

Practical Nearest Neighbor Search in the Plane

Michael Connor and Piyush Kumar

Department of Computer Science, Florida State University
Tallahassee, FL 32306-4530

{mic Connor, piyush}@cs.fsu.edu

Abstract. This paper shows that using some very simple practical assumptions, one can design an algorithm that finds the nearest neighbor of a given query point in $\mathcal{O}(\log n)$ time in theory and faster than the state of the art in practice. The algorithm and proof are both simple and the experimental results clearly show that we can beat the state of the art on most distributions in two dimensions.

Keywords: Nearest Neighbor Search, Delaunay Triangulation, Morton ordering, Randomized algorithms.

1 Introduction

Nearest neighbor search is a fundamental geometric problem important in a variety of applications including data mining, machine learning, pattern recognition, computer vision, graphics, statistics, bioinformatics, and data compression [7, 1]. Applications of the nearest neighbor problem in the plane are particularly motivated by problems in Geographic Information Systems [17].

Linear space, $\mathcal{O}(n \log n)$ pre-processing and $\mathcal{O}(\log n)$ query time algorithms are known but seem to have large constants [13, 9]. $(1 + \epsilon)$ -approximation algorithms seem to be faster than the exact algorithms in practice [16]. When one requires exact answers, one of the most practical algorithms available for this problem with provable guarantees is due to Devillers [10, 4]. Very recently, Birn et al. [3] have announced a very practical algorithm for this problem which beats the state of the art [16, 10] in query times but unfortunately does not have any worst case query time guarantees better than linear. In this paper, we present an algorithm which is faster in both pre-processing as well as query times compared to [3] on most distributions. We also show that if the query comes from a doubling metric, our queries are bounded by $\mathcal{O}(\log n)$ query time in expectation.

For the nearest neighbor search problem, the seminal work of Arya et al. [2] is the de facto standard for distribution independent approximate nearest neighbor search problems in fixed dimensions. Their C++ library ANN [16] is very carefully optimized both for memory access and speed, and hence has been the choice of practitioners for many years (in various areas). ANN's optimized kd-tree implementation has only recently been beaten by Birn et al. [3] in two dimensions. Birn et al. [3] also report that CGAL's recent kd-tree implementation is competitive with ANN's.

Our work is strongly related to some previous algorithms on Morton ordering [5, 6, 8], compass routing [14] and Delaunay triangulations [3].

One of the first properties we exploit in this paper is the following: We have shown in previous work [8] that if a point cloud P is shifted using a random shift, one can achieve an expected constant factor approximation for the nearest neighbor of a query point q by just locating q using a Morton order binary search in P and checking $\mathcal{O}(1)$ points around the location returned by the binary search.

Another interesting feature that we use is the fact that compass routing is supported by Delaunay Triangulations [14, 3]. It was shown by Kranakis et al. [14] that if one wants to travel between two vertices s and t of a Delaunay triangulation, one can only use local information on the current node and the coordinates of t to reach t , starting from s . The routing algorithm is simple, the next vertex visited is the one whose distance to t is minimum amongst the vertices connected to the current vertex. This type of local greedy routing has also been studied as the ‘small-world phenomenon’ or the ‘six degrees of separation’ [15]. Recently, this routing algorithm was successfully used for nearest neighbor searching [3] and we base our algorithm on this observation as well.

The key contributions of this paper are:

1. Compared to previous compass routing nearest neighbor techniques, our pre-processing speed on data sets is faster [3].
2. Compared to the state of the art, our query times are faster on most data sets [3, 16].
3. Our implementation has provable expected logarithmic query time.

The main drawbacks of our approach are:

1. The algorithm is randomized and the bounds are expected. To bound the query time, we assume that the query point has an expansion constant $\gamma = \mathcal{O}(1)$. This is described in depth in Section 3.
2. The algorithm is only suited for two dimensions because of its dependence on Delaunay Triangulations which have quadratic space complexity in dimensions greater than two.
3. We assume that each coordinate of the input points fits in a word, and operations on words like XOR and MSB can be performed in constant time [6, 11, 8].

The paper is organized as follows: In the remainder of this section, we define our notation. The next section gives the outline and description of our algorithm. It also briefly describes some of the tools that we use in the algorithm. Section 3 analyzes the computational complexity for our algorithm. Section 4 describes the experimental setup we use. Section 5 presents our experimental results. Section 6 concludes the paper.

Notation: Points are denoted by lower-case Roman letters. $\text{Dist}(p, q)$ denotes the distance between the points p and q in L_2 metric. P is reserved to refer to a point set. n is reserved to refer to the number of points in P .

We write $p < q$ iff p precedes q in Morton order ($>$ is used similarly). We use p^s to denote the shifted point $p + (s, s, \dots, s)$. $P^s = \{p^s | p \in P\}$. p_i is the i -th point in the sorted Morton ordering of the point set.

Upper-case Roman letters are reserved for sets. Scalars except for c , d , m and n are represented by lower-case Greek letters. We reserve i, j, k for indexing purposes.

$\text{Ball}(c, r)$ is used to denote a ball with center c and radius r . We also use $\text{Ball}(p, q)$ to represent the diametral ball of p and q . For point q , let \mathcal{N}_q^k be the k points in P , closest to q . $|\cdot|$ denotes the cardinality of a set or the number of points in P inside the geometric entity enclosed in $|\cdot|$. Let, $\text{nn}(p, \{\})$ return the nearest neighbor of p in a given set. Finally, $\text{rad}(p, \{\})$ returns the distance from point p to the farthest point in a set.

2 The Algorithm

Algorithm 1 describes both our nearest neighbor pre-processing as well as the query algorithm. Our pre-processing algorithm essentially splits the input point set P into three layers. First the Delaunay triangulation, G , of P is computed and a maximal independent set computed on this graph. P' is the set $P \setminus \{\text{Maximal Independent set of } G\}$. We construct Layer 1, with points P' sorted in Morton order, Layer 2, with the Delaunay triangulation of points P' , and Layer 3, with edges connecting the points in Layer 2 to a maximal independent set of points. (see Figure 1).

Algorithm 1. Nearest Neighbor Algorithm

Require: Randomly shifted point set P of size n . Morton order compare operator $<$.

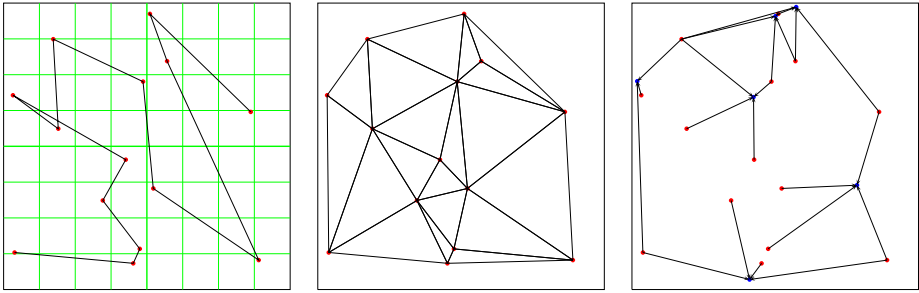
```

1: procedure PREPROCESS( $P$ )
2:    $G = (P, E) \leftarrow$  Delaunay Triangulation of  $P$ 
3:    $P' = P \setminus \{\text{Maximal Independent set of } G\}$ 
4:    $P' \leftarrow \text{Sort}(P', <)$ 
5:    $G' = (P', E') \leftarrow$  Delaunay Triangulation of  $P'$ 
6:   for all  $p \in P'$  do:
7:      $H(p) \leftarrow \{q | e = (p, q) \in G \text{ where } q \in P \setminus P'\}$ .
8:   for all  $F$  in  $\{G', H\}$ 
9:     for all  $v \in F$  and  $\text{degree}(v) = \Omega(1)$  do:
10:      Pre-process VoronoiCell( $v, F$ ) for fast lookups and jumps.
11: end procedure

12: procedure COMPASSROUTING(point  $v$ , point  $q$ , Graph  $G = (P, E)$ )
13:   Require:  $v \in G$ .
14:   repeat
15:     If  $\text{degree}(v) = \Omega(1)$  then
16:       If  $q \in \text{VoronoiCell}(v, G)$  then return  $v$ 
17:       else: Update  $v$  using preprocessed VoronoiCell( $v, G$ ).
18:     else:
19:       for all  $v' \in G$  incident on  $v$  do:
20:         If  $\text{Dist}(v', q) < \text{Dist}(v, q)$  then:
21:            $v \leftarrow v'$ ; break
22:   until No improvement found
23: end procedure

24: procedure QUERY(point  $q$ )
25:    $i' \leftarrow \text{BINARYSEARCH}(P', q, <)$ 
26:    $p'_i \leftarrow \text{nn}(q, \{p'_{i-\eta}, \dots, p'_{i+\eta}\})$  where  $\eta = \mathcal{O}(1)$ 
27:    $p'_j \leftarrow \text{COMPASSROUTING}(p'_i, q, G')$ 
28:   return  $\text{nn}(p'_j, H(p'_j))$  // Uses preprocessed VoronoiCell( $p'_j, H$ ) if  $|H(p'_j)| = \Omega(1)$ 
29: end procedure

```



(a) The first layer, consisting of the non-maximal independent set vertices sorted in Morton order. Queries are processed using a binary search to find an approximate nearest neighbor ball.

(b) The second layer, consisting of the points in the first layer, and the edges of their Delaunay triangulation. The queries are processed by using compass routing, starting at the nearest point found in the previous layer, and ending at the nearest neighbor in this layer.

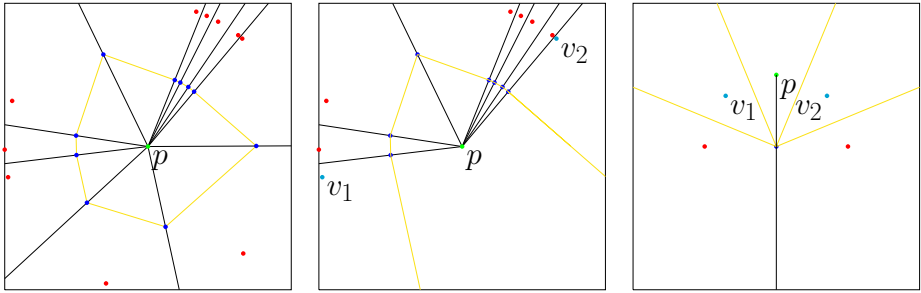
(c) The final layer, consisting of edges that connect the points in the second layer to the points in the maximal independent set. In this layer, we refine the nearest neighbor found in the previous step by scanning those points adjacent to it in this graph. This results in the final answer.

Fig. 1. The three layers of the query algorithm

Our pre-processing is complete unless there is a point p in Layer 2 or 3 with large degree ($\Omega(1)$). In these cases, we compute the vertices of the Voronoi cell of the point p along with the rays emanating from p and going through these vertices, partitioning the space around p into sectors. In order to locate a point closer to our query, we locate it in a sector, and compare the distance to a vertex found there (Figure 2). Note that any point in the plane can be located in these sectors in $\mathcal{O}(\log n)$ time using a binary search.

Our query algorithm starts by locating the query point q in the Morton ordering of points in Layer 1 (P'). Then it scans $\eta = \mathcal{O}(1)$ points around the location returned using binary search and finds the closest point p'_i to q among those points. COMPASSROUTING, which we describe next is used to find the nearest neighbor of q in Layer 2 starting from p'_i . Let this point in Layer 2 be called $p'_j \in P'$. We then find the nearest neighbor of q in Layer 3 and return the answer. We now describe how we do COMPASSROUTING and give the details of handling large degree vertices in Layers 2 and 3.

Our COMPASSROUTING algorithm is simple. Assuming there are no high degree vertices in Layer 2, it starts with a point v and selects the closest point to q that is incident on v . If there are no such vertices, it declares v as the solution, or else it jumps to the next point and repeats. In case it hits a point p that has large degree, it has access to the Voronoi rays emanating from p . It first locates q in the ray system of p in $\mathcal{O}(\log n)$ time using a binary search and then tests whether q lies in the Voronoi cell of p . If this is the case, p is returned as the answer. Otherwise the point opposite to p in the sector containing q is nearer to q compared to p and hence we jump to that point and continue routing.



(a) Here we see the center vertex p , and the sectors defined by rays passing through the Voronoi vertices. To find a nearer neighbor, we locate which sector the query lies in (via a binary search), then check the distance to the adjacent point that lies in that sector.

(b) In the first degenerate case, the center vertex has an open Voronoi cell. If the query point lies in this sector, we check the distance to the two points in the open sector (v_1, v_2).

(c) In the second degenerate case, the center vertex is co-circular with several adjacent vertices, and there is only one Voronoi vertex. In this case, we always check the nearest co-circular points in the clockwise and counter-clockwise directions for a nearer neighbor (v_1, v_2), and ignore the other co-circular points.

Fig. 2. The three cases for linear degree vertices

One optimization we implemented in COMPASSROUTING was to jump to any neighbor of v closer to q instead of jumping to the neighbor of v closest to q (also used by Birn et al. [3]). This fortunately does not have any effect on either the correctness or the running time analysis, but got us a slight improvement in the overall running time of the query.

In case of a vertex in Layer 3 with large degree, we again jump to the preprocessed Voronoi cell of p'_j with respect to the points in $H(p'_j)$ and use the same trick as in Layer 2. The nice property that we use in this case is that, if q does not lie in the Voronoi cell of p'_j , then the point opposite to p'_j in the sector containing q is the nearest neighbor of q , as opposed to just being a nearer neighbor. Note that using the maximal independent set does not actually improve the running time of the algorithm, it just reduces the total number of distance calculations we must do in practice.

It should be noted that there are two degenerate cases when considering the Voronoi cell of p , both of which can be resolved in constant time. The first case occurs when a sector contains an open face of the Voronoi cell (Figure 2b). This sector will contain two adjacent vertices, not one, and if q lies in such a sector, we simply compute the distance to both.

The second degenerate case occurs when an edge of the Voronoi cell of p has length 0 (Figure 2c), implying that some or all of the points bordering p are co-circular with p . In this case, we can identify in pre-processing the points immediately clockwise and counter-clockwise to p on the circle. If q is determined to lie in a sector bordering an

edge with 0 length, one of those two found points must be nearer to the actual nearest neighbor than p .

3 Analysis of the Algorithm

Let P be a finite set of points in \mathbb{R}^d such that $|P| = n$. For the purpose of this section, we will assume that both the query point as well as P are randomly shifted using a random point (s, s, \dots, s) . Let μ be a counting measure on P . Let the measure of a ball, $\mu(\text{Ball}(c, r))$ be defined as the number of points in $\text{Ball}(c, r) \cap P$. A point q is said to have expansion constant γ if for all $k \in (1, n)$:

$$\mu(\text{Ball}(q, 2 \times \text{rad}(q, \mathcal{N}_q^k))) \leq \gamma k$$

This is a similar restriction to the doubling metric restriction on metric spaces [7, 12, 8]. Throughout the analysis, we will assume that our query point q has an expansion constant $\gamma = \mathcal{O}(1)$. Note that for finding exact nearest neighbors in $\mathcal{O}(\log n)$ time, the queries with high γ are precisely the queries which drive provable $(1 + \epsilon)$ -approximate nearest neighbor data structures to spend more time in computing the solution when ϵ is close to zero [16].

We first begin by defining **Compass Routing** formally (our definition is slightly different compared to [14]): Given a geometric graph $G = (P, E)$, an initial vertex $s \in G$ and a destination q (may not be in the graph), let v_i be the closest vertex in G to q . We want to travel from s to v_i , when the only information available to us at any point in time is the coordinates of our destination, our current position, and the edges incident at the vertex we are located at. Starting at s , we will traverse the edge $(s, s') \in E$ incident on s that leads us closest to q . We assign $s = s'$ and repeat this procedure till we can no longer continue decreasing the distance to q . In the next Lemma, we prove a simple property of Compass Routing on Delaunay Graphs in d -dimensions.

Lemma 3.1. *Let $P \subset \mathbb{R}^d$ and $G = (P, E)$ be the graph output from its Delaunay triangulation. Let q be a query point for which we want to compute the nearest neighbor in P . Compass routing on G yields nearest neighbor of q in P .*

Proof. Let the compass routing begin with a vertex $v_0 \in P$. Let v_i be the vertex on which compass routing stops and can not improve the distance to q . Let $\text{Nbr}(v_i)$ be the set of all vertices having an edge with v_i in G .

This implies that $\text{Ball}(q, \text{Dist}(q, v_i))$ is empty of vertices in $\text{Nbr}(v_i)$. For a contradiction, let $v^* \neq v_i$ in P be the nearest neighbor of q . Then $v^* \in \text{Ball}(q, \text{Dist}(q, v_i))$ and there is no edge between v^* and v_i in G .

We will now draw a ball with v_i and v^* on its boundary such that it lies inside $\text{Ball}(q, \text{Dist}(q, v_i))$. If this ball is empty, then $v^* \in \text{Nbr}(v_i)$ which is a contradiction of the Delaunay property of the graph. Otherwise, we shrink the ball keeping it hinged on v_i and inside $\text{Ball}(q, \text{Dist}(q, v_i))$, till it contains only one point $v_j \in P$. This again is a contradiction since v_j is closer to q than v_i and $(v_i, v_j) \in G$ (Compass routing should not have terminated at v_i). See Figure 3. □

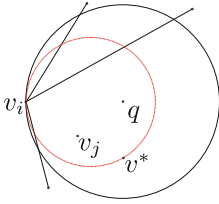


Fig. 3. Proof of Lemma 3.1

The next Lemma proves that our query Algorithm returns correct answers.

Lemma 3.2. *The QUERY function in Algorithm 1 returns the correct nearest neighbor of q in P .*

Proof. We mainly need to prove that the correctness of our query algorithm is not affected by separating P into three layers. Lemma 3.1 ensures that we find the nearest neighbor in Layer 2. Let this neighbor be p'_j . Hence $\text{Ball}(q, \text{Dist}(q, p'_j))$ is empty of points in Layer 2. If $\text{Ball}(q, \text{Dist}(q, p'_j))$ is empty, then p'_j will be returned as the nearest neighbor of q correctly by the QUERY function. Otherwise, $|\text{Ball}(q, \text{Dist}(q, p'_j))| = 1$ because if there were more points than 1 in this ball, there would exist a Delaunay edge between two of these points contradicting the fact that they are a maximal independent set in the Delaunay triangulation of P . Let v^* in Layer 3 be inside $\text{Ball}(q, \text{Dist}(q, p'_j))$ in this case. Then we can draw an empty ball passing through p'_j and v^* keeping it inside $\text{Ball}(q, \text{Dist}(q, p'_j))$. This means there must be a Delaunay edge connecting p'_j and v^* implying that $v^* \in H(p'_j)$ and hence the QUERY function must return v^* correctly. \square

The following two observations help us bound the running time of our query, assuming that q has expansion constant $\gamma = \mathcal{O}(1)$:

Lemma 3.3. *In $\mathcal{O}(\log n)$ time, $\text{Ball}(q, r)$ can be computed such that, in expectation, $|\text{Ball}(q, r)| = \mathcal{O}(1)$.*

Proof. This follows from a lemma by Chan [6] (and is explicitly proved in Lemmas 2.1-2.3 of [8]) that shows the nearest neighbor to q chosen from $\mathcal{O}(1)$ points in P adjacent to q in Morton order is contained in a box that has, in expectation, only a constant factor more points than the box containing $\text{nn}(q, P)$. \square

Theorem 3.4. *Given q , with expansion constant $\gamma = \mathcal{O}(1)$, $\text{nn}(q, P)$ can be found in $\mathcal{O}(\log n)$ time in expectation.*

Proof. Given that P is sorted in Morton order, a binary search for q obviously takes only $\mathcal{O}(\log n)$ time. This yields a ball to be refined with only expected $\mathcal{O}(1)$ vertices of the Delaunay triangulation of P . Compass routing can therefore give us a path containing only $\mathcal{O}(1)$ vertices. Given that any vertex can be processed in $\mathcal{O}(\log n)$ time to find a nearer neighbor by using the Voronoi cell, $\text{nn}(q, P)$ can be found in $\mathcal{O}(\log n)$ time in expectation. Note that splitting P in two layers, does not increase the running time because the number of points visited is still expected $\mathcal{O}(1)$ (By Lemma 3.3). \square

The construction time of the algorithm is $\mathcal{O}(n \log n)$, bounded by the sorting of the input set in Morton order, as well as constructing the Delaunay graph and Voronoi graph, all of which have $\mathcal{O}(n \log n)$ running times. The maximal independent set is found in $\mathcal{O}(n)$ time.

While this proof is independent of dimension, the practicality of the algorithm is questionable in dimensions higher than two, where the Delaunay graph is not constrained to have a linear number of edges. It remains to be seen if there is a practical solution to the nearest neighbor problem using this approach in dimensions higher than two.

4 Experimental Setup

Our Delaunay nearest neighbor algorithm (DelaunayNN) was tested in practice against two algorithms. The first was ANN, the kd-tree nearest neighbor implementation from David Mount [16]. The second was our implementation of the full Delaunay hierarchy (FDH) algorithm presented by Birn et al. [3].

Experiments were conducted on a machine with dual 2.66 GHz Quad-core Intel Xeon CPUs, using a total of 4 GB DDR memory. Each core had 2 MB of total cache. The operating system used was SUSE Linux version 11.2, kernel 2.6.31.8-0.1. All source code was compiled using g++ version 4.4.1, with `-O3` enabled.

DelaunayNN was written using C++. It used the Triangle library by Shewchuk [18] to construct the Delaunay triangulation in pre-processing. In our implementation the constant η was set to the value 4, which we determined empirically to be a good value. It should also be noted that the maximum degree of any vertex for all of the tested data sets was 64, which was small enough that the Voronoi preprocessing of points was not needed for any of the experiments (in line 8 and 27 of Algorithm 1, the lower bound $\Omega(1)$ was replaced by 64). FDH was implemented using C++. It used the CGAL library [4] to construct the Delaunay hierarchy in pre-processing. In both cases, exact predicates were used to construct the Delaunay graphs. To keep a fair comparison with ANN, however, both used inexact floating point arithmetic when computing distances for queries. In all experimental cases this had no impact on the solution. For both DelaunayNN and FDH, points were stored along with edges of the graph in order to take advantage of spatial locality in the cache, at the cost of some storage efficiency. In all experiments, the nearest neighbor to the query point was found exactly. ANN used $\epsilon = 0$.

For comparison purposes, point distributions for the experiments were chosen to be the same as those used by Birn et al. [3, 10]. To recap, there are four distributions used:

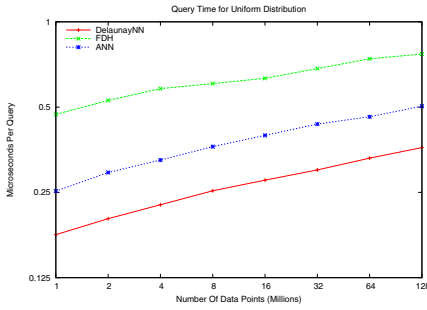
1. Data points chosen uniformly at random from the unit square. Query points chosen uniformly at random from a square 5% larger than the unit square.
2. Data points chosen uniformly at random from the unit circle. Query points chosen uniformly at random from the smallest containing square around the unit circle.
3. Data points chosen with 95% from the unit circle, 5% from the smallest square containing the unit circle. Query points chosen at random from the unit circle.
4. Data points chosen with $x = [-1000, 1000]$ and $y = x^2$. Query points chosen uniformly at random from the rectangle containing the parabola.

We chose to conduct our experiments using large data sets, to better understand the asymptotic behavior of the algorithms. For each experiment, point sets were created ranging in size from one million to 128 million points. 100,000 queries were used in each experiment. To account for randomness in the algorithms and the system, each experiment was run five times (with unique data and query sets for each), and the results were averaged. The next section describes the results and shows graphs of the run time. *Note that all graphs use a base 2 logarithmic scale.* At the end of the paper, we also include tables with the discrete timing results.

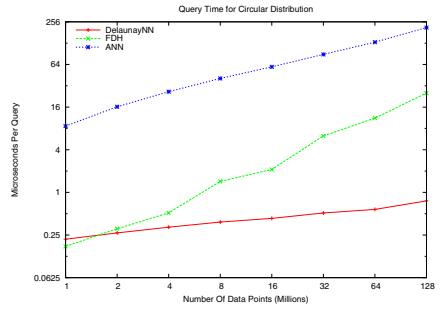
5 Experimental Results

As shown in Figure 4, our algorithm behaves very well in practice on point sets from various distributions. For data sets of sufficient size, the DelaunayNN implementation proves faster than FDH in all cases, and faster than ANN in almost all cases.

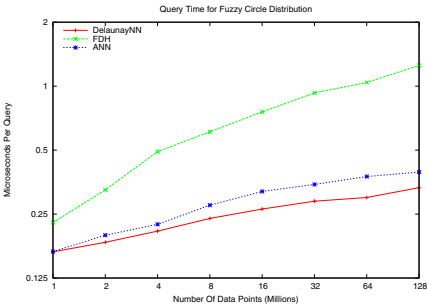
Figure 4(a) shows the results for uniform distribution, which most closely follows the bounded expansion constant considered in the analysis. All implementations performed very well, with low average query times even for very large data sets. Overall, the increase in average query time was significantly less than $\log n$ for all three implementations.



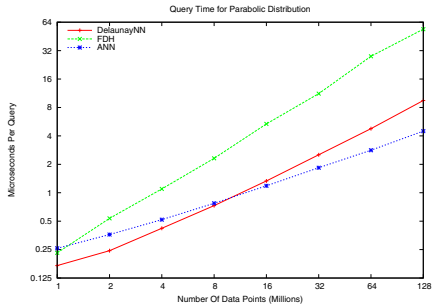
(a) Showing average time per query versus data set size for points taken from uniform distribution.



(b) Showing average time per query versus data set size for points taken from a unit circle. Query points were taken from the smallest square enclosing the circle.



(c) Showing average query time versus data set size for points taken from the unit circle, plus some points chosen uniformly at random from the square containing the circle. Query points taken from the circle.



(d) Showing average time per query versus data set size for points taken from a parabola. Query points were taken from the smallest rectangle enclosing the parabola.

Fig. 4. Average time per query on various distributions

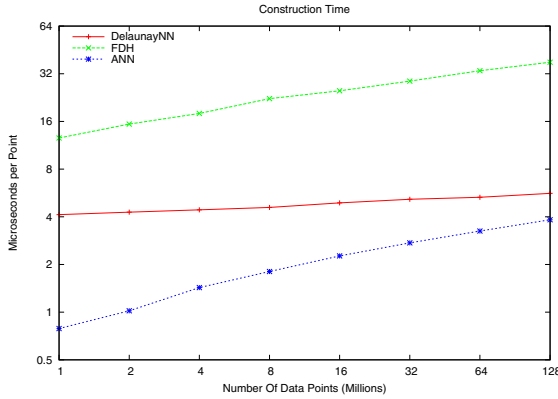


Fig. 5. Showing average time per point to pre-process data sets for queries. Data was taken uniformly at random from the unit square.

Table 1. Query Time for Uniform Dist

Data Size (millions)	ANN (μsecs)	FDH (μsecs)	DelaunayNN (μsecs)
1	0.25270	0.47128	0.17729
2	0.29390	0.52770	0.20186
4	0.32515	0.58120	0.22614
8	0.36260	0.60524	0.25335
16	0.39758	0.63178	0.27608
32	0.43566	0.68395	0.30011
64	0.46234	0.74034	0.33027
128	0.50375	0.76978	0.35981

Table 2. Query Time for Circle Dist

Data Size (millions)	ANN (μsecs)	FDH (μsecs)	DelaunayNN (μsecs)
1	8.63243	0.17260	0.21796
2	16.18790	0.30660	0.26801
4	26.42330	0.51370	0.32227
8	40.67710	1.42964	0.38098
16	59.25260	2.10402	0.42979
32	88.78520	6.26529	0.51216
64	132.12500	11.19470	0.57460
128	212.07100	25.28850	0.75866

Table 3. Query Time for Fuzzy Circle Dist

Data Size (millions)	ANN (μsecs)	FDH (μsecs)	DelaunayNN (μsecs)
1	0.16654	0.22836	0.16604
2	0.19873	0.32517	0.18411
4	0.22376	0.49217	0.20783
8	0.27521	0.61018	0.23853
16	0.31957	0.75705	0.26399
32	0.34461	0.92965	0.28739
64	0.37554	1.04076	0.29913
128	0.39424	1.25171	0.33284

Table 4. Query Time for Parabola Dist

Data Size (millions)	ANN (μsecs)	FDH (μsecs)	DelaunayNN (μsecs)
1	0.25818	0.23096	0.16939
2	0.36093	0.53571	0.24350
4	0.51931	1.09705	0.42040
8	0.77374	2.31555	0.73104
16	1.18471	5.35778	1.33284
32	1.84100	11.21760	2.52317
64	2.82095	27.85140	4.77147
128	4.49590	53.95780	9.48711

In Figure 4(b) we see that for data sets where points are distributed on a circle, the DelaunayNN displays timing results that are very similar to its performance on uniformly distributed data, where both ANN and FDH perform substantially worse than on uniform data. This trend continues in Figure 4(d), with the exception of ANN’s performance, which is again closer to its performance on uniform data.

Table 5. Pre-Processing Time for Uniform Dist

Data Set Size (millions)	ANN (μ secs)	FDH (μ secs)	DelaunayNN (μ secs)
1	0.788120	12.569800	4.126410
2	1.019810	15.401650	4.278760
4	1.428278	17.975875	4.426575
8	1.804425	22.293875	4.581163
16	2.264475	24.980688	4.900006
32	2.736934	28.750000	5.159531
64	3.250344	33.437500	5.312781
128	3.829984	37.812500	5.625000

The one anomalous case we had is documented in Figure 4(c). In this case, ANN had a marked edge in performance for larger point sets over DelaunayNN and FDH. It is also worth noting that for this type of distribution, all implementations had significantly worse scaling than on other distributions.

Figure 5 shows the difference in pre-processing time for the various implementations on uniform data. While ANN maintains a distinct advantage, DelaunayNN scales much better as the data set size increases. It is also clear that using divide and conquer approach allows for Delaunay triangulation with much more reasonable construction times, whereas FDH is forced to use the practically less efficient, incremental construction. Tables 1 to 5 show the data corresponding to the graphs.

6 Conclusions and Future Work

We have presented an algorithm for finding the nearest neighbor for a query in two dimensions that has both an expected run time bound of $\mathcal{O}(\log n)$ and strong experimental performance when compared to existing, state of the art implementations. It remains to be seen if this approach can be applied in a reasonable manner to dimensions higher than two, and if it can be extended to allow for efficient solutions to the k-nearest neighbor problem.

Acknowledgements. We would like to thank Samidh Chatterjee for discussion and suggestions regarding this paper. This research was partially supported by National Science Foundation through CAREER Grant CCF-0643593, Florida State University Committee on Faculty Research Support (COFRS) Summer Award and the Air Force Young Investigator Program.

References

- [1] Arya, S., Mount, D.: Computational geometry: Proximity and location. In: Mehta, D., Sahni, S. (eds.) Handbook of Data Structures and Applications, ch. 3, pp. 63–1, 63–22. CRC Press, Boca Raton (2005)

- [2] Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.: An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM* 45, 891–923 (1998)
- [3] Birn, M., Holtgrewe, M., Sanders, P., Singler, J.: Simple and Fast Nearest Neighbor Search. In: 2010 Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, January 16, pp. 43–54 (2010)
- [4] Boissonnat, J.-D., Devillers, O., Teillaud, M., Yvinec, M.: Triangulations in cgal (extended abstract). In: SCG 2000: Proceedings of the sixteenth annual symposium on Computational geometry, pp. 11–18. ACM, New York (2000)
- [5] Chan, T.M.: Closest-point problems simplified on the ram. In: SODA 2002: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 472–473. Society for Industrial and Applied Mathematics, Philadelphia (2002)
- [6] Chan, T.M.: Manuscript: A minimalist’s implementation of an approximate nearest neighbor algorithm in fixed dimensions (2006)
- [7] Clarkson, K.L.: Nearest-neighbor searching and metric space dimensions. In: Shakhnarovich, G., Darrell, T., Indyk, P. (eds.) Nearest-Neighbor Methods for Learning and Vision: Theory and Practice, pp. 15–59. MIT Press, Cambridge (2006)
- [8] Connor, M., Kumar, P.: Fast construction of k-nearest neighbor graphs for point clouds. *IEEE Transactions on Visualization and Computer Graphics* 99 (PrePrints) (2010)
- [9] de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O.: *Computational Geometry: Algorithms and Applications*, 2nd edn. Springer, Heidelberg (2000)
- [10] Devillers, O.: The Delaunay Hierarchy. *International Journal of Foundations of Computer Science* 13, 163–180 (2002)
- [11] Eppstein, D., Goodrich, M.T., Sun, J.Z.: The skip quadtree: a simple dynamic data structure for multidimensional data. In: Proc. of the twenty-first annual symposium on Computational geometry, pp. 296–305. ACM Press, New York (2005)
- [12] Karger, D.R., Ruhl, M.: Finding nearest neighbors in growth-restricted metrics. In: STOC 2002: Proceedings of the thirty-fourth annual ACM symposium on Theory of computing, pp. 741–750. ACM, New York (2002)
- [13] Kirkpatrick, D.G.: Optimal search in planar subdivisions. *SIAM Journal on Computing* 12(1), 28–35 (1983)
- [14] Kranakis, E., Singh, H., Urrutia, J.: Compass routing on geometric networks. In: Proc. of 11th Canadian Conference on Computational Geometry, pp. 51–54 (1999)
- [15] Milgram, S.: The small world problem. *Psychology Today* 1(1), 60–67 (1967)
- [16] Mount, D.: ANN: Library for Approximate Nearest Neighbor Searching (1998), <http://www.cs.umd.edu/~mount/ANN/>
- [17] Samet, H.: Applications of spatial data structures: Computer graphics, image processing, and GIS. Addison-Wesley Longman Publishing Co., Inc., Boston (1990)
- [18] Shewchuk, J.R.: Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In: Lin, M.C., Manocha, D. (eds.) FCRC-WS 1996 and WACG 1996. LNCS, vol. 1148, pp. 203–222. Springer, Heidelberg (1996); From the First ACM Workshop on Applied Computational Geometry