A Comparison of Six Approaches to the Nearest Neighbor Problem

Jason Grinblat
Indiana University at Bloomington

August 5, 2001

Abstract

Nearest neighbor searching is one of the most frequently occurring problems in computational geometry. Many classes of problems include nearest neighbor searching, and often one approach to finding the nearest neighbor is more suitable for one class of problems than for another. This paper will present six algorithms that solve the nearest neighbor problem with uniformly distributed points in a two dimensional square region, and compare them using the following metrics: time complexity, space complexity, accuracy, ease of implementation, and extensibility to other dimensions and distribution functions.

Contents

1 Introduction 2

2 Time and Space Complexity 3

3 The Setting 3

4 The Straightforward Algorithm 4

5 The Spiral Method 6

6 The Grid-Bin Method 9

7 The Grid-Bin Method with Sweeping Preprocessing 10

8 The Delaunay Triangle Neighbor Search 12

9 The Delaunay Node Neighbor Search 14

10 Conclusions and Discussion 16

1 Introduction

The nearest neighbor problem for two dimensions is defined as follows: given a set of points Ω in the plane, and a point α also in the plane, we are interested in which point in Ω is closest to α. Any distance metric can be used in computing the nearest neighbor, but for simplicity this paper will assume Euclidean distance. It is easy to see how this problem can be extended to higher dimensions. The goal of this paper is to serve as a guide to anyone who is interested in implementing a nearest neighbor algorithm.

Section 2 defines and briefly discusses the notion of time and space complexity. The author presents both *big-oh* notation and probabilistic *big-oh* notation, and mentions how for the purposes of this paper it is sometimes more relevant to discuss probabilistic *big-oh* notation.

Many problems in computational geometry involve computing the nearest neighbor, such as, for example, the construction of centroidal Voronoi tessellations. Section 3 discusses how the author came across the problem of computing the nearest neighbor, and how the computationally intensive nature of the problem demanded a more efficient solution than the straightforward approach.

Section 4 presents the straightforward algorithm for solving the nearest neighbor problem, and discusses its performance with respect to each of the metrics defined in the abstract.

Section 5 describes the spiral method for computing the nearest neighbor proposed by Bentley, Weide, and Yao [1]. The algorithm's performance is once again evaluated, and compared with the performance of the straightforward algorithm.

Section 6 discusses how the ideas proposed by Bentley, Weide, and Yao and the complexities involved in implementing the spiral method of Section 5 lead the author and John Burkardt, professor of mathematics at Iowa State University, to develop the grid-bin method. The performance of the new algorithm and its relation to the performance of the previous two algorithms is also addressed.

Section 7 discusses how problems with the time complexity of the grid-bin method lead the author and Professor Burkardt to improve upon the algorithm, and develop a similar algorithm that uses more efficient means of preprocessing by "sweeping" across the grid. The advantages and disadvantages of the updated grid-bin method are discussed, along with the algorithm's performance with respect to the metrics.

Section 8 presents the notion of the Delaunay triangulation and describes the Delaunay triangle neighbor search. The new method is evaluated and compared with the previous algorithms.

Section 9 expands upon the use of the Delaunay triangulation and presents the Delaunay node neighbor search. The two algorithms based on the Delaunay triangulation are compared, and the new algorithm's performance is evaluated.

Section 10 reviews the topics discussed in this paper and qualitatively compares each of the six algorithms presented, giving suggestions as to when each algorithm is appropriate. The metrics themselves and how they can be applied to other approaches to the nearest neighbor problem are discussed as well.

2 Time and Space Complexity

The notion of time complexity is concerned with the rate at which the time it takes for a program to run grows with respect to some increasing input variable. Space complexity, similarly, is concerned with the rate at which the amount of allocated memory grows. Time and space complexity are often measured using *big-oh* notation. (For a detailed explanation of *big-oh* notation, see [2]). For example, suppose some algorithm performs $f(n)$ operations when its input is of size n. Then that algorithm is $O(g(n))$ if $f(n)$ grows no faster than a constant multiple of $g(n)$. The same definition is true for space complexity. Essentially, $g(n)$ is an upper bound on the growth rate of $f(n)$.

Probabilistic *big-oh* notation is similar to *big-oh* notation, except that $g(n)$ is an upper bound on what the growth rate of $f(n)$ will *probably* be. There are some algorithms such that for almost every case their time complexity is, for instance, $O(n)$, but for some rare cases is $O(n^2)$. These algorithms have time complexity of $O(n^2)$, but have probabilistic time complexity of $O(n)$. Some of the algorithms presented in this paper are of this class, and so are better described using probabilistic *big-oh* notation.

It is evident why, for practical purposes, we prefer to have algorithms with slower time and space growth rates. For example, an algorithm with time complexity of $O(n)$ might perform one thousand operations for $n = 1000$, while an algorithm with time complexity of $O(n^2)$ might perform one million operations for $n = 1000$. Assuming for simplicity that it takes one second to perform one operation, our first algorithm would take 16.7 minutes to execute, while our second algorithm would take 11.6 days.

3 The Setting

The author became acquainted with nearest neighbor searching while working on the problem of constructing probabilistic centroidal Voronoi tessellations using random sampling. Voronoi tessellations in two dimensions can be described as follows: given a set of points Ω in the plane, we can divide the plane into regions, or cells, such that each point in Ω is assigned a cell and each point α in the plane is in the cell of $\beta \in \Omega$ if and only if α is closer to β than to any other point in Ω, with respect to some distance function d. Voronoi diagrams can be constructed in any dimension. A centroidal Voronoi tessellation is a Voronoi tessellation in which each $\beta \in \Omega$, or cell generator, is the centroid of its respective Voronoi cell. A detailed treatment of centroidal Voronoi tessellations is given in [3].

In order to construct centroidal Voronoi tessellations, the author used a revised version of an algorithm known as *Lloyd's method*. *Lloyd's method* is described as follows:

1. Choose n points at random and generate the associated Voronoi diagram
2. Calculate the centroids of each of the Voronoi cells
3. Return to step 1, using these centroids as the new generators

Each iteration brings the calculated centroids closer to their respective generators. Eventually the algorithm converges and the two points are the same. The algorithm used by the author is a probabilistic version of *Lloyd's method.* Instead of generating the Voronoi diagram, we approximate it by sampling random points within the region, and assigning each of those random points to the closest cell generator. With enough points, this method gives a reasonably accurate Voronoi diagram, and is less computationally intensive than directly generating the tessellation. Since we only have an approximation of each Voronoi polygon, we must use an alternative method of calculating the centroid. Instead of computing the centroid directly, we average each of the points assigned to a given generator, and assume that value as the centroid. This revised version of *Lloyd's method* results in a set of generators that, when the Voronoi diagram is generated, are approximately equal to the centroids of their respective Voronoi regions. It should be noted that this algorithm does not generate the Voronoi diagram, but returns a list of possible centroidal Voronoi cell generators.

The problem of computing the nearest neighbor arises during the approximation of the Voronoi diagram using random sampling. During each iteration of *Lloyd's method*, we sample $n * C$ points, where C is some chosen constant, calculate to which generator each sampled point is closest, and average each of the n sets of sampled points to determine the new centroids. Since we sample $n * C$ points, we expect to have on average C sampled points per Voronoi cell.

The straightforward algorithm for computing the nearest neighbor, discussed in the next section, has time complexity of $O(n)$, where n is the number of possible nearest neighbors. Using probabilistic *Lloyd's method* and assuming the straightforward algorithm for computing the nearest neighbor (in which the distance to each possible nearest neighbor is computed), we see that we make approximately $n * (n * C) = Cn^2$ nearest neighbor computations. Since computing the nearest neighbor is the most computationally intensive portion of the algorithm, probabilistic *Lloyd's method* has time complexity of $O(n^2)$, with n equal to the number of cell generators. Applications of centroidal Voronoi diagrams often demand thousands of generator points, and thus the resulting computation is often too slow for practical use. The author was therefore interested in finding a faster method of computing the nearest neighbor.

4 The Straightforward Algorithm

The most intuitive approach to computing the nearest neighbor is to compute the distances from α to each point in Ω and compare them. As this is the most straightforward algorithm, it is also the easiest to implement. On the following page is an implementation of the straightforward algorithm, referred to as *Method 1* for brevity, in MATLAB v5.2 [4].

As we increase the number of points in Ω by one, we must perform one extra distance calculation and one extra comparison. Since there is a constant amount of additional work with each extra point, *Method 1* has time complexity of $O(n)$. Aside from the extra initial storage that is needed to store the new

```
function nearest = nearestNeighbor( omega, alpha )

    distance = realmax;
    omegaSize = size( omega,1 );

    for i = 1:omegaSize
         distSq = sum( ( omega( i,:  )  - alpha ).^2 );
         if distSq < distance
              distance = distSq;
              nearest = i;
         end
    end
```

Figure 1: MATLAB implementation of *Method 1*

points, no additional memory is allocated as n increases. By examining the above MATLAB implementation, we see that no preprocessing of any sort is necessary. Below is a summary of these results.

	Preprocessing time	Preprocessing space	Time	Space	Accuracy
Method 1	NA	NA	$O(n)$	$O(1)$	100%

Table 1: An analysis of *Method 1*

It is evident that *Method 1* works for higher dimensions as well. As far as ease of extensibility, the code in Figure 1 will work for any dimension, provided that the input variables are of the appropriate form for the dimension. Each additional dimension requires only one extra subtraction operation, one extra squaring operation, and one extra addition operation during the distance calculation. We see then that the time and space complexity remain the same for higher dimensions. It is also evident that *Method 1* will find the correct nearest neighbor with 100% accuracy, and that the computation remains unchanged whether the points in Ω are uniformly distributed about the region or not.

Thus with respect to our metrics, the only drawback of *Method 1* is its time complexity. While $O(n)$ may not appear computationally intensive, it must be remembered that in applications of nearest neighbor search such as the construction of centroidal Voronoi tessellations, time complexity of $O(n)$ for computing the nearest neighbor may result in greater time complexities, such as $O(n^2)$, for the whole problem.

5 The Spiral Method

One alternative to the straightforward approach is the spiral method proposed by Bentley, Weide, and Yao, referred to as *Method 2*. *Method 2* involves dividing the region into grid boxes, or squares, of the same size. The number of grid boxes is proportional to the number of points in Ω. For simplicity, our implementation of *Method 2* generates a grid of g by g grid boxes where g is equal to $floor(\sqrt{numberofpoints/pointsperbox})$. This will give us a grid with approximately the number of points per grid box that we have specified. Say, for instance, that we decided upon four points per grid box. If there are one hundred possible nearest neighbor points in the region defined by $0 < x < 1$; $0 < y < 1$, then our grid might look like Figure 2 below.

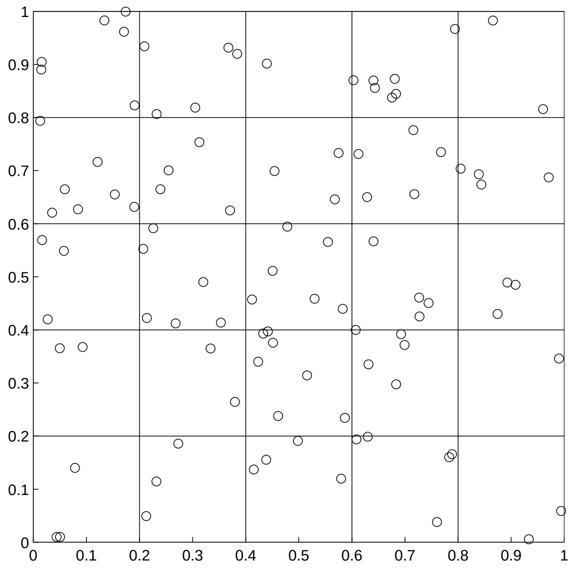

Figure 2: One hundred points in $0 < x < 1$; $0 < y < 1$, and the associated grid

Once the region is divided into a grid, the next step is to assign a grid box to each point in Ω, based upon that point's coordinates. A point $\beta \in \Omega$ is in grid box A if and only if the coordinates of β fall within the region defined by A. These two steps, the division of the region into a grid and the sorting of points in Ω into their appropriate grid boxes, or bins, constitute the preprocessing phase of *Method 2*. It is clear that since the amount of grid boxes is directly proportional to the number of points in Ω, the time and space complexity of the preprocessing phase of *Method 2* are both $O(n)$.

Once we have sorted the points in Ω into grid boxes, or bins, we have enough information to find the nearest neighbor in constant time. First, we locate the grid box that contains α. We then begin to check grid boxes that are close by for possible nearest neighbors. We begin by checking the box that contains α.

If no point is found, we then move on to the surrounding eight boxes. If still no point is found, we check the surrounding sixteen boxes. An intuitive approach to this search is to start with the box that contains α, and "spiral" outwards until a point is found, as is demonstrated in Figure 3 below.

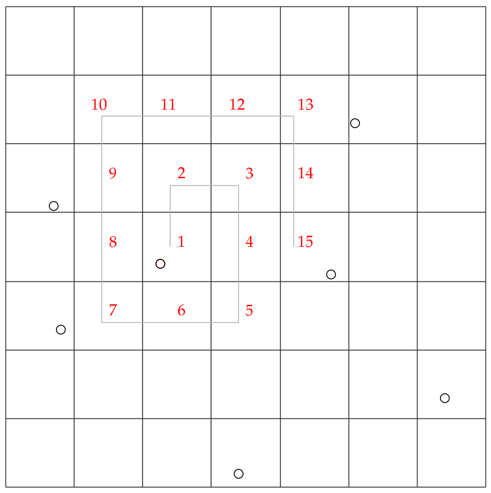

Figure 3: "Spiraling" outward from the center box until a point is found

Once a point has been found, we need only to compute the distances to each point that belongs to a grid box falling within the circle of radius defined by the distance between α and the point found, since we are guaranteed that the greatest distance between α and any point in Ω is at most the length of that radius. On average, this is done in constant time, since we expect to find a constant number of points per grid box regardless of the number of points in Ω. It is clear that we do not require any additional memory allocation as we increase the number of possible nearest neighbors, since we do not store any additional information. It is also evident that *Method 2* finds the correct nearest neighbor every time. Below is a comparison of the performances of *Method 1* and *Method 2*.

	Preprocessing time	Preprocessing space	Time	Space	Accuracy
Method 1	NA	NA	$O(n)$	$O(1)$	100%
Method 2	$O(n)$	$O(n)$	$O(1)$	$O(1)$	100%

Table 2: An analysis of *Method 1* and *Method 2*

The implications of the time complexity of *Method 2* are significant. As far as nearest neighbor searching in the problem of constructing centroidal Voronoi tessellations, the preprocessing phrase takes $O(n)$ time, and the actual neighbor searching only takes $O(n)$ time as well, since we must only perform $n * C$ constant-time computations. This reduces the time complexity of probabilistic *Lloyd's method* from $O(n^2)$ to $O(n)$.

As far as efficient performance with uniformly distributed points in two dimensions, the spiral method is superior to the other five algorithms presented in this paper. The problems with the algorithm are the difficulties that are encountered during its implementation and extension to higher dimensions, and non-optimal performance with distribution functions that are non-uniform. These three obstacles are now discussed in turn.

While the concept is simple, implementation of *Method 2* is more complicated. One of the complications that arises during implementation is the calculation of which grid boxes are intersected by the circle produced when the first point is found. This problem can be circumvented by instead considering every grid box that is contained within the square in which the circle is inscribed, but at the cost of efficiency. Another complication is the problem of implementing the "spiraling" outward from the center grid box. While this isn't a monumental problem, it can provide some difficulty and should be considered. An important issue related to the spiral algorithm's extensibility to other dimensions is the density of points per grid box. To understand why this is a problem, we must consider the following. In one dimension, we see that if we do not find a point in the first grid box, or line segment in this case, we must check two additional line segments, and then two more, et cetera. In two dimensions, if we do not find a point in the first bin, then we must check eight more, and then sixteen. For three dimensions, we have one, then twenty-six, then ninety-nine. In the general case, we must check $(2t-1)^d$ total grid boxes, where t is the outermost "layer" that we consider, or one less than the number of times we have failed to find a point in a layer, and d is the dimension. Thus it becomes increasingly costly to fail to find a point in a layer, though at the same time failure becomes increasingly unlikely. Increasing the point density of the grid boxes would suppress the problem of failing to find a point, but would increase the cost of computing the distances to each point in a specific grid box. In the extreme case, where the point density is maximum and there is one grid box, the algorithm is reduced to *Method 1.* Thus for each dimension, the implementor must find an optimal point-bin density. We expect that for some cases no changes would be necessary, but such results would have to be determined empirically for the specific problem in question. While in any case the spiral method performs in constant time, it must be remembered that the algorithm may not be practical if that constant is too large of a number.

Finally, we see that this method isn't necessarily optimal for point distributions that are not uniform. In the case where it is more likely for points to appear near the edges of a region than the center, for example, we may run into the following complication. If we assume that a point we choose is nearer to the center of the region than to any of its borders, then we most likely will not encounter a box with a point in it until we near the edges of the region. Once we do find a point, we must check every grid box that intersects with the corresponding circle. Since this circle is so relatively large, we expect it will encompass many grid boxes, and thus we will have to compute the distances to many points. For this reason, an implementor should keep the distribution function in mind when considering *Method 2*.

6 The Grid-Bin Method

The complexity involved in implementing *Method 2* resulted in the author's and Professor Burkardt's development of the grid-bin method (*Method 3*). *Method 3* is similar to *Method 2* in that it also involves a preprocessing step in which the region is divided into a grid and each point in Ω is assigned a grid box. *Method 3* involves an additional preprocessing step as well. The closest point in Ω to the corner of each grid box is calculated (via *Method 1*) and stored. Unfortunately, while this extra preprocessing step allows the nearest neighbor search to occur in constant time, the time complexity of the preprocessing is raised to $O(n^2)$. For each additional point in Ω, we add on average b grid boxes, where b is some constant. We then must perform an additional $4 * b * n$ *Method 1* nearest neighbor searches. It is evident that the space complexity of the preprocessing is linear, as with each additional point we require an additional constant amount of storage.

Once we have the points in Ω sorted into the grid-bin, we can find the nearest neighbor by computing into which grid box α falls, and calculating which of the one through four points in Ω assigned to the four corners of the grid box is closest to α. This search is performed in constant time with no additional memory allocation.

The grid-bin algorithm introduces the notion of *approximate nearest neighbor searching*. That is, *Method 3* is not guaranteed to find the nearest neighbor. It is possible to encounter a situation in which a point that lies within a grid box has a different nearest neighbor than any of the four corners of that grid box. Our implementation of *Method 3*, with approximately three points per box, found the correct nearest neighbor 98% of the time. Approximate nearest neighbor searching is suitable for some classes of problems, while for others the precise nearest neighbor is needed. Constructing probabilistic centroidal Voronoi tessellations, for example, is possible using an approximate nearest neighbor algorithm. Below is a comparison of the three algorithms we have now discussed.

	Preprocessing time	Preprocessing space	Time	Space	Accuracy
Method 1	NA	NA	$O(n)$	$O(1)$	100%
Method 2	$O(n)$	$O(n)$	$O(1)$	$O(1)$	100%
Method 3	$O(n^2)$	$O(n)$	$O(1)$	$O(1)$	98%

Table 3: An analysis of *Method 1*, *Method 2*, and *Method 3*

Method 3 presents a simpler, less efficient alternative to *Method 2*. The grid-bin method is relatively easy to implement. In addition, implementation is not complicated by extending the algorithm to higher dimensions, and the performance of the algorithm should remain comparable to its performance in two dimensions. *Method 3* should also perform well regardless of the distribution function, since it embodies the same simple search principle of *Method 1*.

One might question if *Method 3* has any advantage over *Method 1*. While *Method 3* has time complexity of $O(n^2)$, we must remember that this is a measurement of the preprocessing phase of the algorithm, which only occurs once. For instance, if we were to use *Method 1* to find the nearest neighbors of one thousand points for $|\Omega| = 100$, we would have to make $100 * 1000 = 100000$ distance calculations. If we were to use *Method 3*, on the other hand, and we assumed a point density of three points per box, then we would make approximately $36 * 100 = 3600$ distance calculations for the preprocessing phase ($5 \cdot 5$ grid boxes, $6 \cdot 6$ corners), and at most $4 * 1000 = 4000$ calculations during the neighbor searching. Thus we have 100000 distance calculations for *Method 1*, and only 7600 for *Method 3*. For some problems, then, *Method 3* may be more appropriate than *Method 1*. In the case of problems that are extremely computationally intensive in two dimensions, such as constructing centroidal Voronoi tessellations, *Method 2* would be preferable to both *Method 1* and *Method 3*.

7 The Grid-Bin Method with Sweeping Preprocessing

In order to increase the speed of the preprocessing phase of *Method 3*, the author implemented a revised version of the algorithm. The complication with *Method 3* is that for each additional point, we made on the order of n additional nearest neighbor computation. We want to reduce the number of additional nearest neighbor computations so that it is not directly dependent upon n. This is accomplished in the following manner.

We begin by dividing the region into grid boxes, the same way that we did for *Method 2* and *Method 3*. We also create a list of the points in Ω sorted by x-coordinate. Instead of computing the nearest neighbors of each of the box corners by brute force, we only compute the nearest neighbors of the first column of corner points (in the case of Figure 2, the points $\{(0.0,0.0), (0.0,0.2), (0.0,0.4), (0.0,0.6), (0.0,0.8), (0.0,1.0)\}$). For each succeeding column of corner points, we compute the set of nearest neighbors using only a subset of the points in Ω (approximately of size $\sqrt{n}$), which we determine based upon the set of nearest neighbors obtained from the last column of corner points and the relative x-position of the current column of corner points. Thus we severely reduce the number of nearest neighbor computations. (It should be noted that the above description is a summary of the algorithm, as the complexity of the details prevents a full description of the algorithm from being concisely expressed. For more information on the details of the algorithm, please contact the author.)

The number of grid boxes $g \cdot g$ in our grid is on the order of n. For each column of corner points, then, we have approximately $\sqrt{n}$ points, for each of which we make approximately $\sqrt{n}$ nearest neighbor computations (as a result of our improved algorithm). Since there are approximately $\sqrt{n}$ columns, our preprocessing time is expected to be $\sqrt{n} * \sqrt{n} * \sqrt{n} = n^{3/2} + n \log n$ (we require $n \log n$ time to sort the n points in Ω by x-coordinate). Therefore the time complexity of the preprocessing phase of *Method 4* is $O(n^{3/2})$. It is clear that the space complexity of *Method 4* is $O(n)$, since the amount of information we store is dependent on the number of grid boxes we have.

Once we have completed the preprocessing phase, we proceed in the same manner that we did for *Method 3*. The space and time complexity of the neighbor searching phase of the algorithm, then, are both $O(1)$. As with the previous method, this is an approximate nearest neighbor algorithm. Since there is a chance that not only do the four corner points have different nearest neighbors than does α, but that we did not compute the correct nearest neighbors for the four corner points, our percentage error is compounded. Using our implementation of *Method 4* and three points per grid box, we found the correct nearest neighbor about 93% of the time. Below is a summary of these results.

	Preprocessing time	Preprocessing space	Time	Space	Accuracy
Method 1	NA	NA	$O(n)$	$O(1)$	100%
Method 2	$O(n)$	$O(n)$	$O(1)$	$O(1)$	100%
Method 3	$O(n^2)$	$O(n)$	$O(1)$	$O(1)$	98%
Method 4	$O(n * \sqrt{n})$	$O(n)$	$O(1)$	$O(1)$	93%

Table 4: An analysis of *Methods 1 - 4*

Despite the complications in describing *Method 4*, implementation is fairly straightforward. The details of the algorithm require more attention than the previous method, but we do not foresee any major problems. As far as extensibility to other dimensions, we expect that it can be done without much complication. We also expect the time and space complexity to remain unchanged for higher dimensions.

Method 4, though, is largely dependent on a uniform distribution of points. We could not make the assumptions that we do based upon the x-coordinates of the points in Ω if our points were not distributed uniformly. Perhaps the algorithm could be altered so that assumptions are made based upon another distribution function. For instance, if we used a distribution function where points were more likely to appear in the center of the region than toward the edges, we might allow more nearest neighbor computations to be made for the columns near the center of the region. As it stands, though, the implementation of *Method 4* that we present would not perform well with non-uniform distribution functions.

8 The Delaunay Triangle Neighbor Search

The last two algorithms presented in this paper involve the Delaunay triangulation. A Delaunay triangulation of set of points Ω is a triangulation of those points such that no fourth point is contained within the circumcircle of the triangle defined by any three triangulated points. The Delaunay triangulation can also be described as the *dual* of a Voronoi diagram, and can be constructed by connecting each node of a Voronoi diagram with each of the nodes of the adjacent Voronoi regions. Below is a Delaunay triangulation of twenty points in the region $0 < x < 1; 0 < y < 1$.

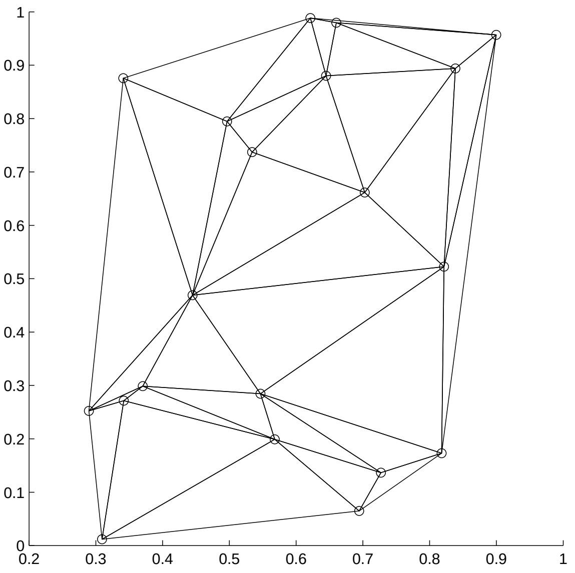

Figure 4: A Delaunay triangulation of twenty points in the region $0 < x < 1; 0 < y < 1$

The preprocessing phases of *Method 5* and *Method 6* involve constructing the Delaunay triangulation. It has been shown that the Delaunay triangulation in the plane can be computed in $O(n \log(n))$ time, using $O(n)$ storage. For a proof of the above assertion and a more detailed explanation of Delaunay triangulations, see [5]. Once we have generated the Delaunay triangulation, we are ready to compute the nearest neighbor. The author's implementation of *Method 5* is based on the implementation offered by Barry Joe in the GEOMPACK [6] software package. To understand the algorithm, we must first understand the concept discussed below.

Given a point δ and a line in the plane, we can define a function f such that $f(\epsilon) = 1$ for some point ϵ if and only if that point lies on the line, or on the same side of the line as does δ, and $f(\epsilon) = 0$ otherwise. With three such functions, one for each line of a triangle, we can determine whether or not a given point is inside that triangle. In addition, we can also determine where the point is relative to the triangle by analyzing the values of f. For instance, if the three values of our three functions are positive, then we know that the point lies within the triangle. If one value is negative, then we know that the point is "behind" that side of the triangle. *Method 5* begins by picking a triangle at random. We then pick one side of the triangle, and determine the value of f with respect to that side and α. If that value is positive, we move clockwise to the next side. If it is negative, we have an idea of where α is relative to our triangle, and we move repeat the process with the triangle that neighbors the current triangle on that side. If all three values of f are positive, then we have found the triangle in which α lies. Once we have found the correct triangle, we simply check which of the three vertices of the triangle is closest to α. The closest vertex is assumed to be the nearest neighbor, although as mentioned below, the triangle search is an approximate nearest neighbor algorithm, and so we are not guaranteed.

We know that the time complexity of the preprocessing phase of *Method 5* is $O(n \log(n))$ and the space complexity is $O(n)$. As with the other algorithms, the space complexity of neighbor search is $O(1)$, since no additional memory allocation is required as n increases. The probabilistic time complexity of the neighbor search is $O(\sqrt{n})$. This is shown as follows. The number of triangles in a Delaunay triangulation is never more than $2n$, where n is the number of points in the triangulation. Consider the region of size $g \cdot g$. In the x-dimension, it is clear that we expect to have $\sqrt{2n}$ triangles per length g. The same is true for the y dimension. Two points in two triangles selected at random are on average separated by a distance of $g/2$ in the x-dimension, and $g/2$ in the y-dimension (in other words, two points selected at random from a line segment of length s are on average separated by a distance of $s/2$). Thus two such points are approximately separated by a distance of $\sqrt{(g/2)^2 + (g/2)^2} = g$. Since there are approximately $\sqrt{2n}$ triangles per distance g, we expect to traverse on average $\sqrt{2n}$ triangles, which is on the order of $\sqrt{n}$.

The Delaunay triangle neighbor search is an approximate nearest neighbor algorithm. We are guaranteed to find the correct triangle if we are careful with the boundary cases, but there are situations where the nearest neighbor of a point within a triangle is not one of the three vertices of the triangle. The accuracy in this case is dependent upon the density of points in the region. If there are a relatively large number of points in the region, then the Delaunay triangles are smaller, and there is a better chance that any triangle containing α has a vertex that is its nearest neighbor. For one hundred possible nearest neighbor points in the region defined by $0 < x < 100$; $0 < y < 100$, *Method 5* found the correct nearest neighbor approximately 92% of the time. For one thousand possible nearest neighbors in the same region, the algorithm was approximately

94% accurate. We summarize these results on the following page. For ease of comparison, we assume 94% accuracy for *Method 5*.

	Preprocessing time	Preprocessing space	Time	Space	Accuracy
Method 1	NA	NA	$O(n)$	$O(1)$	100%
Method 2	$O(n)$	$O(n)$	$O(1)$	$O(1)$	100%
Method 3	$O(n^2)$	$O(n)$	$O(1)$	$O(1)$	98%
Method 4	$O(n\sqrt{n})$	$O(n)$	$O(1)$	$O(1)$	93%
Method 5	$(n\log(n))$	$O(n)$	$O(n\sqrt{n})$	$O(1)$	94%

Table 5: An analysis of *Methods 1 - 5*

Computing the values of each of the functions f is straightforward, as is choosing the appropriate triangle once the values have been computed. The only complication that may arise during implementation of *Method 5* is dealing with boundary cases. For example, consider the situation where the position of α dictates that two of our f values are negative. Depending on the shape of the region, our current triangle could have no neighboring triangle on one of the sides that the f values specify. If this happens, our algorithm should simply choose to follow the path specified by the other negative f value. If this is accounted for, then the properties of the Delaunay triangulation guarantee that we will find the correct triangle [6].

The generalized expected time for computing the Delaunay triangulation is $O(n\log n + n^{\lceil d/2\rceil})$ ([5]). Thus *Method 5* in higher dimensions is impractical for most applications. However, it can be shown using a similar argument as the one we used to prove that the time complexity of *Method 5* in two dimensions is $O(\sqrt{n})$ that the generalized time complexity of *Method 5* is $O(n^{1/d})$. Therefore if the time it takes to compute the overhead of the preprocessing phase can be sacrificed, we expect the triangle search to perform well in higher dimensions.

While the triangle search is not necessary limited to uniformly distributed points, there are some distributions that compromise the algorithm. For example, a distribution function that tended to place points along a straight line would produce a similar line of Delaunay triangles. In this case, the time complexity of the algorithm is increased from $O(\sqrt{n})$ to $O(n)$.

The last algorithm presented in this paper is an improved variation of this algorithm for the general nearest neighbor problem. However, this algorithm might be appropriate for certain types of problems that require not only the nearest neighbor, but also the Delaunay triangle in which α is contained.

9 The Delaunay Node Neighbor Search

Our final algorithm is the Delaunay node neighbor search. A detailed discussion of the node neighbor search (*Method 6*) can be found in [7]. Like the previous algorithm, the preprocessing phase of *Method 6* involves generating the Delaunay triangulation. Once we have done this, we proceed as follows. First, we pick a point β in Ω at random. We then determine if α is closer to β than to any of β's neighbors (connected nodes in the Delaunay triangulation). If this is so, then we are finished, and β is the nearest neighbor. If not, then we repeat this process with β now equal to the neighbor that was closest to α. Essentially, we perform a greedy search that determines the local nearest neighbor and iterate until we find the global nearest neighbor. Using the same argument as we did for *Method 5*, the probabilistic time complexity of *Method 6* is $O(\sqrt{n})$. The straightforward implementation of this algorithm, as described above, requires no additional memory allocation as n increases. The properties of the Delaunay triangulation ensure that we will find the correct nearest neighbor every time. Below is a final summary of our results.

	Preprocessing time	Preprocessing space	Time	Space	Accuracy
Method 1	NA	NA	$O(n)$	$O(1)$	100%
Method 2	$O(n)$	$O(n)$	$O(1)$	$O(1)$	100%
Method 3	$O(n^2)$	$O(n)$	$O(1)$	$O(1)$	98%
Method 4	$O(n\sqrt{n})$	$O(n)$	$O(1)$	$O(1)$	93%
Method 5	$(n\log(n))$	$O(n)$	$O(\sqrt{n})$	$O(1)$	94%
Method 6	$(n\log(n))$	$O(n)$	$O(\sqrt{n})$	$O(1)$	100%

Table 6: An analysis of *Methods 1 - 6*

Implementation of this algorithm is straightforward. One problem we expect to run into is the redundant calculation of distances, since multiple nodes will share neighbors. This can be avoided by keeping track of every node and its distance to α. This will reduce the amount of time that it takes for the algorithm to execute (though it will not reduce the probabilistic time complexity), but will increase the probabilistic space complexity to $O(\sqrt{n})$. Either variation might be appropriate, depending on the requirements of the problem.

As mentioned above, construction of the generalized Delaunay triangulation takes $O(n\log n + n^{\lceil d/2\rceil})$ time, rendering this algorithm inefficient for large dimensions. As with the previous method, the probabilistic time complexity of *Method 6* is expected to be $O(n^{1/d})$, where d is the dimension. Thus as with *Method 5*, the algorithm performs relatively better in higher dimensions. Therefore if we can once again sacrifice the overhead of the preprocessing time, *Method 5* might be an appropriate algorithm for problems in higher dimensions. We expect to encounter the same problems with this algorithm's dependence upon the distribution function as we did with the previous method. It is evident then that for the general problem of computing the nearest neighbor, the node neighbor search is an improvement upon the less accurate triangle neighbor search.

10 Conclusions and Discussion

The purpose of the study presented in this paper is to offer both a comparison of six approaches to the nearest neighbor problem and a set of metrics by which to judge further approaches. The author became acquainted with nearest neighbor searching while working on the problem of constructing probabilistic centroidal Voronoi tessellations, and decided that the subject was interesting after witnessing how the straightforward approach performs under computationally intensive conditions. The author and Professor John Burkardt of Iowa State University proceeded to study four existing nearest neighbor algorithms and develop two more, and determine which metrics were most important in evaluating these algorithms. The results presented in this paper are meant to serve as a guide to anyone who is interested in implementing a nearest neighbor search.

The nearest neighbor problem takes several different forms, and for simplicity the author decided to consider the problem in two dimensions for uniformly distributed points in a square region. In some respects, there is little or no difference between the simplified problem that the author presents and many of the other instantiations of the problem. In other respects, however, the simplified model discussed in this paper serves as a *probabilistic* study of the general problem, in which trends in the simple model may be generalized and compared with similar trends in related problems. Some instantiations of the nearest neighbor problem are so far removed from our simple model, though, that our discussion can only be used as a general heuristic from which we can only draw very general conclusions. It is the goal of this paper to at least in some sense help facilitate the choosing and implementation of a nearest neighbor algorithm, even if that facilitation is minimal.

Each of the metrics that are presented in this paper and the algorithms that perform best with respect to those metrics are now briefly discussed. As for time complexity in two dimensions, the spiral method (*Method 2*) is the most efficient, with preprocessing time of only $O(n)$ and processing time of only $O(1)$. Space complexity is not as important of an issue; all of the algorithms presented have space complexity of at most $O(n)$. If it is imperative that minimal space is used, though, then the straightforward method (*Method 1*) should be considered. Three of the six algorithms (*Method 1, Method 2,* and *Method 6*) are guaranteed to find the precise nearest neighbor, which is an important feature for a large class of problems. The straightforward method is the simplest to implement, but the grid-bin methods (*Method 3* and *Method 4*) are simple as well, if approximate neighbor searching is sufficient. The algorithms based on the Delaunay triangulation (*Method 5* and *Method 6*) are relatively simple to implement, excluding construction of the Delaunay triangulation in higher dimensions. The straightforward algorithm and the grid-bin methods are easily extended to higher dimensions. The spiral method performs well in higher dimensions if the problems presented in Section 5 are considered. The straightforward method and the grid-bin method will perform just as well with any distribution function as they do for the uniform distribution assumed in this paper. The grid-bin method with sweeping preprocessing can be adjusted to perform well with other distribution functions. The spiral method and the Delaunay methods will work well with some distribution functions, but not, as demonstrated in this paper, with others.

The environment in which one is working should also be taken into consideration. For example, while working on the problem of constructing centroidal Voronoi tessellations in MATLAB, the author decided upon the Delaunay node neighbor search because MATLAB provides construction of the Delaunay triangulation. MATLAB also has a procedure called *dsearch*, which finds the nearest neighbor given α, Ω, and the Delaunay triangulation of Ω. The author presumes that *dsearch* is based upon the Delaunay neighbor node search. It is evident, then, that it might be in the best interest of the implementor to take advantage of the resources presented by the work environment, depending on how much time the implementor wishes to spend on the nearest neighbor aspect of his problem.

There are doubtlessly several other approaches to the nearest neighbor problem, each of which might be better suited to one class of problems than are any of the six algorithms presented in this paper. While these six algorithms may not be directly suited to the purposes of every reader, the metrics provided should be useful in evaluating how appropriate a specific algorithm is, regardless of whether or not it is presented here. With that in mind, it is the author's hope that this paper will assist the reader in deciding which nearest neighbor algorithm is most appropriate for the reader's purposes.

References

[1] Bentley, J.L., Weide, B.W., Yao, A.C., Optimal Expected-Time Algorithms for Closest Point Problems *ACM Trans. Math. Softw. 6, 4 (Dec. 1980), 563-580.*

[2] Knuth, D.E., The Art of Computer Programming Volume 1: Fundamental Algorithms *Addison-Wesley (1968)*

[3] Gunzberger, M., Du, Q., Faber, V. Centroidal Voronoi Tessellations: Applications and Algorithms *SIAM Review 41 (1999) 637-676*

[4] Matlab is a software product of TheMathWorks, Natick, MA Web page: http://www.mathworks.com

[5] de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O., Computational Geometry: Algorithms and Applications *Springer-Verlag (2000)*

[6] Joe, B., "GEOMPACK - A Software Package for the Generation of Meshes Using Geometric Algorithms," *Advances in Engineering Software, Elsevier, Vol 13, Num 5 (1991) 325-331*

[7] Green, P. J., Sibson, R., Computing Dirichlet Tessellations in the Plane *The Computer Journal Vol 21 (1978) 168-173*