# A study of data structures for orthogonal range and nearest neighbor queries in high dimensional spaces


**CSE 523/524 Master's Project**
**Department of Computer Science**
**State University of New York at Stony Brook**
**Stony Brook, NY 11790-4400**

Michael Murphy

Dr. Steve Skiena

# Contents

# 1 Introduction

Given a point set, $S = \{p_1, p_2, ...p_n\}$ in a d-dimensional space, there are two similar searching problems that can arise. The first problem, *range searching*, involves finding a subset of $S$ whose members fall within a given portion of the search space. For simplicity, we restrict our attention to orthogonal rectangles. The second problem, *nearest neighbor searching* entails finding the point $p_k$ in $S$ to a given point $q$, not necessarily in $S$, under a given metric. This problem can usually be extended to finding the $j$-closest neighbors in $S$ to $q$ with a minimal amount of effort in implementation.

In one dimension, these problems can be easily solved in logarithmic time using traditional methods, after an O(n lg n) preprocessing step. We would like to extend this notion to higher dimensions, however, the generalization is not straightforward. The field of Computational Geometry has developed many beautiful techniques for solving these problems, such as the Voronoi Diagram. In low dimensional settings, these techniques are quite efficient. However, as the dimension increases, the preprocessing and storage needed often increase exponentially thus rendering these methods impractical.

As an alternative, we have studied heuristic tree based searching algorithms, that require little storage and roughly O(n lg n) preprocessing, in hopes of attaining sublinear search times.

# 2 Algorithms Studied

We have identified and implemented several candidate algorithms. We briefly summarize them below.

## 2.1 Naive

The *Naive* algorithm is the most straightforward searching algorithm imaginable. For range searches, it simply passes through $S$, keeping those points which fall inside the rectangular query. For nearest neighbor queries, it computes the distances of each point to the query point $q$ and selects the $j$ smallest distances. This was implemented as a benchmark for timing operations and as a debugging tool.

## 2.2 Naive $k-d$ Tree

The Naive $k-d$ Tree is the original multi-dimensional tree defined by Bentley [1].

## 2.3 Median $k-d$ Tree

The Median $k-d$ Tree, [3] refines the $k-d$ tree to employ median cuts, a more sophisticated choice of discriminator, and bucketing.

### 2.4 Sproull $k - d$ Tree

The Sproull $k - d$ Tree [4], is a radical refinement to the $k - d$ tree. The choice of a partition planes is not orthogonal or "coordinate based," as in a typical $k - d$ tree. Rather, it is chosen by computing the principal eigenvector of the covarience matrix of the points. Since finding an eigenvector of a $dxd$ symmetric matrix is an $O(d^3)$ operation, this increases preprocessing time as the dimension increases.

### 2.5 $VP$-tree

The Vantage Point tree is a data structure that chooses vantage points to perform a spherical decomposition of the search space. Yianilos [5] claims this method is suited for general metric spaces and for lower dimensional objects embedded in a higher dimensional setting.

## 3 Performance

### 3.1 Point Sets

Because the algorithms we are studying are heuristic, their performance is highly sensitive to the distribution of the input data. To get an idea of how well these structures perform, we have implemented several non-uniform distributions, many of which were defined by Bentley in [2].

- *Uniform* - random points inside the unit $d$-cube $[0, 1]^d$.
- *Ball* - random points inside the unit $d$-sphere.
- *Arith* - $x_1$ = 0,1,4,9,16,...; $x_2 = ... = x_{k-1} = 0$
- *Normal* - random points where $x_1, \ldots, x_d$ are generated from a normal distribution with mean 0 and variance 1.
- *Annulus* - $x_1$, $x_2$ uniformly random on the unit circle, with $x_3, \ldots, x_d$ from *uniform(1)*.
- *Cubediam* - $x_1$ from *uniform(1)*, $x_2, \ldots, x_d = x_1$, giving uniformly sampled points from a line segment in $d$ space with slope 1, starting at the origin and going to $(1, 1, \ldots, 1)$.
- *Corners* - $x_1, x_2$ drawn uniformly around $(0, 2)$, $(2, 0)$, $(0, 0)$, and $(2, 2)$ with $x_3, \ldots, x_d$ from *uniform(1)*.
- *Clusnorm* - Normal(0.05) at ten points uniformly sampled from $[0, 1]^d$.

- *Grid* - $N$ points from a grid hypercube with $1.3N$ points.
- *Spokes* - $N/d$ points uniformly sampled from $([0,1], 0.5, \ldots, 0.5)$ $N/d$ points uniformly sampled from $(0.5, [0,1], \ldots, 0.5)\ldots$
- *Henon* - A canonical chaotic attractor defined by a pair of interacting non-linear equations, viewed as a time series.

### 3.2 Experiments

Limited experiments were performed on the data sets. The $v-p$ tree was not analyzed because the current implementation requires $O(n^2)$ preprocessing time, making the analysis of large data sets difficult. This problem can be fixed easily, if the need arises.

Our first experiment was to test the different preprocessing times of the algorithms as the size of the data set changes. Table 1 summarizes the results of this experiment on a uniformly distributed data set in three dimensions. As expected, the preprocessing time for the Naive $k-d$ tree is the smallest, while the $Mediank-d$ tree and the Sproull $k-d$ tree are require more time. The second half of the table shows the search times of the $Naive$ method, the Median $k-d$ tree, and the Sproull $k-d$ tree. (The Naive $k-d$ tree does not support nearest neighbor queries). Observe that the search times for both trees are superior to the $Naive$ algorithm. Similar results are obtained in Table 2, where the data set is the Henon Attractor in three dimensions. However, the preprocessing time is greater in the latter experiment.

The rather disconcerting results of another experiment are shown in Table 3. Here, we fix the number of points, and progressively increase the dimension. Notice that the Sproull $k-d$ tree requires more preprocessing time as the dimension increase because of the eigenvector computations. The search times show that that even in dimensions as low as 4, the $Naive$ algorithm is superior. This places some doubt in the usefullness of multi-dimensional trees in searching higher dimensional spaces.

## 4 Conclusions

The disturbing performances of the search trees listed in Table 3 can be explained by "the curse of dimensionality" which states that as the dimension increases, points start to become very nearly equidistant. This problem restricts the use of multi-dimensional search trees in higher dimensions, and suggests that better methods must be developed for applications in higher dimensional settings.

| | Preprocessing Time | | | Search Time | | |
|---|---|---|---|---|---|---|
| $N$ | Naive kd | Median kd | Sproull kd | Naive | Median kd | Sproull kd |
| 10000 | 0.7 | 1.2 | 2.5 | 0.06 | 0.09 | 0.03 |
| 20000 | 1.8 | 2.8 | 6.1 | 0.12 | 0.10 | 0.04 |
| 30000 | 2.5 | 4.8 | 11.1 | 0.18 | 0.11 | 0.05 |
| 40000 | 4.4 | 8.4 | 14.5 | 0.23 | 0.12 | 0.04 |
| 50000 | 6.0 | 10.5 | 18.9 | 0.29 | 0.12 | 0.05 |

Table 1: Results of preprocessing times and searching for the thousand points closest to the origin in a three dimensional uniformly distributed data set containing $N$ points. (All times given are in seconds)

| | Preprocessing Time | | | Search Time | | |
|---|---|---|---|---|---|---|
| $N$ | Naive kd | Median kd | Sproull kd | Naive | Median kd | Sproull kd |
| 10000 | 0.8 | 1.8 | 3.2 | 0.08 | 0.08 | 0.03 |
| 20000 | 2.0 | 6.3 | 8.2 | 0.13 | 0.10 | 0.04 |
| 30000 | 3.6 | 9.9 | 15.9 | 0.17 | 0.12 | 0.05 |
| 40000 | 4.9 | 17.0 | 24.2 | 0.22 | 0.12 | 0.05 |
| 50000 | 6.1 | 18.5 | 27.0 | 0.29 | 0.12 | 0.05 |

Table 2: Results of preprocessing times and searching for the thousand points closest to the origin in a three dimensional Henon attractor containing $N$ points. (All times given are in seconds)

| | Preprocessing Time | | | Search Time | | |
|---|---|---|---|---|---|---|
| Dim | Naive kd | Median kd | Sproull kd | Naive | Median kd | Sproull kd |
| 2 | 0.8 | 1.0 | 1.7 | 0.07 | 0.05 | 0.03 |
| 4 | 0.9 | 1.4 | 3.2 | 0.08 | 0.12 | 0.07 |
| 6 | 1.0 | 1.9 | 6.3 | 0.09 | 0.15 | 0.12 |
| 8 | 0.8 | 2.4 | 10.7 | 0.10 | 0.17 | 0.16 |
| 10 | 0.9 | 2.8 | 16.1 | 0.11 | 0.20 | 0.20 |
| 12 | 1.0 | 3.4 | 27.0 | 0.15 | 0.23 | 0.23 |
| 14 | 1.0 | 4.1 | 33.6 | 0.16 | 0.26 | 0.25 |
| 16 | 0.9 | 4.3 | 45.0 | 0.17 | 0.27 | 0.28 |
| 18 | 0.9 | 4.8 | 58.9 | 0.19 | 0.29 | 0.29 |
| 20 | 1.0 | 5.4 | 77.4 | 0.19 | 0.31 | 0.31 |

Table 3: Results of preprocessing times and searching for the thousand points closest to the origin in uniformly distributed data sets of varying dimensions containing 10,000 points. (All times given are in seconds)

# References

[1] J. Bentley. Multidimensional binary search trees used for associative searching. *Comm. ACM*, 18:509–517, 1975.

[2] J. Bentley. $k-d$ trees for semi-dynamic point sets. In *Proc. Sixth ACM Sympos. Computational Geometry*, pages 187–197, 1990.

[3] J. Friedman, J. Bentley, and R. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3:209–226, 1977.

[4] R. Sproull. Refinements to nearest-neighbor searching in $k$-dimensional trees. *Algorithmica*, 6, 1991.

[5] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. Fourth ACM-SIAM Sympos. Discrete Algorithms*, pages 311–321, 1993.

# A User's Manual for Ranger

## A.1 Introduction

The *Ranger* program was developed to study several algorithms for nearest neighbor and orthogonal range queries in higher dimensional spaces. It provides facilities for viewing projections of point sets in high dimensional spaces, the decomposition of the spaces associated with a searching algorithm on that point set, queries, and in some cases, animation of the search algorithm. In addition, timing operations are included to decide how appropriate a given data structure is for an input distribution.

One design principle in windows programming is to display buttons and choices on the screen if and only if they are useful at that moment. This principle was used extensively in *Ranger*. The top menu bar of *Ranger* consists of the following selections: **Data, Display, Algorithm, Query, Timing,** and **Help**. Descriptions of each of the functions follow below.

## A.2 Data

The **Data** pull down button allows the user to specify the source of the input data. The data can come from an external file or from one of the "canned" distributions explained in the report (**From File** and **Distribution**, respectively).

### A.2.1 From File

If the data is to be loaded from an external file, select the **From File** option and enter the appropriate file name in the file selection window. The file should be a text file with the number of points in the file as the first record, the dimension of the point set as the second record, with the coordinates of the points as subsequent records. A successfully loaded file will result in an informational box being placed below the main window telling the filename, the number of points, and the dimension. In addition, the **Display** menu option should be enabled.

### A.2.2 Distribution

If the data is to be loaded from one of the canned distributions, defined in the above report, select the **Distribution** option. A data entry box should appear below the main window. Pressing the left mouse button over the **Distribution** pull down window allows the selection of the desired distribution. Text windows are given that allow the specification of the number of points and the dimension of the point set to be generated. Press the **Generate Data** button to generate the data. After doing so, the **Display** menu option should be enabled.

## A.3 Display

After the input has been specified, the **Display** button is now enabled. This options allows the user to view the point set, data structure, and in some cases animation of the algorithm. Choosing either **2-d** or **3-d** will display a selection area entitled **Display Parameters** below the **Data Parameters** board. After all of the parameters have been set, pressing the **Display Data** button will show the picture.

### A.3.1 2-d

For two dimensional projections of the point set, choose the **2-d** option of the display. Use the *X-Coord* and the *Y-Coord* pulldowns to set the projection dimension for the X and Y Axis.

The **Picture** pull down allows the choice of several different views of the data.

- *Points* simply shows the point set, choosing the projection dimensions from what was specified in *X-Coord* and *Y-Coord.*
- *Data Structure* show the spatial decomposition associated with the algorithm currently selected. If no algorithm has been selected or the algorithm is *Naive* no data structure will be shown.
- *dxd Projection* shows a Dimension by Dimension matrix of each projection of the point set. The $i, j$th element of this matrix represents the $i$th dimension plotted vs the $j$th dimension of the point set. Observe that the "diagonal" of this matrix is simply a straight line and that in some sense this is "symmetric."
- *Delaunay* displays the Delaunay Triangulation of the current projection of the point set. Try running this on a *grid* distribution to see robust the implementation really is.

### A.3.2 3-d

The *3-d* option is similar in functionality to the *2-d* option. The salient differences are that the *d x d projection* and the *Delaunay* triangulation cannot be shown and that rotation buttons appear that can be used to rotate the object around the screen.

## A.4 Algorithm

The *Algorithm* pull down menu allows the specification of many candidate algorithms described in the report. After the algorithm has been selected, the **Build Data Structure** button will be enabled.

## A.5 Query

The *Query* pull down menu allows the user to choose between the following queries: **Rectangular, Nearest Neighbor,** and **Nearest Neighbor Graph**.

### A.5.1 Rectangular

Choosing a rectangular range query displays a menu box beneath the *Display Parameters* board. To specify a range, type the intervals of the range in the *Range* text box, for each of the coordinates using the the following syntax:

$$[(x_{1_{min}}, x_{1_{max}}), (x_{2_{min}}, x_{2_{max}}), \dots(x_{d_{min}}, x_{d_{max}})]$$

For example, to specify a range query on a two dimensional point set, with the x-interval ranging from -0.5 to 0.5 and the y-interval ranging from 0.25 to 0.5, type:

$$[(-0.5, 0.5), (0.25, 0.5)]$$

### A.5.2 Nearest Neighbor

Choosing a nearest neighbor query displays the nearest neighbor menu box beneath the **Display Parameters** board. To specify a nearest neighbor query, enter the number of nearest neighbors you would like to return in the *# of Neighbors* text box and specify the query point using the following syntax:

$$(x_1, x_2, \dots, x_d)$$

Then specify the metric under which you would like distances comparisons to be made. *Ranger* supports the following metrics:

- *Euclidean* is the standard Euclidean ($L_2$)metric.
- *Manhattan* is the $L_1$ metric.
- *L_infinity* is the $L_\infty$ metric.
- *L_P (specify P)* is a generalized Minkowski metric, where the user is prompted to enter a value for $p$.

### A.5.3 Nearest Neighbor Graph

Choosing the nearest neighbor graph query displays the the nearest neighbor graph menu box beneath the **Display Parameters** board. To specify a nearest neighbor graph, first enter the number of nearest neighbors you would like to see on the graph. (Note: this number should be greater than one, because the nearest neighbor of a point in the set $S$ is itself.) Next, select the metric in a manner similar to the Nearest Neighbor query.

## A.6 Timing

Selecting the pull down menu **Timing** will allow you to toggle timing on or off. If timing is enabled, timing messages will be written to stdout.

# B Programmer's Manual for Ranger

This appendix describes the design and implementation of the *Ranger* program.

## B.1 Environment

*Ranger* is written in C on an SGI workstation under the UNIX operating system and has been successfully ported to Hewlett-Packard Workstations. X-Windows with the Motif widget set is used for the user interface. A graphics library, VOGL *Very Ordinary Graphics Language*, a C implementation of hardware encoded graphics language of the SGI and IBM RISC workstations, is used to display the point sets, data structures, and animations of searches. VOGL can be easily ported to any UNIX workstation supporting X-Windows.

## B.2 Implementation

Care was taken to categorize modules to isolate library (XWindows and VOGL) calls. Here is a breakdown of the modules used by *Ranger*:

X-Windows and interface routines:

- **XWidgetsCreator.c** creates all of the X-Motif widgets in the *Ranger* window. This code was generated using *X-Designer*, a Motif code generator.
- **XCallbacks.c** contains all of the callbacks to the widgets created by XWidgetsCreator.c.
- **queryparser.c** is used by XCallbacks.c to parse the queries for nearest neighbors and range searches.
- **distrib.c** contains code to generate point sets from the "canned" distribution functions.
- **fileoper.c** contains code to read an external file as data into the program.

VOGL Graphics routines:

- **drawing.c** contains code to perform drawing operations on the *Tonto* window.
- **drawsproull.c** is an #include file to drawing.c which draws a Sproull kd-tree.
- **drawvp.c** is an #include file to drawing.c which (only partially correctly) draws a vantage point tree.
- **animateoptkd.c** is an #include file to optkd.c which contains code to animate a nearest neighbor search.
- **graphs.c** contains code to display nearest neighbor graphs.

Algorithms routines:

- **metric.c** contains code to implement the various distance metrics used by the program.
- **naive.c** contains code to perform nearest neighbor and rectangular range queries in the most naive fashion imaginable.
- **pqueue.c** contains code to maintain a priority queue for nearest neighbor searches. This module is used by every algorithms module that performs nearest neighbor searches except for naive.c.
- **naivekd.c** contains code to build k-d trees as originally described by Bentley and perform rectangular range queries using them.
- **optkd.c** contains code to build refined k-d trees and perform nearest neighbor and orthogonal range queries using them.
- **sproullkd.c** contains code to create Sproull k-d trees and perform nearest neighbor queries using them.
- **eigens.c** contains code to compute the principle eigenvector of a symmetric matrix. This is used by sproullkd.c.
- **vptree.c** contains code to implement vantage point trees and perform nearest neighbor queries using them.
- **VorMain.c** contains code written by Steve Fortune to display the Delaunay triangulation. For some reason, the Voronoi Diagrams output by this program do not seem to work.