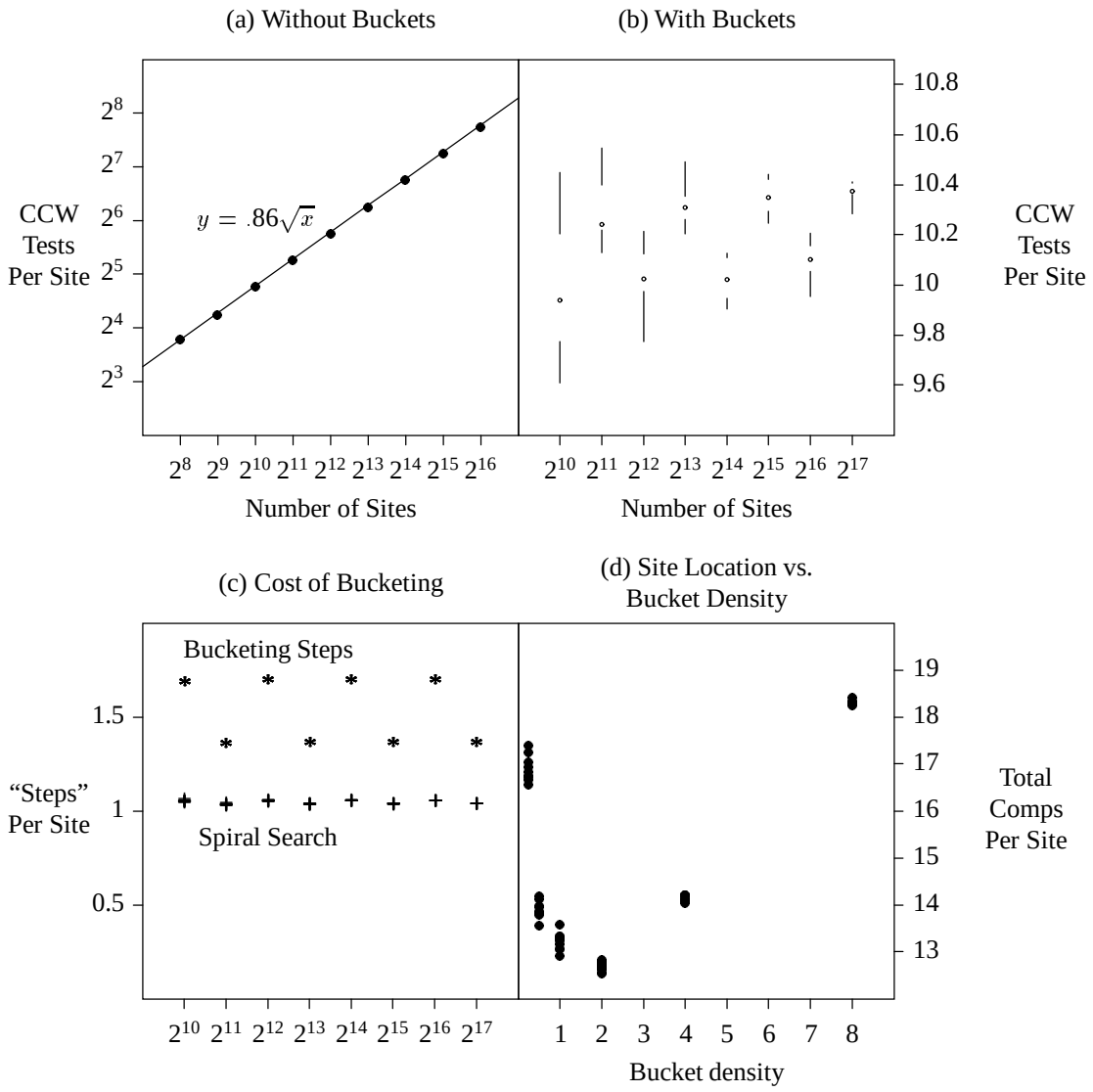

Figure 1: Comparison of point location costs for Inc and BucketInc.

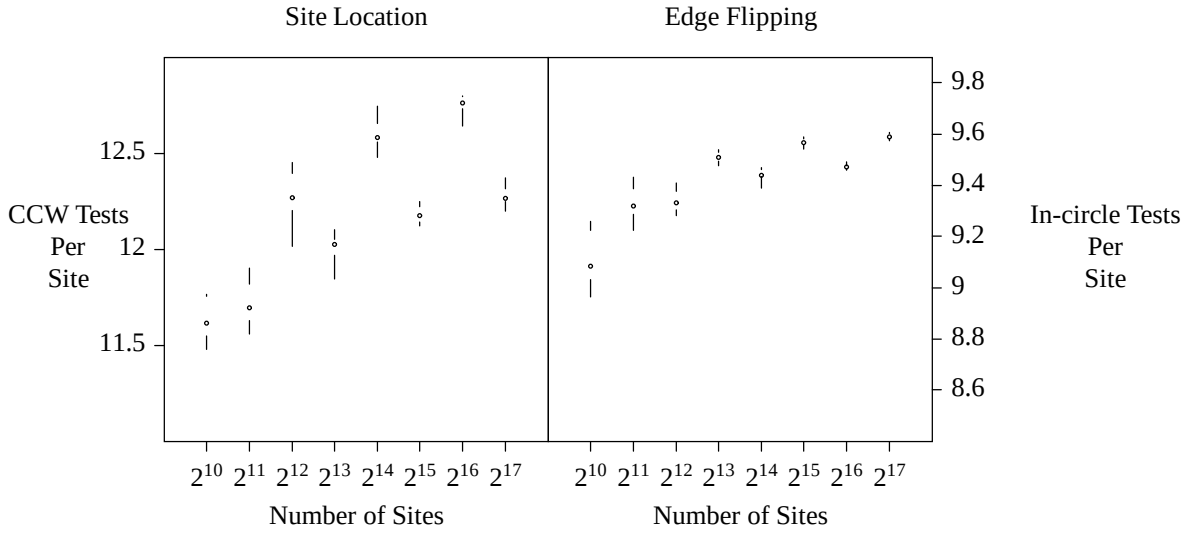

Figure 2: Performance of QtreeInc.

We note that the average number of times a point gets put into a bucket can range between 1.33 and 2.33, with the exact value a function of n and c. The best case is just before a re-bucketing is scheduled to occur. Then every point has been bucketed once, 1/4 of them have been bucketed a second time, 1/16 a third time, etc. This sum adds up to 1.33. Adding one more point re-buckets every point already bucketed and buckets the last one, raising the average by the one additional time that (almost) every point has been re-bucketed.

The results in Figure 2 show that QtreeInc's point location performs somewhat worse than BucketInc's. QtreeInc performs about 20% more CCW orientation tests than BucketInc.

For both BucketInc and QtreeInc the cost of point location fluctuates depending on whether $\log_2 n$ is even or odd. In BucketInc this is due to the fact that the algorithm re-buckets the sites at each power of four. Because of this, at each power of four the average bucket density drops by a factor of 4. The implicit quad-tree structure in QtreeInc gains another level at each power of 4.

We now consider the update operation. Since Inc and BucketInc insert the sites in random order, the analysis of Guibas, Knuth and Sharir [16] shows that the total number of in-circle tests is asymptotically $O(n)$. Sharir and Yaniv [23] tightened this bound to about $9n$. Figure 3 shows that this analysis is remarkably accurate. Figure 2 shows that the quad-tree insertion order in QtreeInc actually leads to 5 to 10% more in-circle tests than the random order in BucketInc and Inc.

Which of these incremental algorithms should run fastest? Inc's $\Theta(\sqrt{n})$ point location time make it impractical for large sets of sites. BucketInc does fewer CCW orientation tests and fewer in-circle tests than QtreeInc. The extra work that BucketInc does re-bucketing sites and examining buckets during spiral search is much less important than the reduced number of in-circle and CCW orientation tests. Therefore we can conclude that for the uniform case BucketInc should perform slightly better than QtreeInc and should be the fastest of this group.

Both the Dtree and the LEDA versions of the incremental algorithm were many times slower than BucketInc. (These results will be discussed later in this paper.) The update phase in these algorithms is identical to the update in Inc and the BucketInc, so we know its performance. The operations in maintaining the Delaunay tree are different enough from the other insertion algorithms that analyzing them in detail did not seem worth the effort, given the poor runtime performance.

We now consider Dwyer. Also shown in Figure 3 is the number of in-circle tests performed by Dwyer. The plot shows that Dwyer performs about 25% fewer in-circle tests than BucketInc. Figure 4 shows that Dwyer also does a few less CCW orientation tests than BucketInc. (Dwyer does about 16.5 tests per site, while BucketInc does a bit more than 10 in the locate portion and 9 in the update portion.) There are other

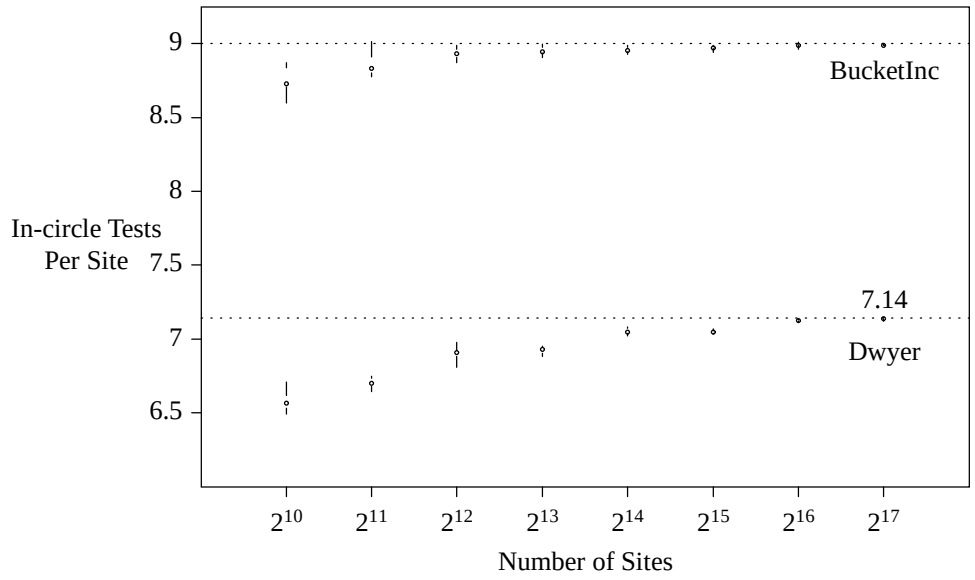

Figure 3: In-circle tests per site for BucketInc and Dwyer.

overhead costs that we are not considering, but as long as they are roughly comparable or a fairly small fraction of the total time Dwyer should run faster.

2.2. Performance of Giftwrapping

The determining factor in the run time of Giftwrapping is the cost of `Site-search`. This can be factored into two parts. The first is the number of times that a site is tested against a circumcircle. The second is the number of buckets examined. By *examined* we mean that a bucket is at least tested to see if the algorithm should search its contents for a new site.

Figure 5 summarizes the behavior of these parameters in our tests. Figures 5a and 5b show the performance of the algorithm for sites chosen from the uniform distribution in the unit square, while 5c and 5d show the performance of the algorithm for sites chosen from the uniform distribution in the unit circle. The reason for looking at both distributions in detail is that the behavior of Giftwrapping is heavily influenced by the nature of the convex hull of the input. In the square distribution, the expected number of convex hull edges is $O(\log n)$ [22]. The graph shows that the number of points tested per site stays constant over our test range, while the number of buckets examined actually decreases. This reflects the fact the number of edges on or near the convex hull of such point sets is relatively small, and that the algorithm only examines a large number of useless buckets on site searches near the convex hull.

However, it is apparent that this isn't the case when the sites are distributed in the unit circle, where the expected size of the convex hull is $O(n^{1/3})$ [22]. Here, there are a larger number of edges on or near the convex hull, and this is reflected by the fact that the number of buckets that the algorithm examines increases dramatically when compared to the earlier case. This sensitivity to the distribution of the input is an important feature of the algorithm. If the algorithm is to be used for a variety of input distributions a more adaptive data structure is needed to support the `Site-search` routine.

Pre-computing the convex hull of the sites and searching from hull edges inward may help to minimize the effect of this problem. But the bucket-based data structure will still be sensitive to clustering of the sites. The best way to fix these problems may be to replace the buckets with a nearest-neighbor search structure that is less sensitive to the distribution of sites, so that it can perform non-local searches more efficiently. Bentley's adaptive k-d tree [3] is a good example of such a data structure.

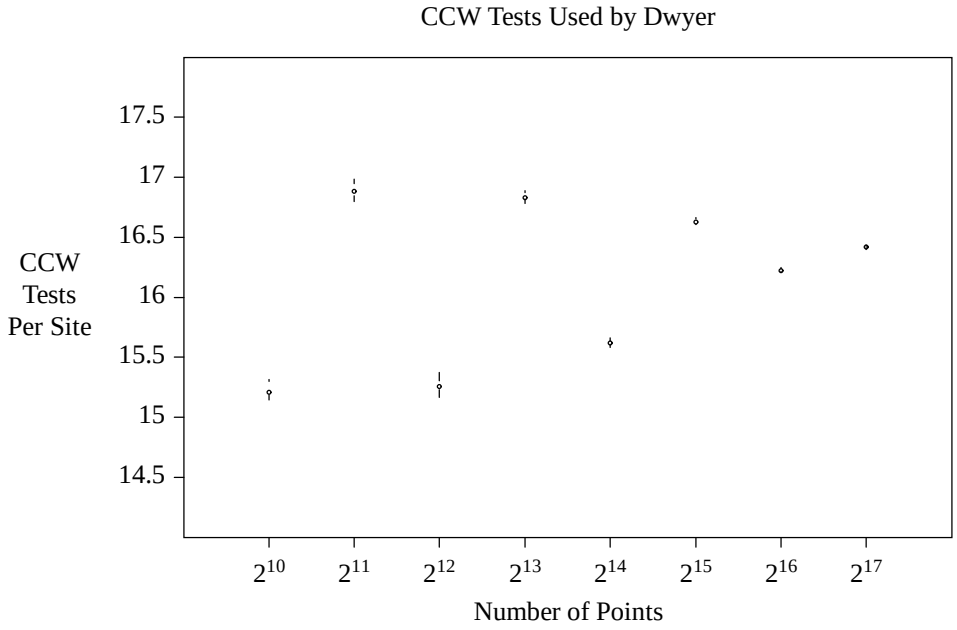

Figure 4: CCW orientation tests per site for Dwyer.

2.3. Performance of Fortune

The run time of Fortune 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 bucketing for this purpose. We would expect that these data structures would perform well on uniform inputs. In fact, for small input sets the algorithm seems to run in linear time.

Figure 6 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 6a indicates that the search procedure performs around 11.5 tests per site, on average. (Here we are testing whether the new site should be connected to a given site on the frontier by a Delaunay edge. This test can be done in constant time.)

The main bottleneck in Fortune 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 bucketed according to their y-coordinate. In addition, it is important to realize that only circle events are explicitly placed in the event queue. The $O(n)$ site events are stored implicitly by initially sorting the sites.

The problem here is that while the sites are uniformly distributed, the resulting priorities are not. Circle events tend to cluster close to the current position of the sweepline. This clustering increases the cost of inserting or deleting events into Fortune's bucket structure. Each operation requires a linear time search through the list of events in a particular bucket. With large buckets, this becomes expensive. Regression analysis shows that the number of comparisons per site grows as $9.95 + .25\sqrt{n}$ (see Figure 6b).

Watching animations of large runs of Fortune provides a heuristic explanation for this behavior. Since the sites are uniformly distributed, new site events tend to occur close to the frontier. If the new site causes a circle event to be added to the priority queue, chances are that the circle will not be large. Thus the y-coordinate of the top of the circle, which is the priority of the new event, will be close to the current position of the sweepline. If the circle is large, so the priority of the resulting event is far above the sweepline, it is likely that the event is invalid since large circles are likely to contain sites. Eventually some site or circle event will invalidate the large circle and replace it with a smaller one that lies closer to the sweepline. The result is the clustering that is clearly observable in the animations (Figure 7).

Given the behavior of the bucket data structure, it is natural to speculate as to whether a different

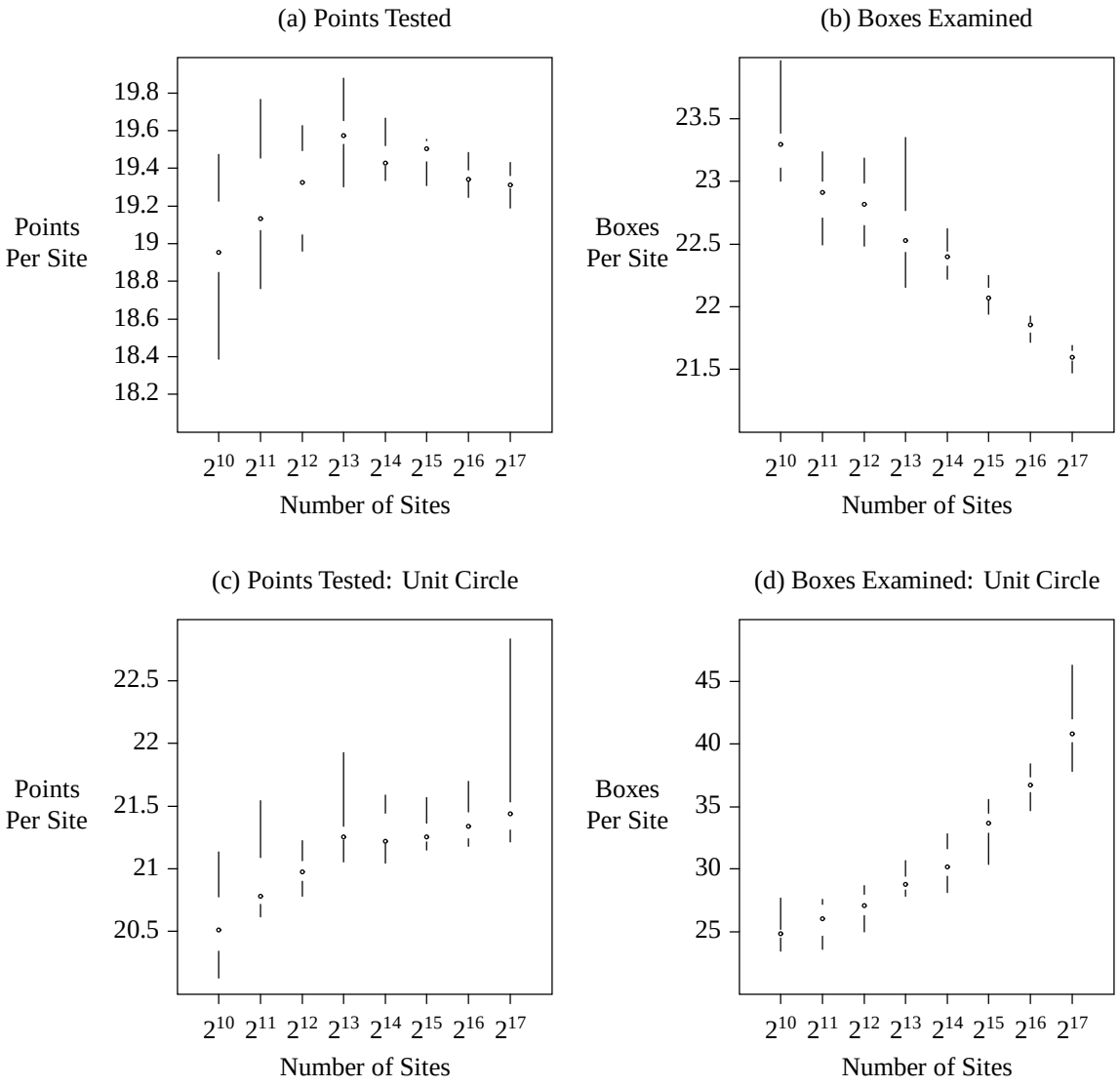

Figure 5: Performance of Giftwrapping

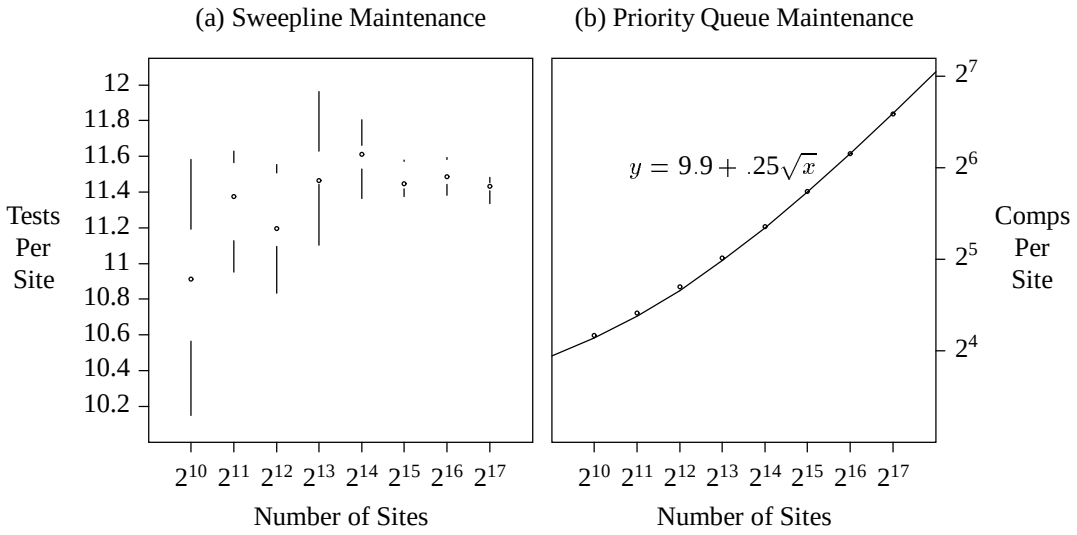

Figure 6: Cost of Fortune. The two main factors determining the performance of the algorithm are the work needed to maintain the heap and sweepline data structures.

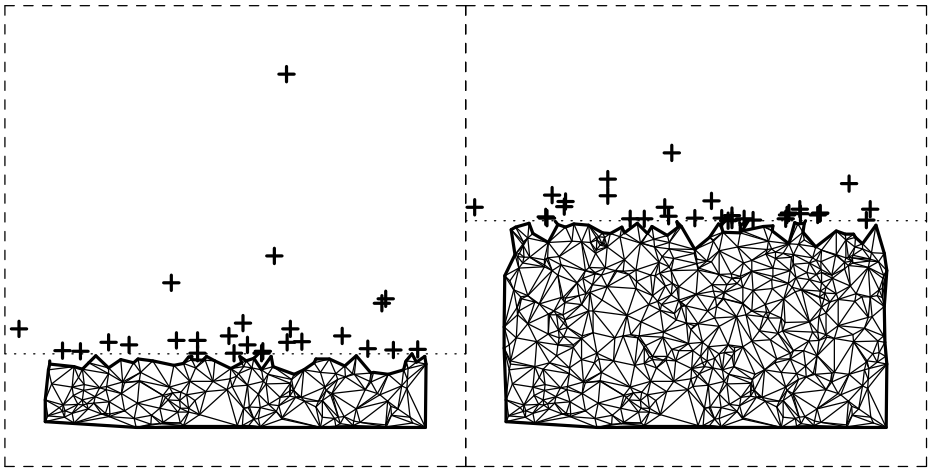

Figure 7: Circle events cluster close to the sweepline in Fortune. The first frame is early on one run in the algorithm, the second frame is later in the same run. Note how the "cloud" of circle events (+ signs) moves with the sweepline.

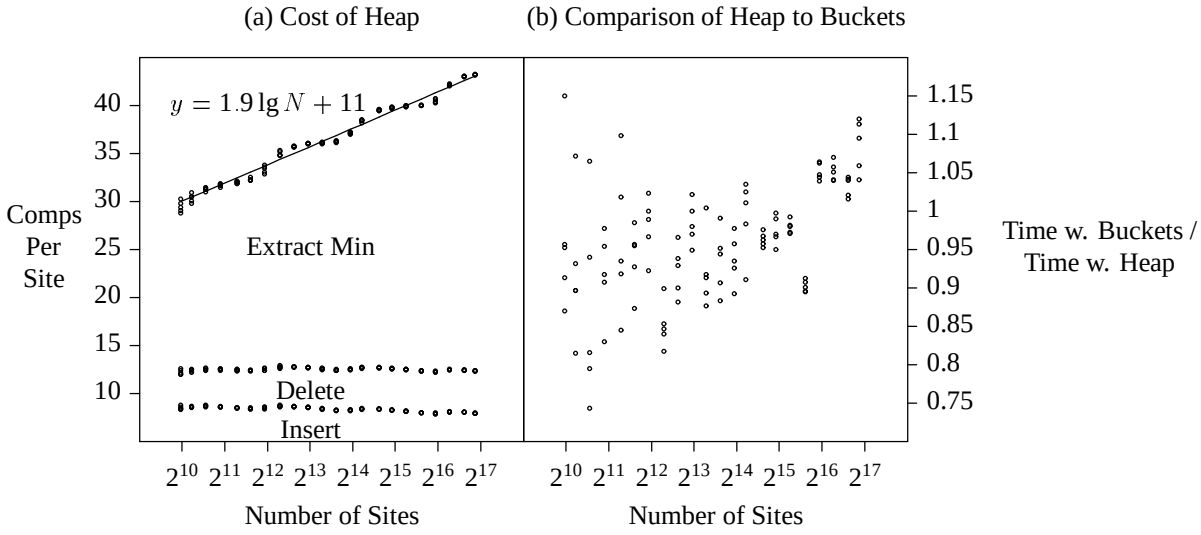

Figure 8: Cost of Fortune using a heap. Part (a) shows a breakdown of the cost of using a heap to represent the event queue. Part (b) plots the ratio of the run time of the old algorithm to the new.

data structure would provide better performance for larger problems. To investigate this possibility, we re-implemented Fortune using an array-based heap to represent the priority queue. This guarantees that each operation on the priority queue costs $O(\log n)$ time in the worst case, and using the array representation minimizes additional overhead.

To test the effectiveness of the new implementation, we performed a more extensive experiment. Each algorithm was tested on uniform inputs with sizes ranging from 1,024 to 131,072 sites. Figure 8a shows the performance of the heap data structure in the experiment. The line $2 \lg n + 11$ shows that the number of comparisons used to maintain the heap is growing logarithmically in n, rather than as $\sqrt{n}$. The plot also shows a more detailed profile of these comparisons. This profile indicates that most of the work is performed by the `extract-min` routine. By comparison, `insert` and `delete` are relatively cheap.

In actual use, the heap does not significantly improve the run time of the algorithm for data sets of the sizes that we considered. Figure 8b compares the run times of the two algorithms over the same range of inputs. In this graph, each data point is the ratio of the run time of the bucketing algorithm to the run time of the heap-based algorithm. The graph shows five trials for input sizes of between 2^{10} and 2^{17} sites at evenly spaced intervals.

The timings were taken using the machine and configuration described in Section 2.5. The plot shows that bucketing tends to be faster until 2^{16} sites, when the heap version starts to dominate. At 2^{17} sites, the heap version is roughly 10% better. The main reason that the improvement is not greater is that maintaining the heap seems to incur more data movement overhead than maintaining the bucket structure. The bucket structure appears to use the workstation's cache more effectively, and stays competitive, even though it is doing much more "work".

An interesting feature of the graph in Figure 8a is the fact that the number of comparisons periodically jumps to a new level, stays relatively constant, then jumps again. This is due to the fact that the heap is represented using an implicit binary tree. Thus, the number of comparisons jumps periodically when the size of the heap is near powers of two. On the graph, these jumps occur at powers of four, rather than two, because the average size of the heap over one run of Fortune is $O(\sqrt{n})$ rather than $O(n)$. We can prove that this is the case for uniformly distributed sites using a lemma by Katajainen and Koppenin [18].

2.4. Performance of Qhull

Barber's Qhull algorithm turned out to be much slower than the algorithms analyzed above. It also uses different primitives from the algorithms described above, so it was not clear how counts of primitives would aid comparisons between algorithms. We therefore did not analyze it in detail. It will be discussed further in the next subsection.

2.5. The Bottom Line

The point of all of this is, of course, to develop an algorithm that has the fastest overall run time. In the following benchmarks, each algorithm was run on ten sets of sites generated at random from the uniform distribution in the unit square. Run times were measured on a Sparcstation 2 with a 40 Mhz clock speed using the `getrusage()` mechanism in UNIX. The graphs show user time, not real time, and all of the inputs sets fit in main memory, and were generated in main memory so I/O and paging would not affect the results. Finally, the graph for Fortune shows the version using a heap rather than buckets, since this algorithm was better on the larger problems, and not much worse on smaller ones.

Figure 9 shows the results of the timing experiments. The graph shows that Dwyer gives the best performance overall. BucketInc is a bit faster than Fortune, particularly for larger problems where the $O(\log n)$ growth of the heap data structure overhead is more of a factor. QtreeInc is a somewhat slower than BucketInc. Giftwrapping is not competitive with any of the other four. These results are consistent with the analysis of primitives in the previous section.

The run times of the three other algorithms (Dtree, LEDA, and Qhull) are shown in Figure 10. These algorithms proved to be several times slower than the other algorithms that we considered. Part of this is undoubtedly due to implementation details. (For example, they frequently use malloc where the faster algorithms tend to allocate large arrays.)

The amount of overhead needed to maintain the Delaunay tree is much higher than that needed for the bucketing scheme so it seems unlikely to be competitive with the faster incremental algorithms no matter how it is implemented. (The comparison we made actually favors Dtree and LEDA somewhat, because Dtree and LEDA use the standard in-circle test that they came with, while the incremental algorithms and Dwyer use Fortune's stable in-circle test.)

Qhull is actually a program to compute convex hulls in arbitrary dimensions in a provably stable way. It therefore has more overhead than would be needed if its only task were to compute planar Delaunay triangulations, and if it were less concerned about stability.

While substantially different implementations could speed up these programs, it seems unlikely that they would be competitive. In their current form none of these programs can be recommended for its speed.

3. Nonuniform Point Sets

Each of the faster algorithms that we have studied uses a uniform distribution of sites to its advantage in a slightly different way. BucketInc uses the fact that nearest neighbor search is fast on uniformly distributed sites to speed up point location. QtreeInc buckets points to get a good insertion order. Dwyer uses the fact that only sites near the merge boundary tend to be affected by a merge step. Fortune uses bucketing to search the sweepline. Giftwrapping depends on a uniform bucket grid to support site searching.

In order to see how each algorithm adapts to its input, we will study further tests using inputs from very nonuniform distributions. In Table 2 the notation $N(\mu, s)$ refers to the normal distribution with mean μ and standard deviation s, and $U(a, b)$ is the uniform distribution over the interval $[a, b]$.

The graphs show each algorithm running on five different inputs of 10K sites from each distribution. The uniform distribution serves as a benchmark.

Figure 11 shows the effect of these site distributions on BucketInc and QtreeInc. For these runs we changed the value of the average bucket density used by BucketInc (see Section 2.1). Before we used a value $c = 2$, but for these tests we used a value $c = 0.25$. This slows down the uniform case somewhat

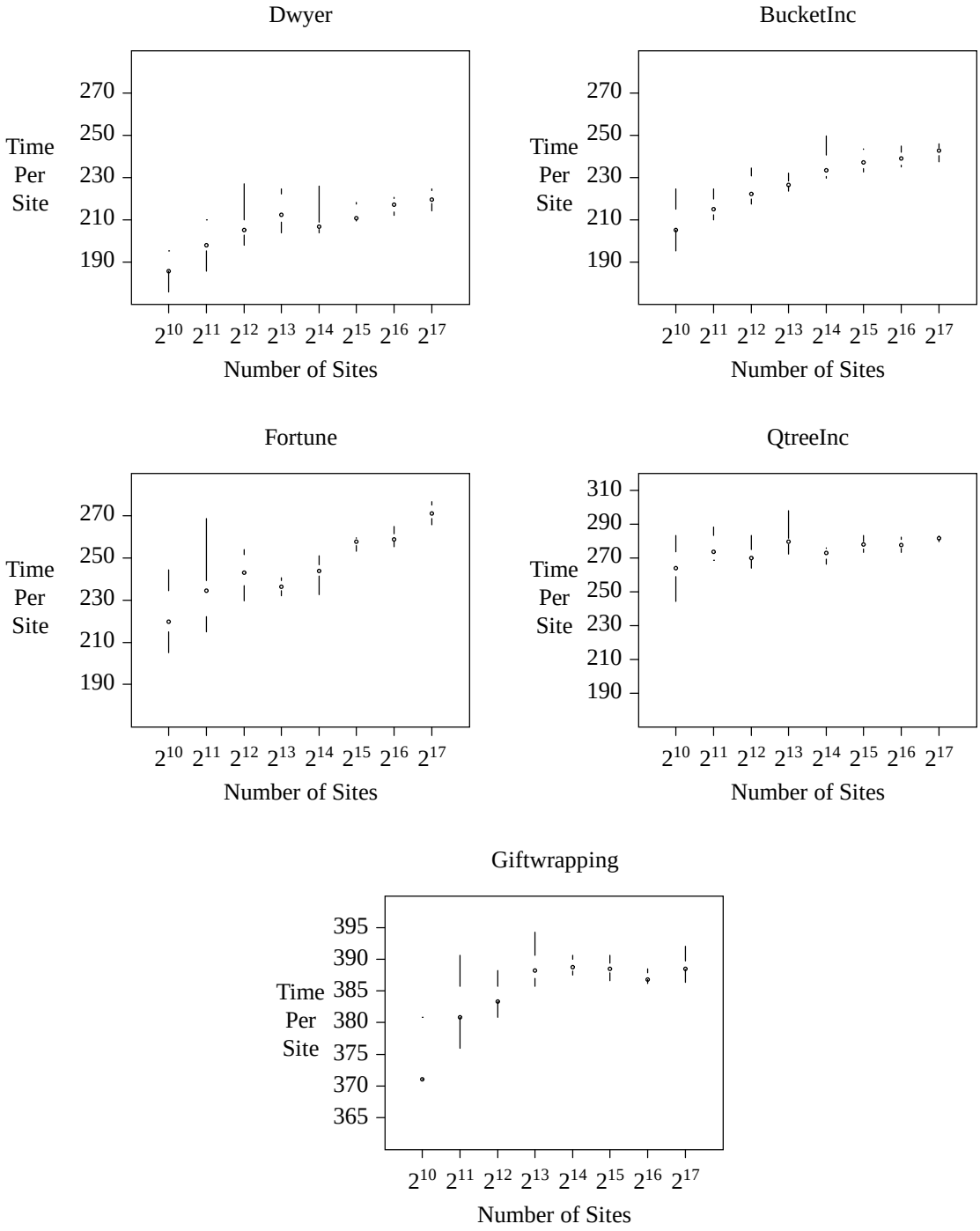

Figure 9: Comparison of the expected run times of different algorithms on sites chosen at random from a uniform distribution in the unit square. Times are in microseconds.

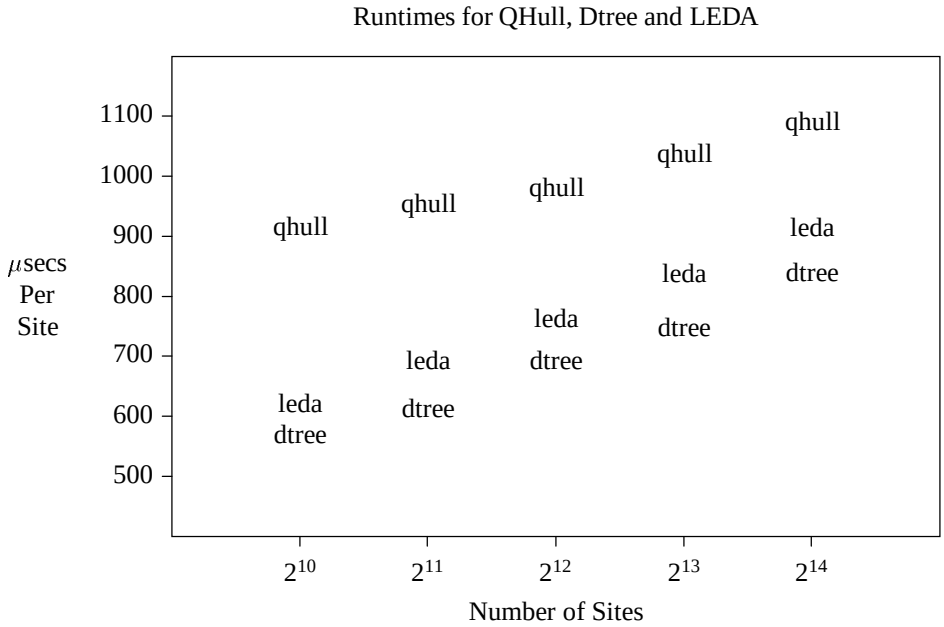

Figure 10: Comparison of the expected run times of Dtree, LEDA, and Qhull on sites chosen at random from a uniform distribution in the unit square.

Name	Description
unif	uniform in the unit square.
ball	uniform in a unit circle.
corn	$U(0, .01)$ at each corner of the unit square.
diam	$t = U(0, 1)$, $x = t + U(0, .01) - .005$, $y = t + U(0, .01) - .005$
cross	$n/2$ sites at $(U(0, 1), .5 + U(-0.005, .005))$; $n/2$ at $(.5 + U(-0.005, .005), U(0, 1))$.
norm	both dimensions chosen from $N(0, .01)$.
clus	$N(0, .01)$ at 10 points in the unit square.
arc	in a circular arc of width .01

Table 2: Nonuniform distributions.

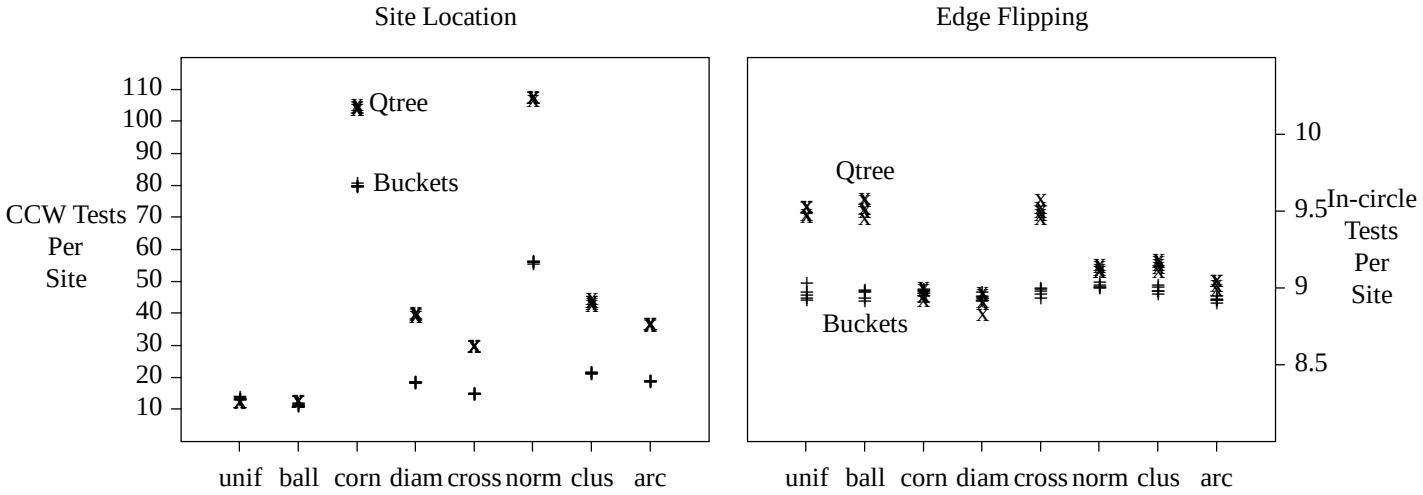

Figure 11: Primitive tests for BucketInc and QtreeInc on non-uniform inputs, n is 10K.

(BucketInc's run time goes from about 240 μsecs per site to a bit over 250 μsecs) but it gives better times for the non-uniform distributions.

As expected, site distributions with heavy clustering (such as corn and norm) stress the point location data structures in each algorithm, increasing the number of CCW orientation tests by up to a factor of ten. QtreeInc degraded more than BucketInc. Figure 12 shows that the average number of buckets examined during the spiral search in BucketInc remains small. It is larger for the uniform distributions than it was in the uniform distribution tests in the previous section because of the change in c, but this is a case where the more heavily clustered distributions actually do better than the uniform distributions. Some isolated sites may have to look at a lot of buckets, but clusters mean that most sites are close to a lot of other sites. As noted earlier, the cost of bucketing and re-bucketing points depends only on c and n, and averages between 1.33 and 2.33 bucket insertions per point.

Figure 11 also shows that the distribution of the input has little effect on the number of in-circle tests that BucketInc performs, as was predicted by the theoretical analysis. QtreeInc actually uses fewer in-circle tests on clustered distributions than on uniform ones. This is probably because sites within a given bucket are inserted in random order, so the clustering of sites in the same bucket makes the site insertion order more random. Because the CCW orientation tests in the point location routines are substantially faster than in-circle tests, we will later see that the total run time of the worst cases is not much more than a factor of two times the best case even though the cost for the locate portion goes up by a factor of 10.

Figure 13 summarizes the performance of Fortune in this experiment. The first graph shows that the bucket-based implementation of the event queue is very sensitive to site distributions that cause the distribution of priorities to become extremely nonuniform. In the cross distribution, this happens near the line $y = 0.5$. At this point, all of the circle events associated with $n/2$ sites near the line cluster in the few buckets near this position. The corn distribution causes a similar problem, but to a lesser degree. Here, all of the events associated with the $O(\sqrt{n})$ circles in the event queue tend to stay clustered in one bucket at $y = 0$ and another at $y = 1$. In both of these cases, the non-uniform distribution of sites in the x-direction also slows down site searches on the frontier, but this effect is less pronounced than the bad behavior of the event queue.

The second graph shows that the performance of the heap is much less erratic than the buckets. The small jumps that do appear are due to the fact that the event queue does become larger or smaller than its expected size on some distributions. However, since the cost of the heap is logarithmic in the size of the queue, this does not cause a large degradation in performance.

Figures 14 and 15 show the performance of Dwyer in the experiment. Dwyer is slowed down by

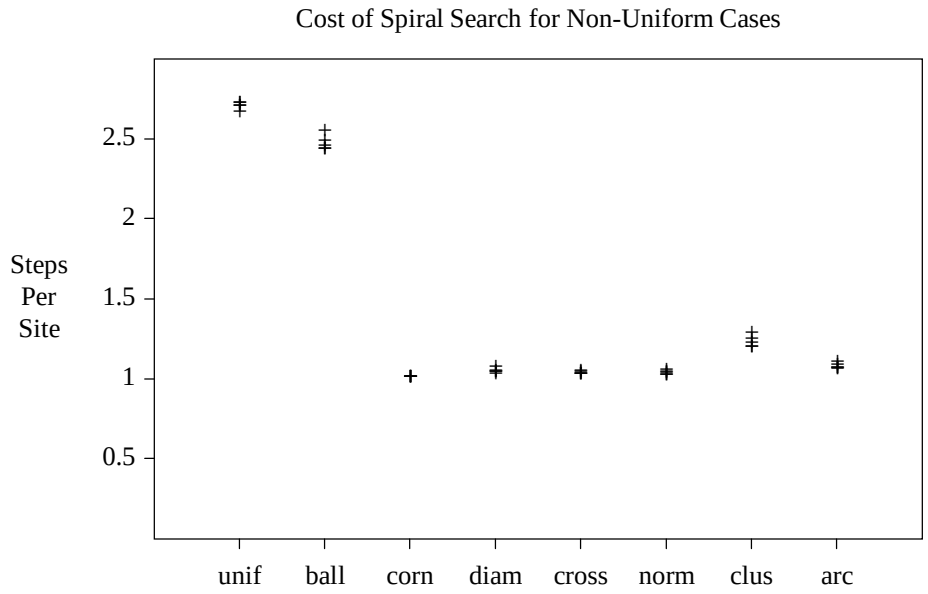

Figure 12: Buckets examined for BucketInc on non-uniform inputs, n is 10K.

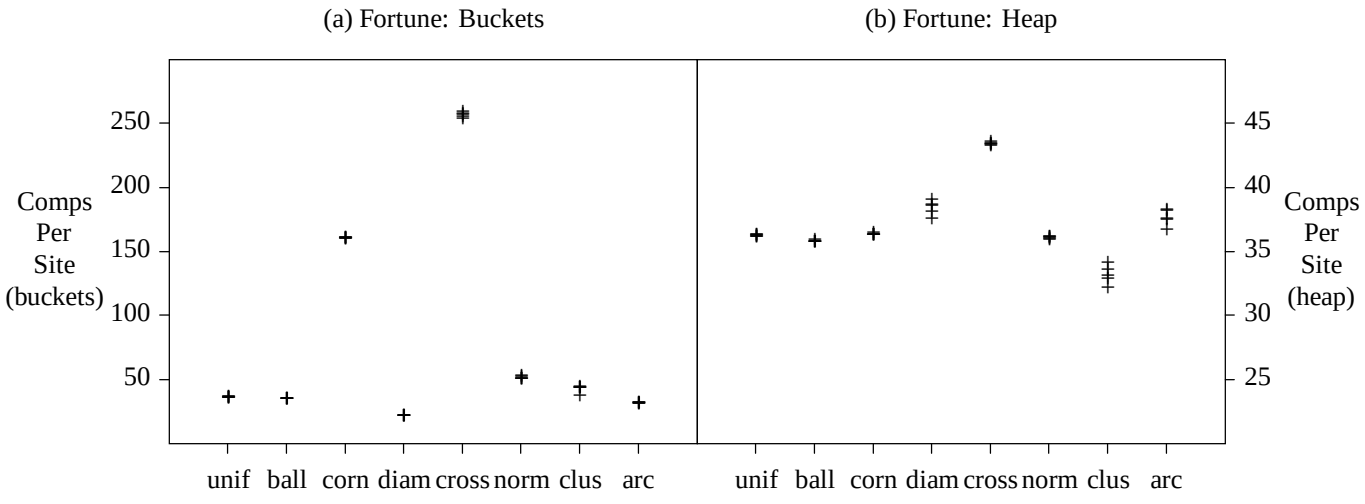

Figure 13: Fortune's priority queue on non-uniform inputs, n is 10K.

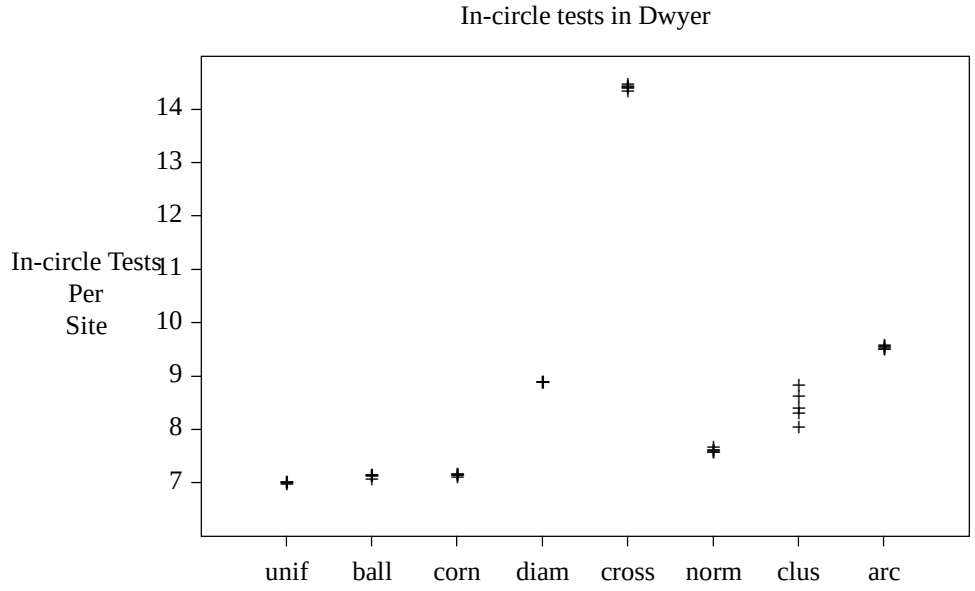

Figure 14: Dwyer in-circle tests on non-uniform inputs, n is 10K.

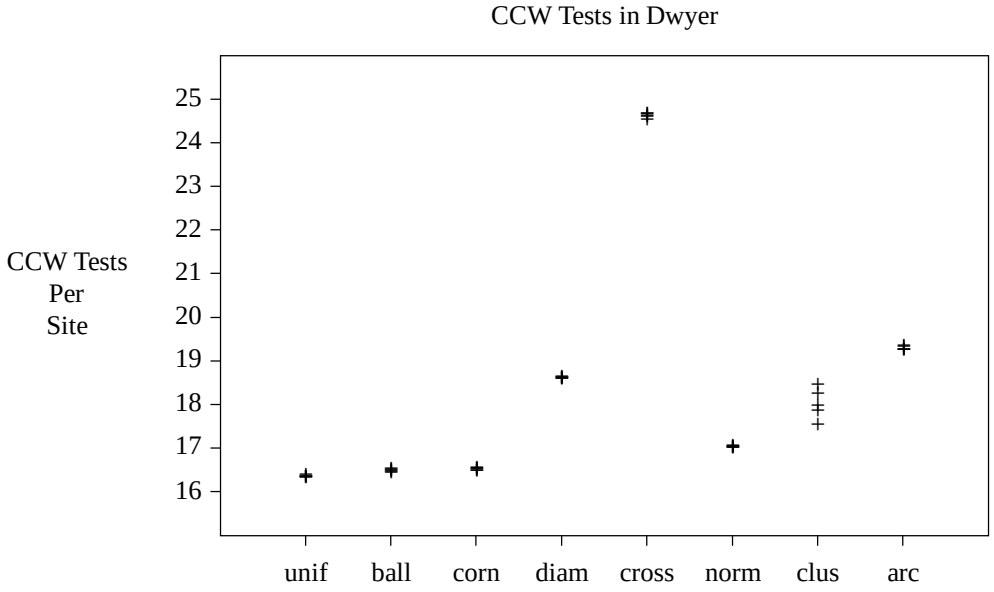

Figure 15: Dwyer CCW tests on non-uniform inputs, n is 10K.

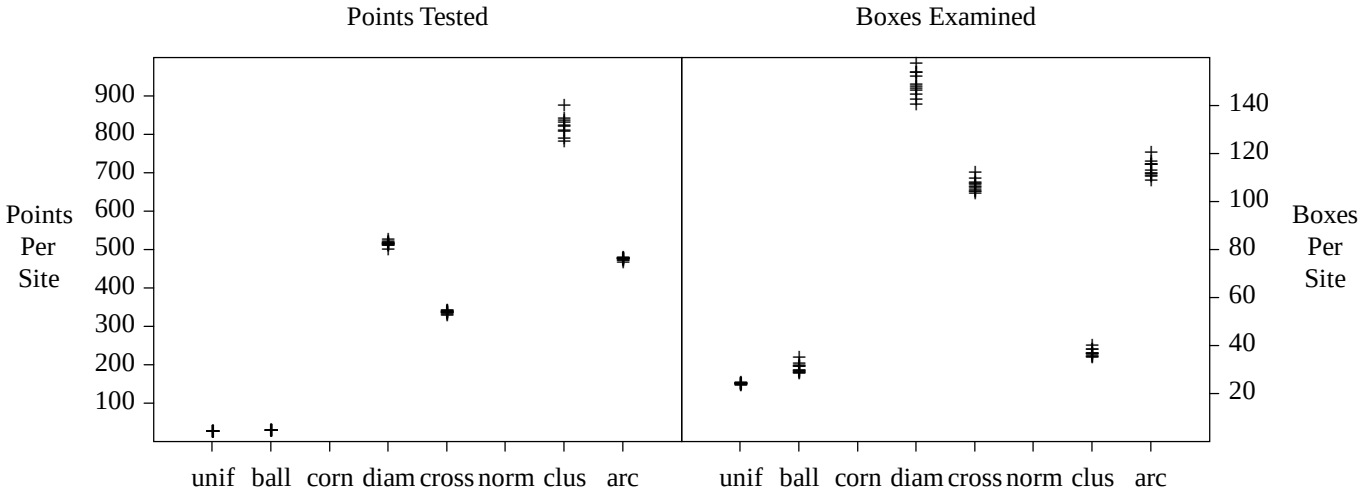

Figure 16: Giftwrapping is very sensitive to bad inputs.

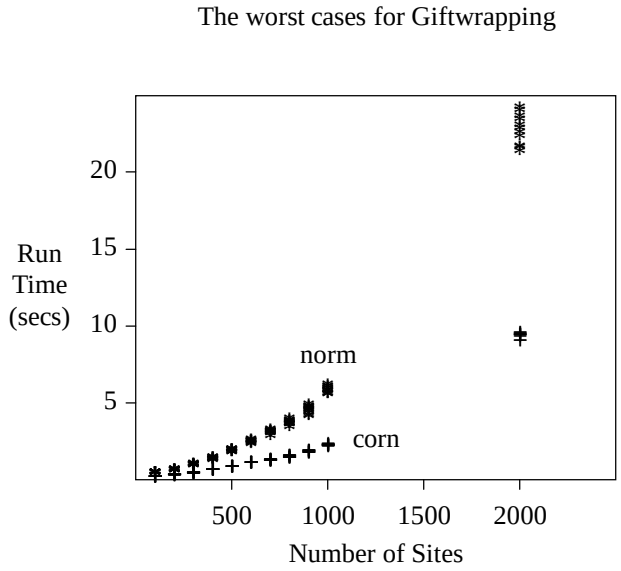

Figure 17: Giftwrapping for the norm and corn distributions.

distributions that cause the algorithm to create many invalid edges in the subproblems, and then delete them later in the merge steps. This effect is particularly pronounced with the cross distribution because the group of sites near the line $x = 0.5$ is very tall and skinny, creating a worst case for the merge routine.

It appears that there are about 9.5 more CCW tests than in-circle tests for every distribution except cross, where there are about 10 more. Each time through the merge loop the algorithm does two CCW orientation tests, but can do between 0 and an arbitrarily large number of in-circle tests. Therefore there is no reason to assume that the in-circle tests would be a fixed fraction of the CCW tests. We do not know why the difference between them should remain almost constant.

Figure 16 shows how the bad inputs affect Giftwrapping. These figures leave out the two worst inputs for this algorithm: corn and norm, because the algorithm would have taken several hours to finish the benchmark. The $\Theta(n^2)$ behavior of the algorithm on these inputs is shown in Figure 17.

Giftwrapping is easily the most sensitive to the distribution of its input. This is not surprising, since it depends on essentially the same routine that BucketInc uses for point location, and we have already seen that the point location subroutine performed badly on bad inputs. This did not handicap to BucketInc to a

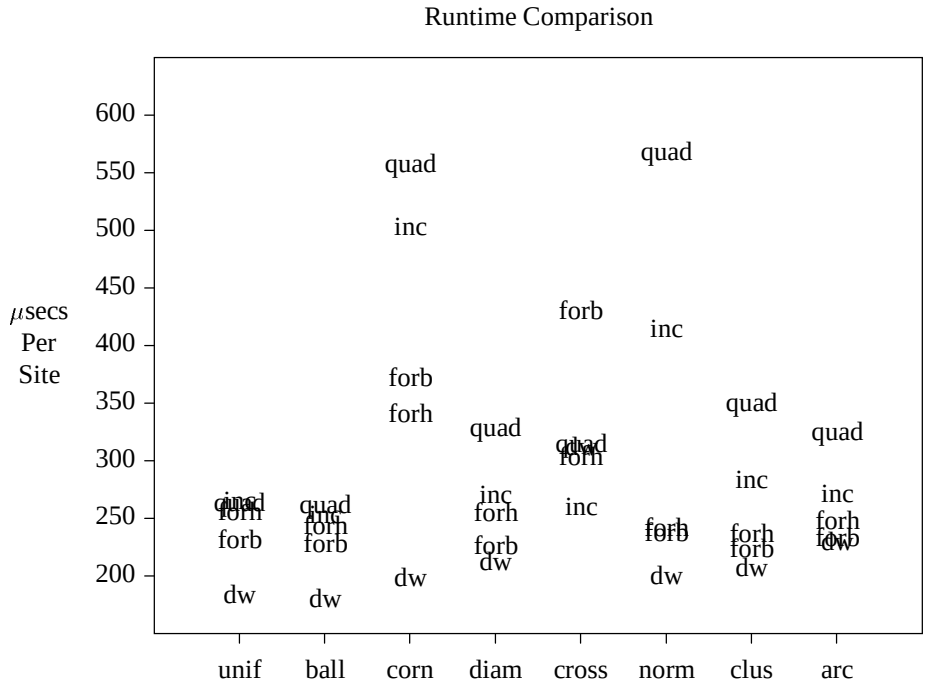

Figure 18: Run times on non-uniform inputs for Dwyer (dw), Fortune with buckets (forb) and heap (forh), BucketInc (inc), and Qtree (quad). n is 10K.

large degree because the point location routine is not the major bottleneck in that algorithm. However, the performance of `Site-search` largely determines the run time of Giftwrapping.

Finally, to understand how the abstract measures actually effect performance, Figure 18 shows the average run time of the five trials with each algorithm except Giftwrapping. Since none of the run times in the graph are much greater than Giftwrapping's performance even in the uniform case, we eliminated it from this graph.

4. Timings on a Different Machine

While the analysis of primitive operations gives some independence from the particular machine we were running on (a Sparcstation 2), it seemed wise to see whether the run time results held for another machine. We ported the best three algorithms (Dwyer, BucketInc, and Fortune using a heap) to a DEC Alpha 3000 with a clock rate of 125 Mhz and ran the timing suite for uniformly distributed sites. The results appear in Figure 19.

The good news is that for both machines Dwyer was fastest, BucketInc was second (for all but the largest case), and Fortune was slowest. However, there are at least two surprising results from these runs. The first is how much Dwyer improved relative to the other two algorithms. Instead of having a 10% advantage it takes half the time of the others.

The second is the way that the run times change as the number of sites increase. The "linear" run time of BucketInc is not linear, but grows more rapidly than either of the others. Fortune's run time grows slowly up to 2^{13} sites and then increases more rapidly.

Profiling shows that in-circle tests now take up only a quarter of the time of Dwyer and BucketInc instead of half, so data structure manipulation and loop overhead are more significant relative to in-circle testing and CCW testing. This could explain non-uniform speedups, but not the apparent changes in growth rates. Such a change would favor algorithms using simpler data structures than the full quad-edge data structure if this reduced the time spend on data-structure manipulation.

Our suspicion is that the ability of the algorithms to efficiently use the cache is also a factor, but

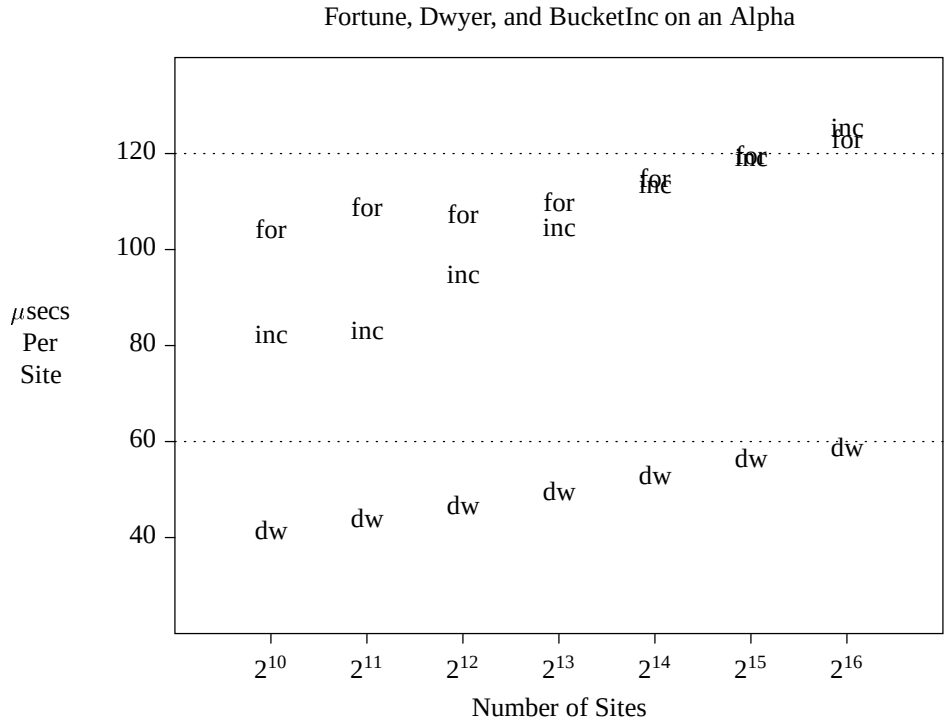

Figure 19: Run times for Dwyer (dw), BucketInc (inc), and Fortune using a heap (for) on a DEC Alpha for uniformly distributed sites.

determining exactly what is happening will remain as future work.

5. Notes and Discussion

The experiments in this paper led to several important observations about the performance of serial algorithms for constructing planar Delaunay triangulations. These observations are summarized below:

- Dwyer is the strongest overall for this range of problem sizes. This advantage was small on a Sparcstation 2, but was a factor of 2 on a DEC Alpha. It is also the most resistant to bad data distributions, with an $O(n \log n)$ worst case. On the other hand, it was not substantially faster than BucketInc and Fortune on the Sparcstation 2. One of these algorithms could prove to be the fastest on some other system.

- A simple enhancement of the naive incremental algorithm results in an easy to implement algorithm on-line algorithm that runs in $O(n)$ expected time for uniformly distributed sites. It is faster than previous incremental variants and competitive with other known algorithms for constructing the Delaunay triangulation for all but very bad site distributions.

- On uniformly distributed sites circle events in Fortune cluster near, and move with, the sweepline. Because the frontier and the event queue are expected to be of size $O(\sqrt{n})$ this causes his bucketing-based event queue implementation to perform badly on large inputs. However, for moderate size data sets (up to 50,000 sites on the Sparcstation that we used for our experiments) the bucketed event queue is a bit faster than a heap-based event queue.

- Dtree, LEDA, and Qhull are several times slower than Dwyer and the other fast algorithms in their current form.

5.1. Future Work

A question first raised by Steve Fortune in a discussion at the conference where this paper was presented is, "How much can you gain by optimizing the best algorithms?" He suggested the idea of modifying the in-circle test by computing the plane through the lifted versions of the three points defining the circle. Testing to see if a fourth point lies within the circle then becomes determining whether the lifted version of that point lies above, below, or on the plane. This can be determined by computing a dot product. For a single test this would not gain, but it would make subsequent tests against the same circle much faster. Therefore caching this plane with each triangle in an incremental algorithm could speed things up considerably. Similarly, during the merge step of Dwyer the same three sites can be tested repeatedly against different fourth points, so Dwyer might also benefit from this idea. This is only one of many possible optimizations.

Developing such a "best" code and making it available would be a service to the community, but is beyond the scope of this paper. We hope to work on this in the future.

The timings that appear for the Alpha workstation also bring up many intriguing questions for future study. As the CPUs in workstations and PCs get faster, the problems facing algorithm designers include the efficient use of the cache and memory system as well as efficient use of the CPU. While there have been attempts at theoretical analysis of algorithms on complex memory systems, the models involved are generally complicated and the analysis of even simple algorithms is highly challenging [1, 26, 14, 27]. Such analysis is also made more difficult by the fact that the "memory system efficiency" of an algorithm is very dynamic and data dependent. Therefore, finding ways to combine a more tractable abstract analysis with good tools and methods for experimentally analyzing the behavior of algorithms on different memory systems should be a fruitful area for future research.

5.2. Acknowledgements

We are indebted to Steve Fortune, Rex Dwyer, and Olivier Devillers for sharing their code with us and to Brad Barber for making his code publicly available. Steve Fortune and John Reif made a number of helpful suggestions along the way. John Reif and the sysadmin staff at Duke let us use a Sparcstation to run our experiments. Finally, the anonymous referees made many suggestions that greatly improved the final paper.

References

[1] B. Alpern, L. Carter, and E. Feig. Uniform memory hierarchies. *ACM Symposium On Theory Of Computing*, pages 600–608, 1990.

[2] C. Brad Barber. *Computational geometry with imprecise data and arithmetic*. PhD thesis, Princeton, 1993.

[3] J. L. Bentley. K-d trees for semidynamic point sets. *Proc. 6th Annual ACM Symposium on Computational Geometry*, pages 187–197, 1990.

[4] 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):563–580, 1980.

[5] J.-D. Boissonnat and M. Teillaud. On the randomized construction of the Delaunay tree. *Theoret. Comput. Sci.*, 112:339–354, 1993.

[6] K. Brown. Voronoi diagrams from convex hulls. *IPL*, pages 223–228, 1979.

[7] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, ii. *Discrete and Computational Geometry*, 4:387–421, 1989.

[8] R. A. Dwyer. A faster divide-and-conquer algorithm for constructing delaunay triangulations. *Algorithmica*, 2:137–151, 1987.

[9] R. A. Dwyer. Higher-dimensional voronoi diagrams in linear expected time. *Discrete & Computational Geometry*, 6:343–367, 1991.

[10] H. Edelsbrunner and R. Seidel. Voronoi diagrams and arrangements. *Disc. Comp. Geom.*, 1:25–44, 1986.

[11] S. Fortune. A sweepline algorithm for voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[12] S. Fortune. Stable maintenance of point-set triangulations in two dimensions. *IEEE Symposium on Foundations of Computer Science*, pages 494–499, 1989.

[13] S. Fortune. Numerical stability of algorithms for delaunay triangulations and voronoi diagrams. *Annual ACM Symposium on Computational Geometry*, 1992.

[14] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, , and J. S. Vitter. External-memory computational geometry. *Symposium on Foundations of Computer Science*, 34, 1993.

[15] P. Green and R. Sibson. Computing dirichlet tessellations in the plane. *Computing Journal*, 21:168–173, 1977.

[16] L. Guibas, D. Knuth, and M. Sharir. Randomized incremental construction of delaunay and voronoi diagrams. *Algorithmica*, 7:381–413, 1992.

[17] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):75–123, 1985.

[18] J. Katajainen and M. Koppinen. Constructing delaunay triangulations by merging buckets in quad-tree order. Unpublished manuscript, 1987.

[19] A. Maus. Delaunay triangulation and the convex hull of n points in expected linear time. *BIT*, 24:151–163, 1984.

[20] 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*, 27:306–337, 1984.

[21] F. P. Preparata and S. J. Hong. Convex hulls of finite sets of points in two and three dimensions. *Commun. ACM*, 20:87–93, 1977.

[22] L. A. Santaló. *Integral Geometry and Geometric Probability*. Addison-Wesley, Reading, MA, 1976.

[23] M. Sharir and E. Yaniv. Randomized incremental construction of delaunay diagrams: Theory and practice. *Annual ACM Symposium on Computational Geometry*, 1991.

[24] Peter Su. Efficient parallel algorithms for closest point problems. Technical report, Dartmouth College, Nov. 1994.

[25] M. Tanemura, T. Ogawa, and N. Ogita. A new algorithm for three dimensional voronoi tessellation. *Journal of Computational Physics*, 51:191–207, 1983.

[26] J. S. Vitter. Efficient memory access in large-scale computation. *Annual Symposium on Theoretical Aspects of Computer Science*, 1991.

[27] J. S. Vitter and E. A. M.Shriver. Optimal algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12:110–147, 1994.