

THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

A SIMPLE, THREAD-SAFE, APPROXIMATE NEAREST NEIGHBOR
ALGORITHM

By

MICHAEL F. CONNOR

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Fall Semester, 2007

The members of the Committee approve the Thesis of Michael F. Connor defended on November 7, 2007.

Piyush Kumar
Professor Directing Thesis

David Whalley
Committee Member

Xiuwen Liu
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

This thesis is dedicated to my parents.

ACKNOWLEDGEMENTS

I would like to thank my major professor, Dr. Piyush Kumar, for his assistance and guidance in the course of this research. In addition, I would like to thank the members of my committee. Finally I would like to acknowledge support from the faculty of the Florida State University Department of Computer Science.

— Michael Connor

TABLE OF CONTENTS

List of Figures	vi
Abstract	viii
1. INTRODUCTION	1
2. PROBLEM DEFINITIONS	4
3. THE ALGORITHM	8
4. RUNTIME ANALYSIS	14
5. EXPERIMENT	17
5.1 SETUP	17
5.2 RESULTS	18
6. CONCLUSION AND FUTURE WORK	25
APPENDIX	26
A. FLOATING POINT LESS THAN OPERATOR	26
REFERENCES	28
BIOGRAPHICAL SKETCH	30

LIST OF FIGURES

2.1	A 2-Dimensional Nearest Neighbor search. The query point is shown in green, the nearest neighbor in red intersects the ball centered at the query point.	4
2.2	A 2-Dimensional 3-Nearest Neighbor search. The query point is shown in green, the nearest neighbors in red intersect the ball centered at the query point.	5
2.3	A 2-Dimensional Approximate 3-Nearest Neighbor search. The query point is shown in green, the nearest neighbors in red. Note that the inner circle is the correct nearest neighbor ball, and the search does not return the correct answer.	6
2.4	A 2-Dimensional, 1-Nearest Neighbor Graph. Each point is connected to it's nearest neighbor, arrowheads indicate the the direction of association	7
3.1	This figure shows an example of z-ordering in 2 dimensions, as well as the quadtree. (a) shows one subdivision of the space, (b) shows two subdivisions	9
3.2	Pictorial representation of Line 4, Algorithm 1. Since all the points inside the approximate nearest neighbor ball of q have been scanned, we must have found the nearest neighbor.	12
5.1	Query time versus number of points using different number of processors for exact nearest neighbor searches. Input: up to 1 million input points from uniform distribution.	20
5.2	Query time versus number of points using different algorithms for exact nearest neighbor searches. Input: up to 1 million input points from uniform distribution.	20
5.3	Query time versus number of k -nearest neighbors queried using different algorithms for exact nearest neighbor searches. Input: 1 million input points from uniform distribution.	21
5.4	Query time versus approximation factor for $(1 + \epsilon)$ -approximate nearest neighbor search using different algorithms. Input: 1 million input points from uniform distribution.	21

5.5	Query time versus number of dimensions using different algorithms for exact nearest neighbor searches. Input: 1 million input points from uniform distribution.	22
5.6	Query time versus number of points for uniform distribution, with floating point coordinates $\epsilon = 0, k = 1$	22
5.7	Query time versus number of points for gaussian distribution. $\epsilon = 0, k = 1$	23
5.8	Query time versus number of points for uniformly distributed points on a sphere. $\epsilon = 0, k = 1$	23
5.9	Query time versus number of points for clustered gaussian distribution. $\epsilon = 0, k = 1$	24
5.10	Query time versus number of points for points from <i>Sphere+Noise</i> distribution. $\epsilon = 0, k = 1$	24

ABSTRACT

This thesis describes the implementation of a fast, dynamic, approximate, nearest-neighbor search algorithm that works well in fixed dimensions ($d \leq 5$), based on sorting points in Morton (or z-) ordering. This algorithm scales well on multi-core/cpu shared memory systems, and can run on multiple processors simultaneously. The implementation is competitive with the best approximate nearest neighbor searching codes available on the web [1], especially for creating approximate k -nearest neighbor graphs of a point cloud. An extensive C++ library has been built implementing the research presented here. It can be found at: <http://www.compgeom.com/~stann>.

CHAPTER 1

INTRODUCTION

Nearest neighbor search is a fundamental geometric problem important in variety of applications including data mining, machine learning, pattern recognition, computer vision, graphics, statistics, bioinformatics, and data compression [2, 3]. Applications of the nearest neighbor problem are particularly motivated by problems like reconstruction, visualization, and simplification for geometry processing applications [4]. With such a wide ranging impact, it becomes critical to make algorithms for finding nearest neighbors as efficient as possible.

The advent of new multi-core architectures and their anticipated widespread adoption is another motivation to redesign algorithms for the nearest neighbor problem, even with the added difficulty and complexity of developing multi-threaded applications. In addition, as processing power becomes greater, algorithms will become increasingly bound by memory. Cache efficiency will lead to greater gains in performance.

In low dimensions, the seminal work of Arya et.al. [5] was the first to offer linear size space with logarithmic query time for the $(1 + \epsilon)$ -approximate nearest neighbor problem. Their ANN library is the de-facto standard for distribution independent approximate nearest neighbor search problems in lower dimensions and we do not know of any result that beats their library in practice [6]. Their C++ library is very carefully optimized both for memory access and speed, and hence has been the choice of practitioners for many years (in various areas). The two main problems with ANN are that it does not allow queries to be made in parallel, and it does not allow dynamic access to the data structure. Many implementations of approximate nearest neighbor search algorithms based on space filling curves have been reported previously in the literature but none of them were competitive with ANN in low dimensions [7, 8, 9]. Distribution dependent codes like NearPt3 do beat ANN in practice for some distributions important for particular applications but unfortunately can show a

dramatic slow down for types of input that might be important for other applications (like LIDAR data processing, where there are a lot of outliers) [10]. More general approximate nearest neighbor codes (for example ANN in general metric spaces) or solutions that work for higher dimensional spaces are also available online, but these are not competitive with specially crafted low dimensional codes for approximate nearest neighbor search, and hence they were not considered in this research [2].

In this thesis it will be shown that by combining ideas from Chan’s technique [11] and ANN’s algorithm [6], one can obtain a solution which can beat both these algorithms in practice with expected logarithmic query time. This new algorithm is based on sorting points in Morton ordering, is easy to implement, can beat ANN given a high enough number of processors and gives provably correct answers. The preprocessing time of this algorithm is $O(n \log n)$ and query time is expected $O(\log n + \epsilon^{1-d} \log 1/\epsilon)$ for a single approximate nearest neighbor query (where d is the dimension of the pointset and ϵ is the accepted error in the approximate solution). This is slightly better than similar previous results [11, 12]. If m points are available in advance for making queries (batched queries), and $d < p < m$, then the expected query time for a single point reduces to $\frac{1}{p}O(\log n + \epsilon^{1-d} \log 1/\epsilon)$.

The key contributions of this research are:

1. The design and implementation of an approximate nearest neighbor algorithm that is easy to implement in a thread-safe way. This thesis is the first to present extensive empirical results on a multi-core/cpu system.
2. The design of this algorithm is cache efficient, leading to improved timings as processing power is increased.
3. The implementation has provable expected logarithmic query time and is fast in practice compared to the best codes available in its class.
4. While Chan’s algorithm only operated on unsigned integer coordinates, this implementation extends that work to support both signed integers and floating point coordinates.
5. The paper presents the first dynamic version of an approximate nearest neighbor search data structure that is competitive with ANN in practice.

6. The timings for k -nearest neighbor computation are faster than previous codes as k increases. It can also compute k -nearest neighbor graphs faster than ANN in most cases.

The main drawbacks of this approach are:

1. This algorithm is slower than ANN on single processor systems or if at least $O(p)$ queries are not available in batches. Four or more cores are recommended for this implementation.
2. The bound on the running time is randomized. The expected running time bounds are independent of distributions from which the point cloud is sampled.
3. The algorithm is only suited for usage in low dimensions ($d \leq 5$).
4. It is assumed 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 [9, 12].

The C++ library implemented based on this library is available for use at <http://www.compgeom.com/~stann>.

The remainder of the thesis is organized as follows. Chapter 2 describes in detail the different types of nearest neighbor problems. Chapter 3 details the algorithm, and Chapter 4 describes the runtime. Finally, Chapter 5 explains the experimental setup, and gives the results from using this algorithm in practice.

CHAPTER 2

PROBLEM DEFINITIONS

The following problem definitions are used as the basis for the research done in this thesis.

- Nearest Neighbor Search: Given an input point set P composed of an arbitrary number of d -dimensional points, and a d -dimensional point q , return the point $x \in P$ that has the minimum distance to q .

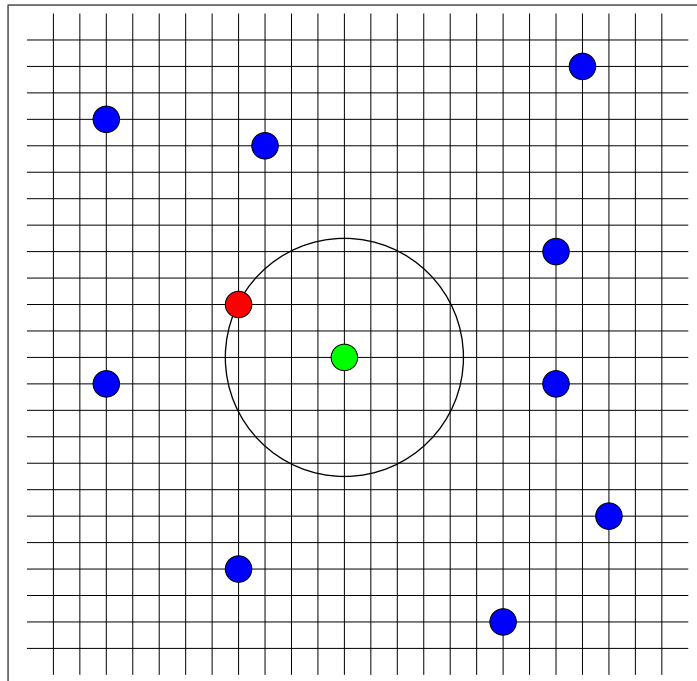


Figure 2.1: A 2-Dimensional Nearest Neighbor search. The query point is shown in green, the nearest neighbor in red intersects the ball centered at the query point.

- K Nearest Neighbor Search: Given an input point set P composed of an arbitrary

number of d -dimensional points, and a d -dimensional point q , return the set of k points $X \subseteq P$ such that the smallest ball containing X and q has the minimum possible radius.

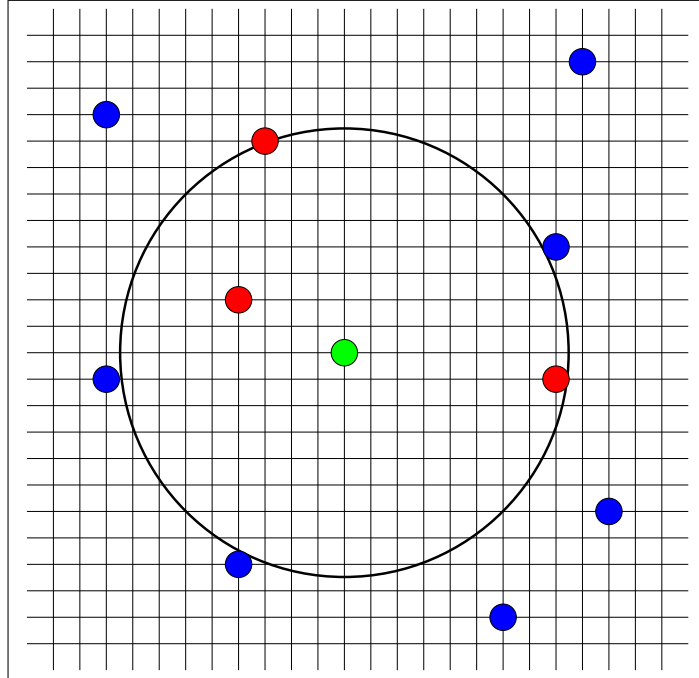


Figure 2.2: A 2-Dimensional 3-Nearest Neighbor search. The query point is shown in green, the nearest neighbors in red intersect the ball centered at the query point.

- Approximate KNN Search: Given an input point set P composed of an arbitrary number of d -dimensional points, a d -dimensional point q , and an error tolerance ϵ return the set of k points $X \subseteq P$ such that the smallest ball containing $X * (1 + \epsilon)$ and q has the minimum possible radius.
- Dynamic Problem: The dynamic version of the above problems allows for points to be arbitrarily added or removed from the input point set as needed. (By default, the input data set is assumed to be static).
- Nearest Neighbor Ball: The nearest neighbor ball of a given k nearest neighbor search is a priority queue containing the k points $p \in P$ that have the least distance to the query point q .

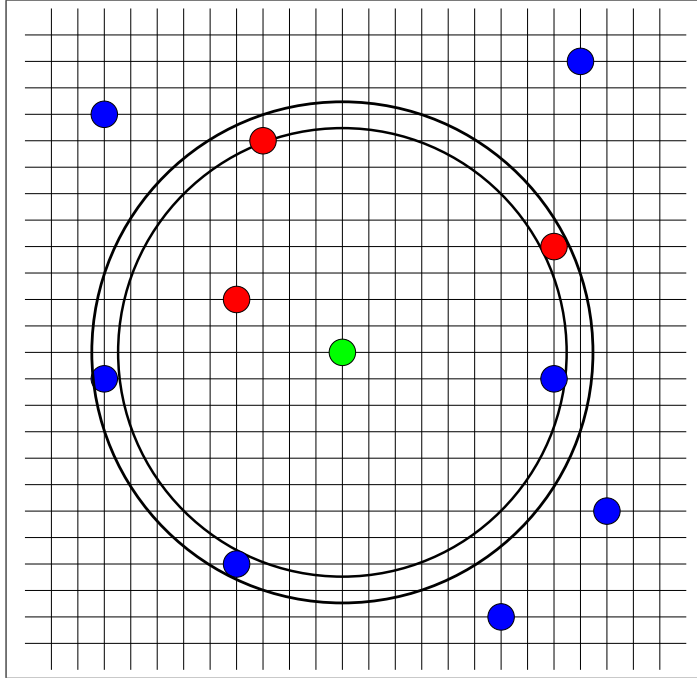


Figure 2.3: A 2-Dimensional Approximate 3-Nearest Neighbor search. The query point is shown in green, the nearest neighbors in red. Note that the inner circle is the correct nearest neighbor ball, and the search does not return the correct answer.

- Parallel Problem: The parallel version of the above problems requires that nearest neighbor queries can be accomplished in parallel, yielding an improvement in query time.
- K Nearest Neighbor Graph: Given an input point set P composed of an arbitrary number of d -dimensional points, for each point $p \in P$, return the set of k points $X \subseteq P$ such that the smallest ball containing X and p has the minimum possible radius. This is equivalent to computing the edges of a graph that connect each point to its k nearest neighbors.

The following definitions are useful in quantifying the performance of a nearest neighbor algorithm.

- Construction Time: The time taken to create the underlying data structures, for a given input point set, that are needed by a nearest neighbor algorithm.

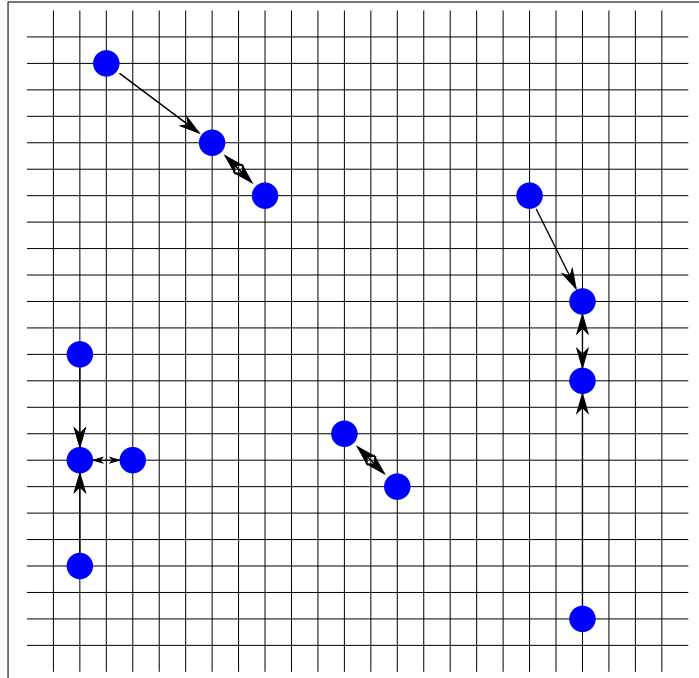


Figure 2.4: A 2-Dimensional, 1-Nearest Neighbor Graph. Each point is connected to it's nearest neighbor, arrowheads indicate the the direction of association

- Query Time: The time taken to return a nearest neighbor result, for a given query point, from an existing nearest neighbor data structure.

CHAPTER 3

THE ALGORITHM

Notation: There are several terms and functions used for clarity.

- p_a is the a – *th* point in the sorted Morton ordering of the point set.
- $d(p_a, p_b)$ refers to the Euclidean distance from the point p_a to the point p_b .
- $D_\infty(p_a, r)$ refers to a box, centered at the point p_a , with side length $2r$.
- $S(p_a, r)$ refers to a sphere, centered at point p_a with radius r .
- $p_a \leq p_b$ iff p_a is ranked lower or equal to p_b in Morton ordering.
- q^s refers to a point q which has a value s added to each coordinate.
- Given a point $p = \{p_0, p_1, \dots, p_{d-1}\} \in P$ where each coordinate p_j is a b -bit number $p_{jb-1}, p_{jb-2}, \dots, p_{j0}$ its z-value

$$z(p) = p_{0b-1}p_{1b-1} \dots p_{d-1b-1} \dots p_{0b-2}p_{1b-2} \dots p_{d-1b-2} \dots p_{00}p_{10} \dots p_{d-10}$$

- The algorithm has an implicit quadtree structure, defined by the Morton ordering of the points [11]. Morton ordering can be achieved by recursively subdividing the space into 2^d boxes until each box contains only one point, then recursively sorting those boxes. See Figure 3.1. Boxes that do not contain points after this subdivision are simply skipped over.
- $B(p_a, p_b)$ refers to the smallest quadtree box that contains the points p_a and p_b .

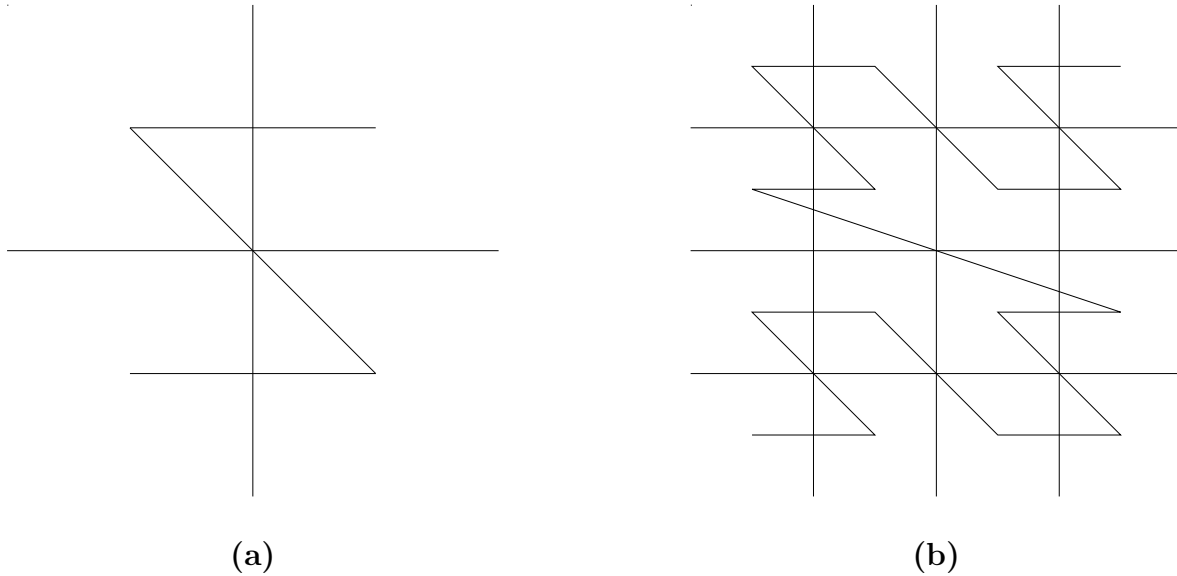


Figure 3.1: This figure shows an example of z-ordering in 2 dimensions, as well as the quadtree. (a) shows one subdivision of the space, (b) shows two subdivisions

Primitives: There are a few primitives required to implement the algorithm. The comparison of two points in Morton ordering ($p \leq q$) can be implemented to work in $O(d)$ time for two d -dimensional integral points using either the **BSR** (Bit Search Reverse) instruction on x86 architectures or by using Chan’s XOR-trick [11]. The **BSR** instruction used to consume several dozen cycles on a Pentium processor, but became a 2-cycle instruction on Pentium II and newer CPUs [13]. Both versions were implemented, and it was found that the assembly code that uses **BSR** runs slightly slower than Chan’s trick due mostly to the fact that **BSR** still takes 2 cycles to run on most modern CPUs (two calls to **BSR** are required). Floating point coordinates can be compared by isolating the exponents using **frexp** and comparing them, and **XORing** the bits of the mantissa if needed. Signed coordinates can be compared by simply noting that all negative coordinates come before all positive coordinates, and when comparing two negative coordinates the result is the opposite of the positive case. An explanation of the implementation of the z-order comparison for floating point numbers is given in Appendix A

The smallest quad-tree box that contains two points can be computed using the following argument: Let

$$i = 1 + \max_{j=1\dots d} \text{BSR}(x_j \text{ xor } y_j).$$

The the m -th coordinate interval of the quad-tree box $B(p, q)$

is then given by $[\lfloor x_m/2^i \rfloor 2^i, \lfloor (x_m/2^i + 1) \rfloor 2^i)$. Thus $B(p, q)$ can be computed in $O(d)$ cycles using shifts and BSR. Using BSR for computation of $B(p, q)$ turned out to be slightly faster than using `frexp` on the testing architecture (although `frexp` is more portable way to compute most significant bit of an integer compared to BSR; both of these methods run in constant time on modern architectures).

Preprocessing: The preprocessing algorithm is the same as [9]. Given a point cloud $P = \{p_1, p_2, \dots, p_n\} \subset \mathbb{N}^d$ and a random number s , sort P^s in z-order. This step runs in $O(dn \log n)$ time.

Query Algorithm: The query algorithm is presented in Algorithm 1. The query is done in two phases. The first phase invokes a binary search and the second invokes a modified binary search. The first phase locates the query point q in P using a simple binary search. Line 3 of the algorithm linearly scans points around this location to reduce the radius of the approximate nearest neighbor ball. c is a constant depending on the architecture (for all experiments it was set to 5). Also, when $i - c < 1$ or $i + c > n$, $i - c$ is set to 1 or $i + c$ to n . Line 4 checks if the scan in the previous line yielded a nearest neighbor (see Figure 3.2).

In case the nearest neighbor has not been found yet, the second phase of the search is invoked (Line 5), which happens to be a modified version of the query algorithm presented in [9] (Line 7-22). Line 8-12 do a linear scan to find the nearest neighbor in case the number of points is smaller than a pre-specified constant ν . This was set to 5 in all experiments. In line 13, the nearest neighbor radius is updated by computing the distance from the query point to p_m , the mid-point of the modified binary search. Line 14 checks if the approximate nearest neighbor ball intersects with the quad-tree box $B(p_l, p_h)$. Line 15-21 completes the binary search by recursing on one or both subsequences of the Morton ordering. The modifications only add an extra $O(\log n)$ time to the total search. The total time taken by the algorithm, assuming the dimension is a constant, on a single cpu system is expected $O(\log n + \epsilon^{1-d} \log 1/\epsilon)$. It is easy to extend Algorithm 1 to support k -nearest neighbor queries using a priority queue. For k -nearest neighbor computation, the expected running time of this implementation is $O(\log k(\log n + \epsilon^{1-d} \log 1/\epsilon) + k)$.

Algorithm 1 Query algorithm

Require: P is sorted in z-order. q and P are shifted. $\epsilon \geq 0$.

```
1: procedure QUERY
2:    $i \leftarrow \text{BinarySearch}(q, P)$ 
3:    $r \leftarrow nn(q, p_{i-c} \dots p_{i+c})$ 
4:   if  $(q^{-\lceil r \rceil} \geq p_{i-c}$  and  $q^{\lceil r \rceil} \leq p_{i+c})$  then return  $r$ 
5:   return CSearch(1,  $n$ )
6: end procedure
7: procedure CSEARCH( $l, h$ )
8:   if  $(h - l) \leq \nu$  then
9:      $r \leftarrow \min(r, nn(q, p_l \dots p_h))$ 
10:    return
11:   end if
12:    $m \leftarrow (h + l)/2$ 
13:    $r \leftarrow \min(r, d(q, p_m))$ 
14:   if  $d(q, B(p_l, p_h)) \geq r/(1 + \epsilon)$  then return
15:   if  $q \leq p_m$  then
16:     CSEARCH( $l, m - 1$ )
17:     if  $q^{\lceil r \rceil} \geq p_m$  then CSEARCH( $m + 1, h$ )
18:   else
19:     CSEARCH( $m + 1, h$ )
20:     if  $q^{-\lceil r \rceil} \geq p_m$  then CSEARCH( $l, m - 1$ )
21:   end if
22: end procedure
```

The first phase of the search helps the second phase in many ways. It brings the binary search path into the cache. It helps eliminate some of the calls to the second phase (line 5) and it reduces the size of the approximate nearest neighbor ball, which in turn reduces the number of recursive calls to the second phase (lines 14,17,20). Note that for computing the k -nearest neighbor graph, the binary search in the 2nd line of the query algorithm can be eliminated by just a pointer lookup. This speeds up the computation of the k -nearest neighbor graph slightly.

The following list categorizes the static and dynamic approximate nearest neighbor algorithms that were evaluated with during the course of this study. There are three static algorithms, which includes the one presented above. There are two dynamic algorithms, one of which is the dynamic version of the above algorithm using a data structure called Packed memory arrays (PMA) [14].

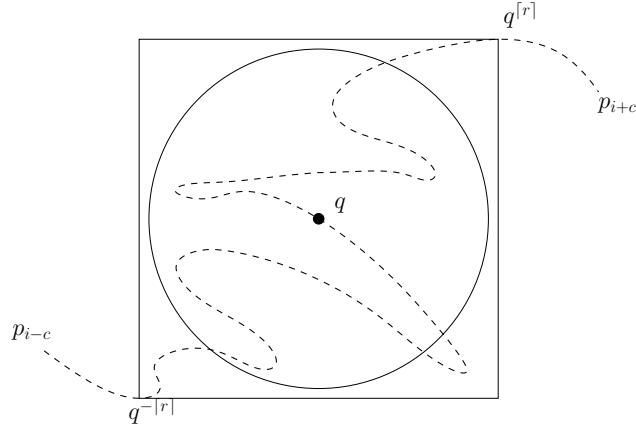


Figure 3.2: Pictorial representation of Line 4, Algorithm 1. Since all the points inside the approximate nearest neighbor ball of q have been scanned, we must have found the nearest neighbor.

- **Chan’s Static Algorithm [9]:** We have incorporated an implementation of Chan’s static algorithm in our code base without any major changes. We have added the capability to do k -nearest neighbor searches to his original implementation and made his code thread-safe.
- **ANN [5]:** This implementation is by Arya and Mount and is essentially an optimized kd-tree implementation. We did not compare our implementation with the BBD-tree implementation in ANN because it seemed to be slower than the kd-tree implementation for all the data sets we tested. ANN does not allow doing either parallel batch searches, insertions or deletions.
- **Chan’s Dynamic Algorithm:** We implemented the algorithm given by Chan in [11] without any major changes. The balanced binary tree implementation uses a red-black tree from the standard template library (STL) of C++ (map). This implementation uses $d + 1$ -shifted copies of the point cloud sorted in Morton ordering to locate the approximate nearest neighbor. We do not include this implementation (or its modifications that we experimented with) for comparison purposes because it is slower than the dynamic version of Algorithm 1 by at least a factor of 2 or more, even for a constant factor approximation.

- **Algorithm 1 + PMA:** This version of our algorithm uses the packed memory array data structure to make the algorithm dynamic. The packed memory array was chosen because of its ease of implementation and its cache friendliness for range searches. PMA is a data structure that maintains a dynamic set of n elements in sorted order in a $\Theta(n)$ -sized array (our implementation uses a C++ `vector`). The idea is to intersperse $\Theta(n)$ empty spaces or gaps among the elements so that only a small number of elements need to be shifted around on an insert or delete. Because the elements are stored physically in sorted order in memory or on disk, the PMA can be used to support binary search (or its modifications) very easily. The PMA maintains locality of reference at all granularities and consequently supports extremely efficient sequential scans/range queries of the elements [14]. Insertions and deletion time for PMAs are $O(\log^2 n)$, both amortized. Binary search still takes $O(\log n)$ time. To scan L elements after a given element p costs $\Theta(1 + L/B)$ memory transfers where B is the cache line length of memory (or the block size of the disk). An advantage of using PMA on multi-core machines is that it enhances the memory bandwidth available to each core because of locality of reference.

CHAPTER 4

RUNTIME ANALYSIS

This proof closely follows that of Chan [9]. The main difference being that it does not rely on conditional expectations, and the resulting deviation results in slightly better query timings. Given a d -dimensional query point q whose coordinates are chosen at random, and a point set P , let $p \in P$ be the nearest neighbor to q . Let $\epsilon \geq 0$. p_* is an ϵ -approximate nearest neighbor of q if $D_\infty(q, p_*) \leq (1 + \epsilon)D_\infty(q, p)$. Let i be such that $2^{i-1} \leq d(p_*, q) \leq 2^i$. Assume, for some fixed value of c , the following condition, Φ_c , holds:

$D_\infty(q, 2^i)$ is contained in a quadtree box with side length 2^{i+c}

For any value of c , Φ_c will be false if any coordinate of q , modulo 2^{i+c} , lies within $\pm 2^i$ of a quadtree box edge. Since q is chosen at random, the probability of Φ_c not occurring is $P(\overline{\Phi_c}) = \frac{2^i}{2^{i+c}}$, and therefore the probability of Φ_c occurring and being the smallest value for c is:

$$P(\Phi_c \cap \overline{\Phi_{c-1}} \cap \dots \cap \overline{\Phi_1}) \leq P(\overline{\Phi_{c-1}})$$

Note that instead of assuming q is randomly chosen, a constant random shift could be added to each point in the data set as well as the query point.

The expected value for c can then be expressed as:

$$\begin{aligned} E(c) &= d \sum_{c=1}^{\infty} P(c) \\ &\leq d \sum_{c=1}^{\infty} (c-1) \frac{2^i}{2^{i+c-1}} \\ &\leq d \sum_{c=1}^{\infty} \frac{c-1}{2^{c-1}} \\ &= O(d) \end{aligned}$$

Lemma 4.0.1. *After the initial binary search, the initial nearest neighbor ball can be size at most $\sigma_0 2^i$, where $\sigma_0 = O(\sqrt{d}2^d)$.*

Proof. Say $p_k \leq q \leq p_{k+1}$. Either $p_* \leq p_k$ or $p_{k+1} \leq p_*$ and whichever point satisfies that condition must be in $B(p_*, q)$, which, from Φ_c , has a side length at most 2^{i+c} . Therefore, after line 3, the radius of the ball $r_0 \leq \sqrt{d}2^{i+c}$.

$$\begin{aligned} \sigma_0 2^i = r_0 &\Rightarrow \sigma_0 2^i \leq \sqrt{d}2^{i+c} \\ &\Rightarrow \sigma_0 \leq \frac{\sqrt{d}2^{i+c}}{2^i} \\ &\Rightarrow \sigma_0 \leq \sqrt{d}2^c \\ &\Rightarrow \sigma_0 = O(\sqrt{d}2^d) \end{aligned}$$

□

Lemma 4.0.2. *All quadtree boxes visited have side lengths at least $\Omega(\epsilon/\sqrt{d})2^i$.*

Proof. Given a visited quadtree box $B = B(p_a, p_b)$ with side length l , we have the following inequalities:

1. $d(p_m, q) - \sqrt{dl} \leq d(q, B)$

Proof: The point p_m lies within the box B , and can lie at most \sqrt{dl} farther than the closest point on the edge of B to q .

2. $d(p_m, q) - \sqrt{dl} \leq d(q, B) \leq \frac{r}{1+\epsilon} \leq \frac{d(p_m, q)}{1+\epsilon}$

Proof: This follows from the if condition in line 14 of the algorithm

3. It then follows that:

$$\begin{aligned} d(p_m, q) - \sqrt{dl} \leq \frac{d(p_m, q)}{1+\epsilon} &\Rightarrow -\sqrt{dl} \leq \frac{d(p_m, q)}{1+\epsilon} - d(p_m, q) \\ &\Rightarrow l \geq \frac{\epsilon d(p_m, q)}{\sqrt{d}(1+\epsilon)} \end{aligned}$$

4. $d(p_m, q) \geq d(p_*, q) \Rightarrow l = \Omega\left(\frac{\epsilon}{\sqrt{d}}\right)2^i$

Proof: By the definition of i , above.

□

Let ϵ_k be an arbitrary error term. Let $\phi_k = 2^k$ such that $2^{k-1} \leq \epsilon_k \leq 2^k$. Given $S(q, \phi_{k+1}2^i)$, the number of boxes, needed to find $S(q, \phi_k2^i)$ is

$$O\left(\frac{\text{volume of } S(q, \phi_{k+1}2^i) - \text{volume of } S(q, \phi_k2^i)}{\text{volume of the smallest quadtree box for } \epsilon_k}\right)$$

Since the quadtree boxes are laid out uniformly on a grid, and the size of the smallest quadtree box visited for a given error term is directly related to the size of the ball for that error term, this gives a bound on the number of boxes needed to refine the nearest neighbor ball, N_k . The bound on the summation are given by Lemma 4.0.1 and the error term ϵ .

$$\begin{aligned} N_k &\leq \sum_{k=-\log \frac{1}{\epsilon}}^d \frac{\eta_d (2^i + 2^{k+i+1})^d - \eta_d (2^i + 2^{k+i})^d}{\left(\frac{2^{k+i}}{\sqrt{d}}\right)^d} \\ &\leq \sqrt{d}^d \eta_d \sum_{k=-\log \frac{1}{\epsilon}}^d \left[\left(\frac{1}{2^k} + 2\right)^d - \left(\frac{1}{2^k} + 1\right)^d \right] \\ &\leq \sqrt{d}^d \eta_d \left(d + \log \frac{1}{\epsilon}\right) \left[\left(\frac{1}{\epsilon} + 2\right)^d - \left(\frac{1}{\epsilon} + 1\right)^d \right] \\ &= O\left(\sqrt{d}^d \eta_d \left(d + \log \frac{1}{\epsilon}\right) \left(\frac{1}{\epsilon}\right)^{d-1}\right) \end{aligned}$$

Therefore, after adding in the term for the initial binary search, the total runtime of the algorithm is

$$dO\left(\log n + \left(\sqrt{d}^d \eta_d \left(d + \log \frac{1}{\epsilon}\right) \left(\frac{1}{\epsilon}\right)^{d-1}\right)\right)$$

Theorem 4.0.3. *If the dimension is considered a constant, the expected query time of our algorithm is $O(\log n + \epsilon^{1-d} \log 1/\epsilon)$.*

CHAPTER 5

EXPERIMENT

5.1 SETUP

To demonstrate the practicality of this algorithm, experiments were performed on a number of different data sizes and with point sets sampled from a number of different distributions. The system used for the experiments is a 8-CPU 2.6Ghz AMD Opteron 885 system, each CPU has dual-core capability with 64GB of total DDR memory. Each core has 1MB of L2 cache and 64kb of L1 cache. Redhat Linux with kernel 2.6.9-34 was running on the system. The compiler used gcc version 3.4.5 for compilation of all code (with `-O3`). ANN was compiled using the defaults, and run with the standard search options.

For generation of random point clouds, coordinates were first generated in doubles and then scaled to a fixed integer grid of size $(0, 2^{24}]^d$. The distributions that were tested are listed below:

- *Uniform*: Coordinates are picked uniformly between $(0, 1)$.
- *Gaussian*: Points are picked from a gaussian distribution with center $(0, 0)$ and variance 1 in each dimension. They are then translated to the center of the grid and scaled.
- *Clustered Gaussian*: Same as gaussian except there were 10 random centers picked around which gaussians were generated.
- *Spherical*: Points were generated on the surface of a unit sphere and then translated and scaled into the grid.
- *Spherical + Noise*: Same as the above distribution except uniformly distributed noise was added to the points. Out of the many distributions experimented with during the course of this study, this one seemed to be the best case for ANN.

5.2 RESULTS

Table 5.1: Computation timings for $k = 10$ -nearest neighbor graphs. All timings are in seconds.

Dataset	Size	Total Time				Construction Time			
		ANN	Static Chan	Alg. 1	Alg. 1 +PMA	ANN	Static Chan	Alg. 1	Alg. 1 +PMA
Screw	27152	.10	.48	.10	.13	.02	.00	.00	.02
Dinosaur	56194	.24	.24	.19	.24	.08	.02	.02	.05
Ball	137602	.56	.51	.41	.52	.21	.05	.04	.12
Isis	187644	.97	.76	.67	.81	.33	.10	.06	.20
Blade	861240	3.34	3.26	2.8	3.44	1.15	.34	.24	.75
Awakening	2057930	7.35	6.66	5.40	7.15	3.50	.90	.63	2.02
David	3614098	15.75	12.76	11.09	13.94	5.54	1.75	1.13	3.57
Night	11050083	41.56	36.67	29.70	37.99	17.07	4.86	3.41	10.43

Data and query from same distribution: For the first set of experiments, data and query points came from the same distributions. For all experiments, the experiment used 5,000 queries and the complete experiment was run 5 times. The results presented are an average of 5 runs for all experiments. Figure 5.1 shows the average time to query the data structure as the number of processors is varied from 1 to 16. It also shows ANN’s query time with 1 processor. The query time is presented in microseconds for 1 query (averaged over $5k$ queries). This figure shows that this implementation is competitive with ANN on modern dual processor systems and can beat ANN given enough number of processors. For the rest of the experiments, all algorithms except ANN use 16 processors.

Figure 5.2 shows time to query various data structures as the number of points is increased. This figure shows that the penalty for making the data structure dynamic using PMA is not too much, as far as searches are concerned (a small constant factor overhead).

Figure 5.3 shows time to query various data structures as the number of k -nearest neighbors queried increases. It seems that for large k , the modified algorithm behaves better than ANN (and the dynamic version runs with a small constant factor overhead).

Figure 5.4 shows time to query various data structures as ϵ is varied from 0 to 1. This graph shows that the algorithm is not very sensitive to ϵ .

Figure 5.5 shows the query times for various algorithms as the dimension varies between 2

to 5. It clearly shows that ANN is the method of choice as the dimension increases. Table 5.1 shows the time taken to create k -nearest neighbor graphs for different surface reconstruction data sets.

Figure 5.6 shows the query time of the out of place version of the algorithm for points with floating point coordinates. The floating point less than comparator is more computationally intensive than the integer version, which can be seen in the graph by the increased query time. The 4 processor case is still faster than ANN, and the 16 processor case is the same as the 8 and 16 processor cases for integer points.

Data and query from different distribution: The second set of experiments was run on data generated using different distributions with queries from uniform distributions. Figure 5.7,5.8,5.9 and 5.10 show the query timings of the different algorithms for normal distribution, points distributed uniformly on the surface of a sphere, points from clustered gaussian distribution and points on surface of the sphere with noise respectively. Points on the surface of a sphere seems to be the worst case for query times in the distributions tested. Points from *Spherical+Noise* distribution turned out to be the best case for ANN. ANN is very fast on this distribution and beats us even with the use of 16 processors.

Construction time: Table 1 lists the construction and preprocessing times for the different algorithms. The main time taken in preprocessing is the sorting step. As is evident from the results, construction time is faster than ANN. The Static Chan implementation uses the `stl::sort` algorithm. The static implemented algorithm (Alg. 1) uses a parallel version of `stl::sort`. The dynamic version has the slowest construction time due to setting up the packed memory array, although as can be seen, it is still faster than constructing ANN’s kd-tree.

Memory requirements: Algorithm 1 can be implemented *in-place* (without requiring any extra space, except the input array stored in z-order). To maintain a similarity with ANN, an out-of-place version was also constructed. Both implementations of Algorithm 1, and its dynamic version take $\Theta(nd)$ space.

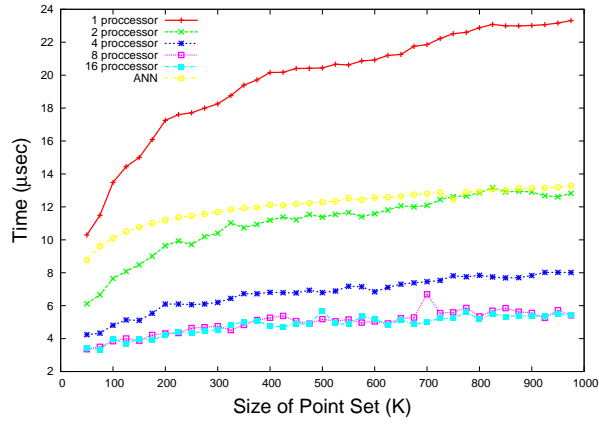


Figure 5.1: Query time versus number of points using different number of processors for exact nearest neighbor searches. Input: up to 1 million input points from uniform distribution.

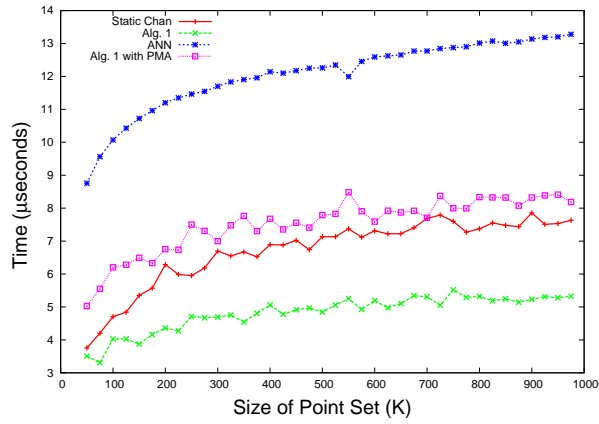


Figure 5.2: Query time versus number of points using different algorithms for exact nearest neighbor searches. Input: up to 1 million input points from uniform distribution.

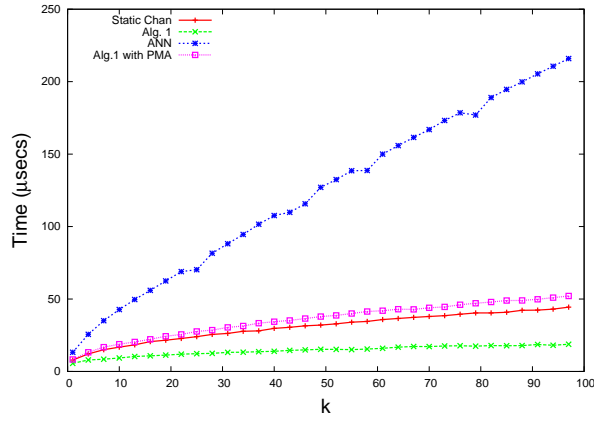


Figure 5.3: Query time versus number of k -nearest neighbors queried using different algorithms for exact nearest neighbor searches. Input: 1 million input points from uniform distribution.

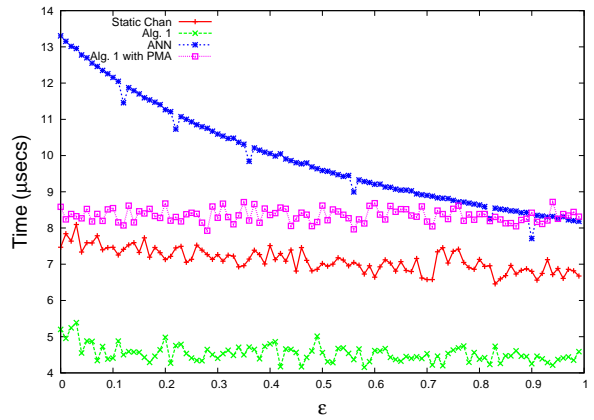


Figure 5.4: Query time versus approximation factor for $(1+\epsilon)$ -approximate nearest neighbor search using different algorithms. Input: 1 million input points from uniform distribution.

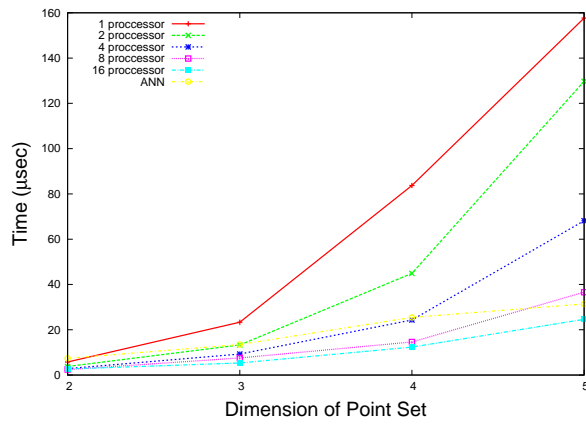


Figure 5.5: Query time versus number of dimensions using different algorithms for exact nearest neighbor searches. Input: 1 million input points from uniform distribution.

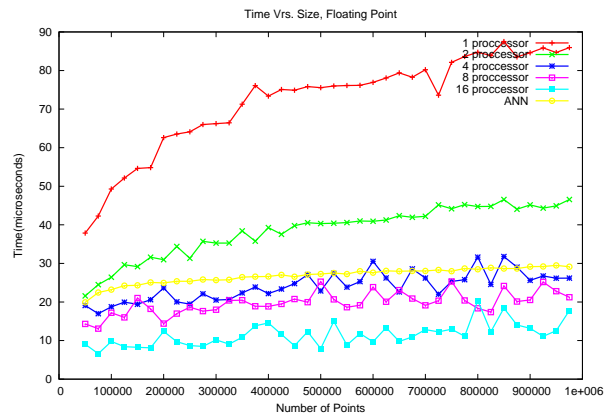


Figure 5.6: Query time versus number of points for uniform distribution, with floating point coordinates $\epsilon = 0, k = 1$

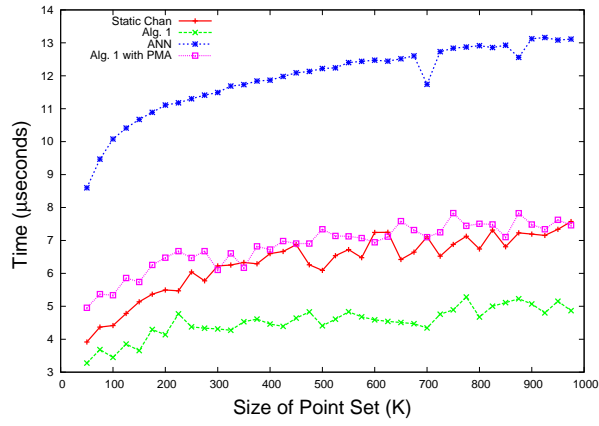


Figure 5.7: Query time versus number of points for gaussian distribution. $\epsilon = 0, k = 1$

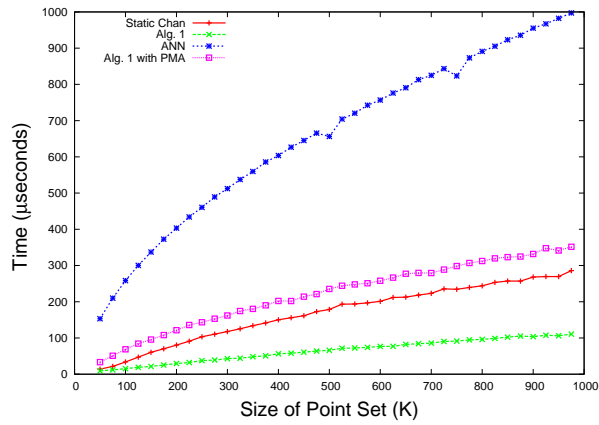


Figure 5.8: Query time versus number of points for uniformly distributed points on a sphere. $\epsilon = 0, k = 1$

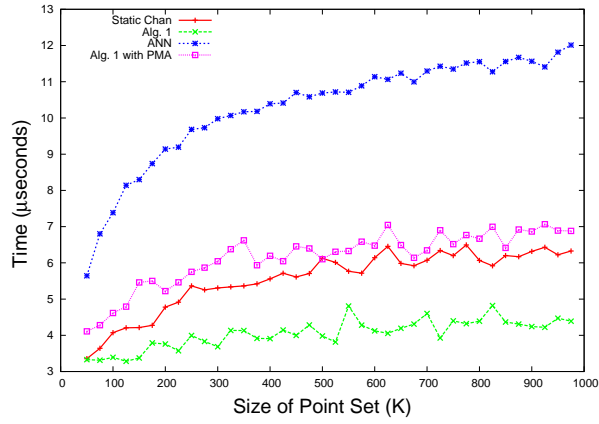


Figure 5.9: Query time versus number of points for clustered gaussian distribution. $\epsilon = 0, k = 1$

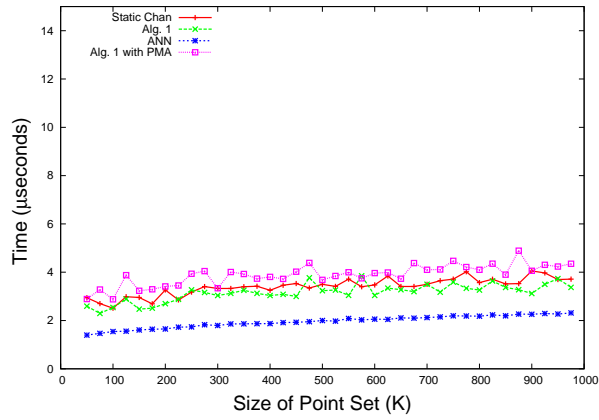


Figure 5.10: Query time versus number of points for points from *Sphere+Noise* distribution. $\epsilon = 0, k = 1$

CHAPTER 6

CONCLUSION AND FUTURE WORK

This thesis has presented an empirical analysis of a simple approximate nearest neighbor algorithm using Morton ordering. The method was designed to scale well on multi-core systems, be cache-friendly and have competitive pre-processing and query times with kd-trees in small dimension.

There are many problems that are motivated by this study that deserve future investigation. How much does the use of Hilbert curves or other clustering methods help in increasing the efficiency of queries? Does it actually help to compute and store the interleaved bits of the Morton-ordering along with the points in the data structure? Does the use of instructions not generally used by compilers (SSE), help us in speeding up queries? Can faster point-location algorithms help in practice? Hopefully, these questions can be studied in future research.

APPENDIX A

FLOATING POINT LESS THAN OPERATOR

This section is an example of code used to compute the Morton ordering of non-integer points. In this code, the points being compared use a double to store the coordinate, and a long long has been determined to be the same byte length.

Morton order is traditionally achieved by interleaving the bits of the coordinates of integer points. Two points are ordered by this interleaved value. This process can be sped up by noting that when two points are ordered, only one bit is actually relevant. Out of each pair of coordinates in points a and b , the relevant bit for that pair is the first different bit between them. Out of all pairs, the pair with highest order differing bit determine the order of the points. This leads to the following function:

```
bool lt_func(const Point &p, const Point &q)
    int j,x,y;
    unsigned int k;
    for (j=k=x=0;k < DIM;++k)
        y = xormsb(p[k], q[k]);
        if (x < y)
            j=k;
            x=y;
    return (p[j] < q[j]);
```

For floating point coordinates, the trick is in determining the order of the first differing bit. In C++, two variables of type double cannot be XORed. Since we only want the most significant bit, this problem can be worked around. First, the exponents of both coordinates are compared using the `frexp` function. If one exponent is greater, than obviously that coordinate has the most significant bit. If they are equal, than the mantissas must be examined. In order to find the most significant bit in the mantissas, they are converted to an equivalent size integer type using a union, and then XORed. Once this is done, the value

is converted back to a double. For each of these cases, the exponent value of the coordinate is returned to the previous function for comparison.

```

union DOUBLELONGLONG
  double d;
  long long l;

//This function computes the most significant
//bit of two doubles, after they have been
//XORed.
int xormsb(double p, double q)
  int xp, xq, y;
          //Calculate the exponent of each double
  mp.d = frexp(p, &xp);
  mq.d = frexp(q, &xq);
          //If one exponent is greater, than that
          //exponent is the power of the msb
  if(xp < xq)
    {y = xq;}
          //If the exponents and mantissa are equal,
          //the power of the msb is 0
  else if(xp == xq)
    if(mp.l == mq.l)
      y = 0;
          //Otherwise, XOR the mantissa, and calculate the
          //new exponent
    else
      (mp.l) = ((mp.l) ^ (mq.l)) | (lzero.l);
      mp.d = frexp(mp.d - .5, &xq);
      y = xp + xq;
  else
    y = xp;
  return y;

```

REFERENCES

- [1] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280, 1993. ([document](#))
- [2] Kenneth L. Clarkson. Nearest-neighbor searching and metric space dimensions. In Gregory Shakhnarovich, Trevor Darrell, and Piotr Indyk, editors, *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, pages 15–59. MIT Press, 2006. [1](#)
- [3] S. Arya and D. Mount. Computational geometry: Proximity and location. In Dinesh Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 63, pages 63–1, 63–22. CRC Press, 2005. [1](#)
- [4] Renato Pajarola. Stream-processing points. In *Proceedings IEEE Visualization, 2005, Online*. Computer Society Press, 2005. [1](#)
- [5] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45:891–923, 1998. [1](#), [3](#)
- [6] D. Mount. [ANN: Library for Approximate Nearest Neighbor Searching](#), 1998. <http://www.cs.umd.edu/~mount/ANN/>. [1](#)
- [7] S. Liao, M. Lopez, and S. Leutenegger. High dimensional similarity search with space filling curves. In *17th International Conference on Data Engineering*, pages 615–622, 2001. [1](#)
- [8] John Shepherd, Xiaoming Zhu, and Nimrod Megiddo. A fast indexing method for multidimensional nearest-neighbor search. In *Proceedings of the SPIE Conference on Storage and Retrieval for Image and Video Databases VI*, volume 3656, San Jose, California, January 1999. [1](#)
- [9] Timothy M. Chan. Manuscript: A minimalist’s implementation of an approximate nearest neighbor algorithm in fixed dimensions, 2006. [1](#), [4](#), [3](#), [3](#), [3](#), [4](#)
- [10] W. R. Franklin. [NearPt3: Nearest Point Query on 184M Points in E3 with a Uniform Grid](#). In *In Proceedings of the 17th Canadian Conference on Computational Geometry (CCCG’05)*, pages 239–242, 2005. [1](#)

- [11] Timothy M. Chan. Closest-point problems simplified on the ram. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 472–473, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. 1, 3, 3, 3
- [12] David Eppstein, Michael T. Goodrich, and Jonathan Z. Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In *SCG '05: Proceedings of the twenty-first annual symposium on Computational geometry*, pages 296–305, New York, NY, USA, 2005. ACM Press. 1, 4
- [13] A. Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel and amd cpu's, 2007. http://www.agner.org/optimize/instruction_tables.pdf. 3
- [14] Michael A. Bender and Haodong Hu. An adaptive packed-memory array. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 20–29, New York, NY, USA, 2006. ACM Press. 3

BIOGRAPHICAL SKETCH

Michael F. Connor

Michael F. Connor was born in Urbana, Illinois in 1980. He has lived in many areas of the United States, including New Hampshire, California, Texas and Florida. He received his Bachelors degree in Computer Science from Florida State University in the Fall of 2005. Thanks in large part to his adviser, Dr. Piyush Kumar, his Masters degree in Computer Science was awarded from FSU in the Fall of 2007.

Michael currently lives in Tallahassee, FL.