Applying a Randomized Nearest Neighbors Algorithm to Dimensionality Reduction

by

Gautam Jayaraman

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2003

Author .... [illegible]
Department of Electrical Engineering and Computer Science
May 21, 2003

Certified by .... [illegible]
Joshua B. Tenenbaum
Assistant Professor of Cognitive Science and Computation
Thesis Supervisor

Accepted by .... [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Applying a Randomized Nearest Neighbors Algorithm to Dimensionality Reduction

by

Gautam Jayaraman

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering

Abstract

In this thesis, I implemented a randomized nearest neighbors algorithm in order to optimize an existing dimensionality reduction algorithm. In implementation I resolved details that were not considered in the design stage, and optimized the nearest neighbor system for use by the dimensionality reduction system. By using the new nearest neighbor system as a subroutine, the dimensionality reduction system runs in time $O(n \log n)$ with respect to the number of data points. This enables us to examine data sets that were prohibitively large before.

Thesis Supervisor: Joshua B. Tenenbaum
Title: Assistant Professor of Cognitive Science and Computation

Acknowledgments

I would like to thank:

- Professor **Josh Tenenbaum** for supervising and patiently teaching me. He reeled me in with an exciting research topic, and throughout the year he consistently pushed me in constructive ways.
- Professor **David Karger** for helping me find and complete the project. As my academic advisor he helped me plan my scholastic career for the past three years; as a resource in my thesis work he assisted me with analysis of his data structure.
- **Divya Bhat** for constantly encouraging, flattering, and teaching me, all from three thousand miles away.
- **Neil Basu, Casey Muller, Charles Floyd, and Deb Dasgupta** for keeping me entertained and healthy throughout the thesis process, but more importantly, for filling the past five years of our lives with laughter and music.
- My sister **Anu Jayaraman** for her quiet blend of admiration and protectiveness, both of which have always been reciprocated.
- My parents, Drs. **Krishna and Nilima Jayaraman**, for generously funding my work, and supporting me always without hesitation.

Contents

List of Figures

Chapter 1

Introduction

Dimensionality reduction is a technique used to express a high-dimensional data set in fewer, more meaningful dimensions while preserving the intrinsic variability of the data. Nearest neighbor searching is a well-studied problem that has proven very applicable in dimensionality reduction problems.

This chapter provides background by introducing the concepts of dimensionality reduction and nearest neighbor searching. It also explains the purpose of this thesis, as well as how the rest of the paper is organized.

1.1 Dimensionality Reduction

1.1.1 Definition

The problem of dimensionality reduction involves mapping high-dimensional inputs into a lower-dimensional output space. The goal is to represent the given data set in as few dimensions as possible while retaining its intrinsic modes of variability.

The premise of dimensionality reduction is that a data set representing features observed in a high-dimensional space typically lies in a subspace that can be represented in fewer dimensions. When creating a new set of dimensions, each new dimension is some function of the old dimensions. The problem of dimensionality reduction is to define this new set of dimensions that more succinctly describe the data.

1.1.2 Purpose

The intent of dimensionality reduction is to make a data set easier to interpret and analyze. In many areas of science, data is gathered by observing many different characteristics of a situation and grouping the observations together as one data point. In observing many different situations, the result is a data set composed of high-dimensional data points, where each data point is a vector of observations about a certain situation.

Though the observations are taken in a very high-dimensional space, it is often the case that some of the dimensions are related to others. In *nonlinear* dimensionality reduction, the goal is to define a set of new dimensions that describes the relationships as precisely and succinctly as possible, using nonlinear functions as needed.

After applying dimensionality reduction techniques to a data set, the data set is viewed in terms of the new minimal set of meaningful dimensions (as in Figure 1-2). Or in order to understand the significance of each intrinsic dimension individually, the data points can be plotted (on a line) with respect to a single intrinsic dimension.

1.1.3 Applicability and Canonical examples

Dimensionality reduction techniques are applicable to various areas of science. In machine learning a common problem is to mimic the human visual system by gathering data from thousands of sensors and extracting meaningful low-dimensional structures (see Figure 1-2). A goal in climate analysis is to infer meaningful trends using data points composed of all the parameters measured by modern weather instruments.

In order to give the reader a better feel for situations in which dimensionality reduction is useful, this portion presents some canonical examples that illustrate different facets of dimensionality reduction.

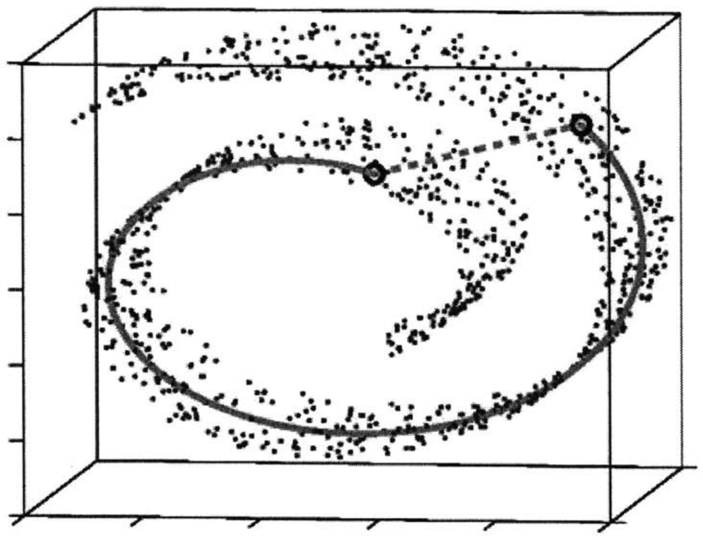

Figure 1-1: The swiss roll data set. Dotted line shows Euclidean distance between two points, and solid line shows distance along the manifold.

Swiss roll

The swiss roll is a set of data points that lie in a manner resembling a swiss cake roll pastry. This data set helps illustrate some of the basic principles and terminology of dimensionality reduction.

What is immediately apparent in Figure 1-1 is a two-dimensional "sheet" of points that has been "rolled up" so that it lies in three-dimensional space. The swiss roll data set allows us to see intuitively the concept of a nonlinear *manifold*, which is a "surface" on which the data points lie. In this case, since the manifold is two-dimensional, the data set is defined to have *intrinsic* dimensionality 2. Similarly, since the embedding is given in three dimensions, the data set is defined to have *extrinsic* dimensionality 3.

Faces

The faces data set presents a dimensionality reduction problem that the human visual system solves all the time. Given sensory data from millions of retinal receptors, how do people distill the data into useful information about the structures that they see?

This data set contains computer-generated images of a human face, each one rep-

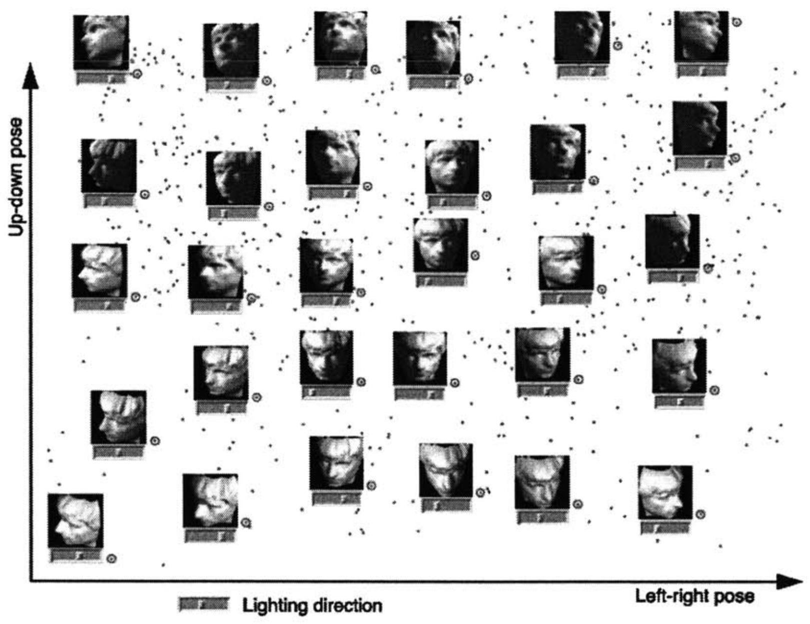

Figure 1-2: ISOMAP output on the faces data set, showing three intrinsic degrees of freedom.

resented as a 64x64 pixel grayscale image. This presents an input space of 4096 dimensions, where each image is expressed as a 4096-dimensional vector of brightness levels for each pixel. Though expressed in a very high-dimensional space, the *variations* among the images are apparent to the human eye upon a quick glance at Figure 1-2. Just three intrinsic degrees of freedom emerge among the images: the pose angle of the head varies both vertically and horizontally, and the lighting angle varies horizontally. Dimensionality reduction allows the computational discovery of what our eyes already know– that all the images lie in a three-dimensional subspace of the original high-dimensional space.

1.2 The ISOMAP Algorithm

This research was initially motivated by a desire to improve the performance of a nonlinear dimensionality reduction algorithm called ISOMAP [2]. The nearest neighbor system implemented in this thesis is applicable to a variety of systems in data analysis [4], machine learning [7], and Internet applications [8]. We use ISOMAP as just one example to assert the applicability of the nearest neighbor structure.

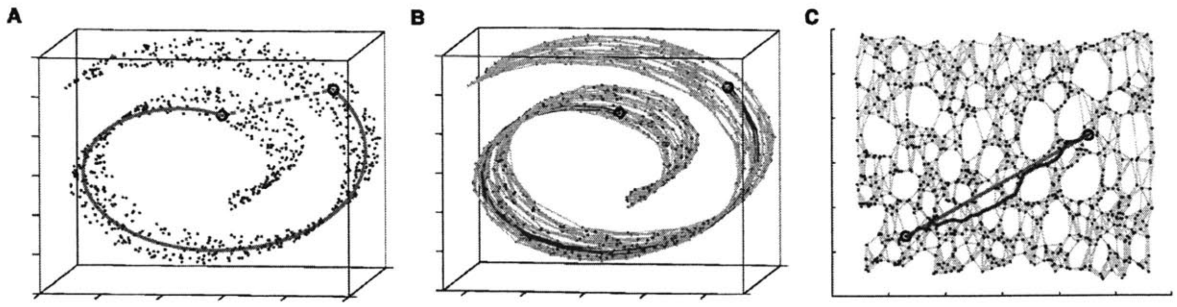

Figure 1-3: The swiss roll data set. Left frame shows data points in 3-d embedding, middle shows neighborhood graph that approximates the manifold, right shows the 2-d embedding output by ISOMAP.

1.2.1 How ISOMAP works

To reduce the dimensionality of a data set, ISOMAP uses a strategy that is fairly intuitive given the concept of a nonlinear *manifold* (introduced in Section 1.1.3). In this class of problem the data points are assumed to all lie on some manifold, which defines the subspace we are trying to determine. Given a set of data points embedded in a high-dimensional space, ISOMAP first approximates the manifold defined by the data points, and then "unwraps" the manifold to embed it in a lower-dimensional space while preserving its geometry. This process is visually summarized with the swiss roll data set in Figure 1-3.

First step: Approximating the manifold

For the first step ISOMAP constructs a graph on the data set by connecting each data point to its nearest neighbors. The resulting *neighborhood graph* approximates the manifold, as we see in the middle frame of Figure 1-3. This step of ISOMAP provides the basis for this thesis work, in that it calls for the use of a nearest neighbor algorithm. The specifics of the neighborhood graph problem are covered in Section 1.3.3.

Remaining steps: "Unwrapping" the manifold

For the remaining steps, ISOMAP first approximates the *geodesic* distances between all pairs of points. The geodesic distance between two points is defined as the distance *along the manifold* between the two points. Once again, this concept can be grasped intuitively using the swiss roll as an example. The geodesic path is visible as a solid line in Figure 1-1.

In order to approximate the geodesic distances between all pairs of data points, ISOMAP calculates all pairs shortest path distances (using Dijkstra's or Floyd's algorithm) on the neighborhood graph. Proofs and extensive discussion [2] about this phase of ISOMAP are outside the scope of this thesis, and therefore omitted. As long as there is adequate density of points on the manifold, the shortest path on the neighborhood graph provides a good approximation (specifically an upper bound) on the actual geodesic distance (see right frame of Figure 1-3). Further, there are assumptions of limited curvature [2] of the manifold that allow us to ignore possibilities of our neighborhood graph "short-circuiting" the manifold (see dotted line in Figure 1-1). Typical data sets conform to these requirements of limited curvature, allowing them to be predictably analyzed by ISOMAP.

In its final step, ISOMAP constructs a lower-dimensional embedding for the manifold while preserving the geodesic distances. The technique used for this is classical *multidimensional scaling* (MDS), which is a proven method for *linear* dimensionality reduction [9]. Though discussion of MDS is omitted due to scope, there is a simple explanation as to how ISOMAP uses MDS to perform nonlinear dimensionality reduction: By applying MDS only to the *geodesic* distances (instead of the distances in the original embedding), ISOMAP effectively removes the nonlinear aspect of the problem prior to invoking MDS.

1.2.2 Interpreting output

In this section we describe the nature of the output given by ISOMAP and show examples using the three data sets we introduced in Section 1.1.3.

Given a new embedding (produced by ISOMAP) of a data set, we can evaluate it by calculating the *residual variance* [2], which quantifies how well the distance between two points in the new embedding approximates their true geodesic distance.

ISOMAP does not, in itself, "figure out" the intrinsic dimensionality of a data set. Rather, it is a tool that allows a user to accomplish this goal. ISOMAP takes an input parameter d and returns the optimal d-dimensional embedding of the input data. Then the validity of that reduction can be evaluated by calculating its residual variance. This suggests that a user interaction with ISOMAP would be an iterative process during which we compute embeddings for values of d. Due to the underlying linear algebra [2], it is more efficient to work with *decreasing* values of d because we can reuse results from previous iterations to speed up the computation of a new embedding. With that consideration we propose the following strategy:

1. Set d to be an upper bound guess of the intrinsic dimensionality of the data set.
2. Run ISOMAP to embed the data set in d dimensions.
3. While residual variance is lower than the required level: decrease d and run ISOMAP again.

Once the user discovers a reasonable range of values for d (which corresponds to a reasonable range of residual variances), a second strategy can be used. This strategy is to try values of d throughout the reasonable range, and find the point where increasing d no longer significantly reduces the residual variance.

1.3 Nearest Neighbors

1.3.1 Definition

The nearest neighbor problem has been the subject of much research due to its wide applicability. The literature has defined it in many ways, so for clarity we provide our own definition here.

The nearest neighbor problem as we consider it takes place within a framework called a *metric space*, which is a combination of a space M and a function $d(m_1, m_2) \to r$ that returns the distance between any two points $(m_1, m_2) \in M$. For the nearest neighbor problem, given a set $S \subset M$, $|S| = n$ of data points, we define the following searches:

Definition 1 *The* nearest neighbor *of a point $q \in M$ is the point $s_i \in S$ such that $d(q, s_i)$ is minimized.*

Definition 2 *The* k-nearest neighbors *of a point $q \in M$ comprise the* set $s \subseteq S$ *such that*

$$|s| = k \text{ and } \forall[s_i \in s, s_j \in \{S - s\}] : d(q, s_i) \leq d(q, s_j).$$

Definition 3 *The* ϵ-nearest neighbors *of a point $q \in M$ comprise the* set $s \subseteq S$ *such that*

$$\forall[s_i \in s, s_j \in \{S - s\}] : d(q, s_i) \leq \epsilon \text{ and } d(q, s_j) > \epsilon.$$

1.3.2 Purpose

Applicability

The nearest neighbor problem is applicable in many contexts. A simple spatial application is mapping a customer to their nearest service provider, e.g. dispatching emergency vehicles to the scene of a fire. The dispatcher consults a map to determine which fire station is closest to the fire.

Another example that is spatial, but in a more "virtual" sense, occurs in delivering content through the Internet. Given several "mirror" servers that each carry the desired content, the user and the network save time if content is delivered from the mirror that is closest, i.e. the least number of routing "hops" away.

There are also applications that are less spatial. In classification we are given a training set of data points, each of which has been classified. For example, each data point in S could be the vector of attributes *(age, sex, education, income)* of a person, as well as a label of either "Democrat" or "Republican." Then given a new point based on a new person, we can classify it with the same label as its nearest neighbor in S.

Another application is in lossy compression, where we can encode a chunk of data using the closest match from among a small set of representatives. One image compression scheme [10] divides a picture into blocks of 8-by-8 pixels. It then checks each block against a predetermined set of "library" blocks, and replaces the block with the most similar representative from the library.

1.3.3 Nearest Neighbor component of ISOMAP

This research focuses on the part of the ISOMAP algorithm that constructs a neighborhood graph on the data set by determining the nearest neighbors for each data point.

In the case of ISOMAP, data sets are embedded in a high dimensional space, and the distance function is typically straight-line Euclidean distance (though this is not a requirement). In constructing the neighborhood graph, ISOMAP connects each data point to either its ϵ- or k-nearest neighbors. This approximates the manifold, relying on the idea that when looking only at nearby points, Euclidean distance is a good approximation for geodesic distance [7].

A subtype of the nearest neighbor problem

For construction of the neighborhood graph, ISOMAP must solve a very specific subtype of the standard nearest neighbor problem. Some literature refers to this as the *all nearest neighbors* problem [12], [11]; here we call it *neighborhood graph construction.* There are two defining characteristics of neighborhood graph construction:

- **Only points within the data set S are queried.**
  Conventional nearest neighbor algorithms support nearest neighbor queries on general points in the space M. Since neighborhood graph construction only involves queries on points in S, there is no need to consider distances other than those between the $O(n^2)$ pairs of points in S.

- **Each of the n data points in S is queried exactly once.**
  This has implications on performance assessment because it adds the knowledge that exactly n queries will take place. In a data-structural framework, there will be one build phase and n queries, instead of a boundless number of queries. Knowing a bound on the number of queries may permit the de-emphasis of

query time in relation to build time by optimizing on the $(1 : n)$ ratio of builds to queries. These considerations are more thoroughly discussed in the analysis in Section 4.3.

Performance bottleneck

The all-points nearest neighbor calculation required by ISOMAP can be solved in $O(n^2)$ time brute-force by determining the Euclidean distances between all $\binom{n}{2}$ pairs of data points. All the other phases of ISOMAP can be executed in $O(n \log n)$ time, so the neighborhood graph-calculating step is considered a performance bottleneck. The goal is to create the neighborhood graph in time $O(n \log n)$, allowing ISOMAP to achieve an $O(n \log n)$ overall running time.

1.4 Purpose and Organization

The purpose of this thesis is to assert the applicability of the KR algorithm to a variety of problems. This is done using dimensionality reduction and ISOMAP as an example of a potential application. The thesis goal is met by demonstrating a full implementation that performs well on low-dimensional data in high-dimensional embeddings.

The remainder of the paper is organized as follows: Chapter two introduces the KR algorithm, describing its design as originally published [1]. Chapter three discusses our implementation of the KR nearest neighbor structure, and the issues that had to be resolved to produce an efficient, effective system. Chapter four assesses the performance of our implementation, evaluating the KR algorithm as well as the results of the decisions and design alterations enacted during the implementation stage. Finally, chapter five provides conclusions and suggestions for further research in this area.

Chapter 2

The Karger-Ruhl Near Neighbor Algorithm

This chapter introduces the Karger-Ruhl near neighbor algorithm and data structure (hereafter called the KR algorithm) [1]. Its distinguishing characteristics, structure, and theoretical performance benefits are discussed here, with emphasis on those facets that benefit ISOMAP directly. Implementation optimizations and empirical performance of the KR algorithm are discussed in subsequent chapters.

2.1 Background and Motivation

Among a diverse field of nearest neighbor schemes, the KR algorithm differentiates itself in important ways. Most nearest neighbor algorithms ([11], [5], [6], [12]) scale exponentially with the extrinsic dimensionality of the data, making them a poor choice for ISOMAP input data, which is, by definition, of inflated extrinsic dimensionality.

The KR algorithm is designed to operate on *growth-constrained* metrics (defined in Section 2.1.3 below), and as a result it scales roughly exponentially in the *intrinsic* dimensionality of the data set (see Section 2.3 for more discussion). This feature alone makes it worthy of implementation and testing with ISOMAP, but its compatibility with a variety of data sets and metrics is also an important strength.

2.1.1 General metrics

Unlike most nearest neighbor research, the KR algorithm is not designed specifically for the Euclidean case. It is compatible with general spaces and a large class of distance metrics.

2.1.2 Distance functions

The KR algorithm places two constraints on the distance function that is used:

1. **It must be symmetric:**
   $\forall p_i, p_j \in M : d(p_i, p_j) = d(p_j, p_i)$

2. **It must obey the triangle inequality:**
   $\forall p_i, p_j, p_k \in M : d(p_i, p_k) \leq d(p_i, p_j) + d(p_j, p_k)$

These criteria are obeyed by most intuitive distance metrics, including Euclidean (L_2) distance. But note that certain alterations to the distance metric can "break" its support of these criteria. Consider the case of Euclidean distance squared (which is easier to compute because the square-root operation is omitted): If we have three points (a, b, c) on a number line at $(0, 2, 4)$, respectively, then using the Euclidean distance squared metric, $d(a, b) = 4$, $d(b, c) = 4$, and $d(a, c) = 16$. The triangle inequality is violated because $16 > (4 + 4)$.

2.1.3 Low-growth data sets

The KR algorithm guarantees correctness in arbitrary metric spaces that obey symmetry and the triangle inequality. But certain aspects about the *growth rate* of the data set are also important to the KR algorithm's correctness and efficiency. Before elaborating on this growth requirement, some definitions are necessary:

Definition 4 $B_r(p) := \{s \in S \mid d(p, s) \leq r\}$ *indicates the ball of radius r around p in S.*

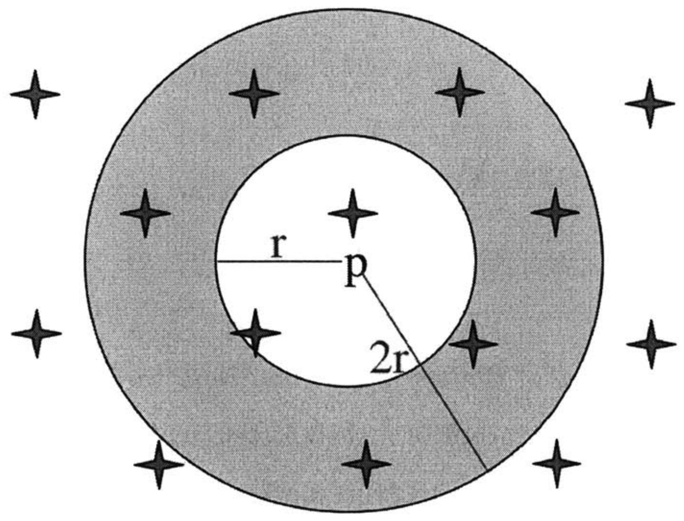

Figure 2-1: Expansion rate example in two dimensions. Since $|B_r(p)| = 2$ and $|B_{2r}(p)| = 8$, expansion around $p = 8/2 = 4$.

Definition 5 *We say that a data set S has* (ρ,c)-expansion *iff for all $p \in M$ and $r > 0$,*

$$|B_r(p)| \geq \rho \Longrightarrow |B_{2r}(p)| \leq c \cdot |B_r(p)|$$

Throughout this thesis we refer to c as the "expansion rate" of S. Intuitively, c bounds the rate at which new points from S "enter" a ball that we expand around some point $p \in M$.

We now consider the relationship between expansion rate and intrinsic dimensionality. If we think of points scattered uniformly on a two-dimensional manifold, then a "ball" is a disk (see Figure 2-1). We find that doubling the radius of a ball results in a ball containing four times as many points, since the area of the disk (given by πr^2) scales with the square of the radius. Similarly if the points come from a three-dimensional manifold, a sphere of double the radius will contain eight times as many points. Thus we infer that expansion rate is roughly 2^d, where d represents the intrinsic dimensionality of the data.

The expansion rate c of a data set is important in several different ways. The correctness of some parts of the KR algorithm (see Sections 3.3.2 and 2.2.8) rely on the knowledge of c and the smoothness it implies by definition. The time analysis of the KR algorithm assumes c is a constant, so real-world efficiency relies on c being small. This issue becomes a focus in later chapters.

2.2 How the KR Algorithm Works

2.2.1 Search step

A few basic principles govern the structure of the KR algorithm. Here we introduce these principles as lemmas, followed by the concept of a *search step*, which is a situation that arises in nearly every function the data structure performs.

Subset Lemma:
A random subset of m points from a metric space with (ρ, c)-expansion will have $(\max(\rho, O(\log m)), 2c)$-expansion with high probability.

Sandwich Lemma:
If $d(p,q) \leq r$, then $B_r(q) \subseteq B_{2r}(p) \subseteq B_{4r}(q)$.

Sampling Lemma:
Let M be a metric space, and $S \subseteq M$ be a subset of size n with (ρ, c)-expansion. Then for all $p, q \in S$ and $r \geq d(p,q)$ with $|B_{r/2}(q)| \geq \rho$, the following is true:

When selecting $3c^3$ points in $B_{2r}(p)$ uniformly at random, with probability at least $9/10$, one of these points will lie in $B_{r/2}(q)$.

Though their proofs from the original paper [1] are not repeated here, these lemmas combine to reveal a good basic search strategy. First we put all the data points from S in an absolute ordering, $s_1, s_2, ..., s_n$. A *search step* refers to a situation where we are given a query point q and a "current point" s_i, $d(s_i, q) = r$ where our search currently stands. We look in a ball around s_i to find the next search point s_j. The hope is that we find an s_j that is closer to q than s_i was. For analysis we define a criterion for the "success" of a search step. In the case of the KR algorithm, this criterion ranges from $(d(s_j, q) \leq r/2)$ to $(d(s_j, q) < r)$.

Though this phenomenon occurs in all functions of the data structure, it is easiest to grasp in terms of a single-nearest neighbor query. For a nearest neighbor query on a point q, we keep taking search steps to bring the current point closer to q, until the current point *is* the closest point to q, at which time we return the current point.

The three lemmas allow us to analyze a single search step, and determine how to maximize the probability that it is successful and efficient. The subset lemma allows us to work with a random subset of data points without suffering unbounded increases in the growth rate. The sandwich lemma uses the triangle inequality to allow a near neighbor search to "close in" on its target output. The sampling lemma tells us how many points we need to sample in order to find a "good" one with high probability. The sampling lemma and subset lemma together guarantee that if we sample $3(2c)^3 = \mathbf{24c^3}$ points from a ball around s_i then with probability $> 9/10$, an s_j within $r/2$ of q will exist in our sample.

2.2.2 Build and Query framework

Given a set of data points, the KR algorithm builds its data structure by inserting the points individually. Once every data point has been inserted, the structure can answer near neighbor queries on the data set. There are three kinds of queries supported, corresponding to the nearest neighbor definitions given earlier:

- `nearest-neighbor(`q`)`
- k`-nearest-neighbors(`q`,`k`)`
- ϵ`-nearest-neighbors(`q`,`ϵ`)`

2.2.3 Search strategy

The basic search strategy is to step through a *metric skip list* on the points, keeping track of the best choice(s) of points to return. Due to the structure of the skip list, all "good" points are visited but only a small, bounded number of "bad" points are seen during the course of the search.

A search starts at the beginning of the ordering (s_1), and jumps forward in the ordering with each search step until it reaches the end of the ordering (s_n). Each jump can be modeled as a step in a random walk: with some probability p the jump goes to a point that is closer to the query point, and with another probability $1 - p$ it gets farther away. The data structure is designed such that in the random walk along current points, the current distance r to the query point has a negative drift to converge quickly on the goal.

2.2.4 Structural components

The data structure is composed of n *nodes*, where a node is created by inserting a data point, such that node i represents the point s_i, and the "distance" between nodes i and j is just $d(s_i, s_j)$.

For each node the structure maintains several **finger lists**, each associated with a certain node and radius. The finger list at node i of radius r is defined as a random subset of all nodes within distance r of node i. The finger list size is a constant f (equal to $24c^3$ in the worst-case as shown in Section 2.2.1) chosen based on the sampling lemma and expansion rate of the data set.

To construct the finger lists as random samples from balls around a node, we first impose a *random* ordering ($s_1, s_2, ..., s_n$, corresponding to a random permutation of the points) on all the nodes. We can now precisely define a finger list in the context of the structure:

Definition 6 *The* finger list $F_r(i)$ *is the list comprised of the next f nodes in the ordering that are within distance r of node i*

Thus building a finger list is a deterministic process, but the resulting list represents a random sample since the ordering is random. Using this definition of a finger list, we can prove [1] that there will only be $O(\log n)$ distinct finger lists per node. This allows us to bound the space complexity of the data structure (see Section 4.3.1).

We briefly defer discussion of the insertion algorithm in order to explain the *Find* algorithm. Then *Insert* is explained as an extension of *Find.*

2.2.5 *Find* operation

Once all the data points have been inserted into the structure and all the finger lists have been constructed, queries on S are supported. The *Find* function uses the finger list framework to implement the search strategy suggested by the sampling lemma. It returns the single nearest neighbor.

```
Find(q) (finds nearest neighbor of $q$ in $S$)
    let $i = 1$ // $i$ is current node
    let $m = 1$ // $m$ is best node so far
    while $i < n$ do:
        let $r = d(s_i, q)$
        if $\exists j \in F_{2r}(i)$ such that
        $d(s_j, q) < d(s_m, q)$ or $d(s_j, q) \leq r/2$ then
            let $i$ be the smallest index $\in F_{2r}(i)$ with that property
            if $d(s_i, q) < d(s_m, q)$ then let $m = i$
        else let $i = \max F_{2r}(i)$
    output $s_m$.
```

2.2.6 *Insert* operation

When inserting a new node, we place it at the beginning of the ordering. We insert the nodes in reverse order $(s_n, s_{n-1}, s_{n-2}, ..., s_2, s_1)$, so that the resulting structure will have the nodes in the proper order. Recall that since this ordering was originally determined randomly, we can still model the finger lists as random samples.

Adding nodes only to the front of the structure simplifies insertions because we only need to construct finger lists for the new node; no other nodes or finger lists are affected. The new node's finger lists are constructed based on all the existing nodes, since all existing nodes occur later in the ordering. For the same reason, no existing finger lists need to be modified, since finger lists only "look" forward in the ordering.

The brute-force approach to construct the finger lists would be to start at the beginning of the data structure and move forward in the ordering, maintaining an array of the f nodes that are closest to the one being inserted. Every time a node is encountered that is closer than some node in the array, we replace the worst array element with the node just encountered. That way every time the array is altered, a new finger list is made, with a smaller radius than the previous one.

The brute-force approach will construct all the finger lists by the time it moves through all the existing nodes in the data structure. This is efficient in the beginning, when the array is not full and every node encountered is added to what will become the first finger list. But as the search goes on, updates to the array become less frequent and the brute-force method wastes more time looking at nodes that will never be in a finger list. To reduce the time it takes to build finger lists, the original publication [1] suggests using the *Find* algorithm to jump to the nodes that will actually appear in the finger lists. Details about our implementation of *Insert* are in Section 3.3.1.

2.2.7 *FindRange* and *FindK* operations

FindRange operation

This variant of the *Find* algorithm returns the ϵ-nearest neighbors of a query point q. The basic idea is to modify *Find* so that in its random walk over the ordering, it never misses any points within distance ϵ of q. This is accomplished with a couple of changes to the search step criteria.

```
FindRange($q, \epsilon$) (finds $\epsilon$-nearest neighbors of $q$ in $S$)
    let outputset = $\emptyset$
    let $i = 1$ // $i$ is current node
    let $m = 1$ // $m$ is best node so far
    while $i < n$ do:
        let $r = d(s_i, q)$
        if $r < \epsilon$ then $r = \epsilon$
        if $\exists j \in F_{2r}(i)$ such that
        $d(s_j, q) < d(s_m, q)$ or $d(s_j, q) \leq r/2$ or $d(s_j, q) \leq \epsilon$ then
          let $i$ be the smallest index $\in F_{2r}(i)$ with that property
          if $d(s_i, q) < d(s_m, q)$ then let $m = i$
          if $d(s_i, q) \leq \epsilon$ then add $s_i$ to outputset
        else let $i = \max F_{2r}(i)$
    output outputset.
```

FindK operation

This operation, which returns the k-nearest neighbors of a query point q, uses *FindRange* as a subroutine. The strategy is to make a series of ϵ-nearest neighbor queries around q, increasing ϵ until a query returns at least k points. Then we return the k best points from among those returned in the last ϵ query.

```
FindK($q, k$) (finds $k$-nearest neighbors of $q$ in $S$)
    let $B$ = FindRange($q, r$) for some $r$ such that $|B| = O(\log n)$
    while $|B| < k$ do:
        $r$ = Augment($r, B, q$)
        $B$ = FindRange($q, r$)
    output the $k$ points in $B$ closest to $q$.
```

In order to increase r appropriately in the *while* loop, we use the *Augment* procedure, which assures that the r value returned will increase $|B| = |B_r(q)|$ by a constant factor each time.

```
Augment($r, B, q$) (finds a new value of $r$ such that $|B_r(q)|$ grows geometrically)
    let $B = \{b_1, b_2, ...b_l\}$ be an enumeration of $B$'s elements
    for $i = 1$ to $l$ do:
        let $a_i$ be the element closest to $b_i$ in $b_i$'s finger lists with $d(a_i, b_i) > 2r$
        let $r_i = d(a_i, b_i)$
    output the median of $\{r_i \mid 1 \leq i \leq l\}$
```

These two algorithms defer a certain problem, namely determining the initial value of r for the *FindK* procedure, to the implementation phase. We discuss our solution to this problem in Section 3.3.2.

2.2.8 Dynamic updates

The KR algorithm is also designed to support dynamic structural updates. Supporting online insertion and deletion of nodes adds significant complexity to the structure, including a requirement of knowing c for correctness [1]. The primary reason for the added complexity is that we cannot dynamically insert a new node at the "back" of the structure, because the randomness of the ordering (and thus the randomness of the finger lists) would not be maintained.

New nodes must be inserted at a random position in the ordering; as a result, all of the finger lists for preceeding nodes are subject to change. The dynamic structure presented in the original publication [1] inserts a node to a random place in the ordering, and adjusts all of the affected finger lists automatically.

The dynamic structure was not implemented in this research because it is out of the scope of the thesis work, which focuses on building neighborhood graphs offline on a given data set. The value of the dynamic structure is undeniable, however, so others are encouraged to implement it using our offline implementation as a starting point.

2.3 Theoretical Performance

The KR algorithm is Las Vegas, which means that its correctness is guaranteed, but its time bounds are not. The time bounds given here hold with high probability; proofs from the original publication [1] are not repeated here.

The analysis leading to these bounds takes the expansion rate c as a constant, which is sensible because the expansion rate does not scale with the number n of data points. Since the finger list size f is theoretically a function of c, it is also treated as a constant.

In practice, we expect f to be polynomial in c (e.g. $24c^3$). Since c is exponential in the intrinsic dimensionality, we say that the KR algorithm's performance scales exponentially with respect to intrinsic dimensionality.

2.3.1 Single-nearest neighbor queries

The data structure performs nearest neighbor queries in $O(\log n)$ time. This is because a query is answered in $O(\log n)$ search steps. Each search step involves $O(f)$

distance computations, so in practice a factor of f could be included in the bound.

2.3.2 Insertions

Insertions are also performed in $O(\log n)$ time. This is because an insertion is like running a *Find* on the new point, except instead of maintaining the closest point so far, it maintains the f closest points so far. This causes the random walk of search steps to be slower than a regular *Find* by a factor of f, so in practice a factor of f^2 could be included in the bound.

2.3.3 ϵ- and k-nearest neighbor queries

The structure answers ϵ-nearest neighbor queries in $O(\log n + k)$ time, where k is the number of points returned. It answers k-nearest neighbor queries in $O(\log n + k)$ time. Each search step involves $O(f)$ distance computations, so in practice a factor of f could be included in the bound.

Chapter 3

Implementation

This chapter discusses the implementation of the KR algorithm. First the implementation context is discussed, in terms of the languages used and the interface between ISOMAP and the KR algorithm. Then the implementation is described in terms of the structure of its code modules. Finally, the important design completions, modifications and optimizations are discussed.

3.1 Implementation Framework

3.1.1 Integration with ISOMAP code

ISOMAP is implemented in MATLAB, which provides rich libraries to support all the necessary linear algebraic computation with minimal code. The KR algorithm requires requires a heavy load of primitive calculations, and so is not well-suited to the MATLAB environment.

3.1.2 Languages and technologies used

The KR algorithm was implemented in C++, which offers fast, efficient computation and control over memory, both valuable to our purpose. Also, MATLAB is designed to interface with C++ programs, allowing them to be invoked easily from within MATLAB programs.

Another valuable benefit of implementing the KR algorithm in C++ is the accommodation of object-oriented design. In our case this allows for the development of a modular system with a few different classes.

3.1.3 Motivations in modular design

The modularity of our implementation makes the code easier to understand and more extensible. For example, our implementation of the `Point` class focuses on the Euclidean case. In order to use the KR algorithm in a different metric space, only the `Point` class needs to be rewritten. Another example benefit of this modularity is that it simplifies the addition of real-world optimizations, some of which are discussed in Section 3.5.

3.2 Description of Modules

This section will introduce the modules used to implement the KR algorithm. Each module is introduced in terms of its purpose, as well as its member variables and functions. Some simplifications are made for clarity, so this should considered a pseudocode representation. The actual code that this section describes can be found in Appendix A.

3.2.1 `Point`

The `Point` class is used to represent data points. This class fully encapsulates the metric space. In our implementation a `Point` represents a point in Euclidean space. There are many different metric spaces that obey symmetry and the triangle inequality, and any of them can be accommodated by the KR algorithm with the appropriate changes to the `Point` class.

Member variables

- an integer `d` representing its dimensionality

- an array `coords` of decimals representing its coordinates in space.

Member function

- `dist(p)`, which calculates the Euclidean distance from this point to another `Point p`.

3.2.2 `FingerList`

The `FingerList` class stores a finger list, which is a list of points that represent a random sample of all the data points within a certain distance of the associated node.

Member variables

- an integer `length` representing its size
- a decimal `r` representing its radius (the distance of the farthest point, and hence the effective radius of the ball)
- an array `fingers`, which stores integers that refer to nodes by their rank in the ordering. For example, if points s_i and s_j are in the finger list, then the integers i and j are in the array `fingers`.
- a pointer `nextList` to the next `FingerList` (of smaller radius) belonging to this node

Member functions

- a method `getSize()` that returns `length`
- a method `getR()` that returns `r`
- a method `getListElt(i)` that returns `fingers[i]`
- a method `getNext()` that returns `nextList`
- a method `setNext(fl)` that sets the value of `nextList` to `fl`

- a method `findJ(q,rad)` that identifies (by rank) the lowest-ranked node in this finger list that is within distance `rad` of point `q`. If no such node exists, it returns the highest-ranked node in this finger list. This method implements the bulk of a search step.

3.2.3 `Node`

The `Node` class encapsulates the information that is added to the data structure when a `Point` is inserted. Accordingly, a node just keeps track of a set of `FingerLists`. It does this in the form of a linked list, so it only keeps track of the first `FingerList` then follows along the `nextList` pointers to subsequent finger lists.

Member variable

- a pointer `firstList` to the `FingerList` of largest radius that belongs to this node

Member function

- a method `getFirstList()` that returns `firstList`

3.2.4 `NNStruct`

The `NNStruct` class encompasses the entire data structure.

Member variables

- an integer `fingListSize` that represents the size of every finger list used or created by the structure
- an integer `numnodes` that represents the number of data points that the structure is allocated to represent
- an array `nodes` of the `Node` objects that comprise the structure

Member functions

- a method `getNode(k)` that returns `nodes[k]`
- a method `insert(p)` that inserts a point `p` at the front of the existing data structure, by creating a new `Node` and determining every `FingerList` associated with it
- a method `find(q)` that returns (by rank) the point in the data set that is the nearest neighbor of the point `q`
- a method `findRange(q,` ϵ`)` that returns (as an array of node ranks) all the points in the data set that are ϵ-nearest neighbors of the point `q`
- a method `findK(q, k)` that returns (as an array of node ranks) all the points in the data set that are `k`-nearest neighbors of the point `q`

3.3 Design Decisions

During implementation it became clear that certain details and optimizations were omitted from the original [1] description of the KR algorithm. Here we discuss our corresponding design decisions, along with how they impact the structure.

3.3.1 Completing the *Insert* design

The procedure for offline insertions is roughly sketched in the paper [1] in terms of modifications to the *Find* algorithm. No pseudocode was provided; this left some decisions for the implementation phase.

To adapt the *Find* algorithm for insertions we must ensure that it does not skip over any points that should be part of a finger list. This is accomplished by changing the criterion for a "good" node in a search step, so that any node that should be in a finger list is "good."

```
Insert($s_k$) (to be called after Insert has been called on $s_n, s_{n-1}, ..., s_{k+2}, s_{k+1}$)
  let curlist = $\emptyset$ // current finger list
  let $m = \infty$ // $m$ is distance of farthest node in curlist
  let $i = k + 1$
  while ($i < n$) and size(curlist) < fingerListSize do:
     add node $i$ to curlist
     $i++$
  use curlist to make the first fingerList
  $m = \max(d(s_p, s_k) \mid p \in$ curlist)
  while $i < n$ do:
     let $r = d(s_i, s_k)$
     if $\exists j \in F_{2r}(i)$ such that $d(s_j, s_k) < m$ or $d(s_j, s_k) \leq r/2$ then
       let $i$ be the smallest-ranked node $\in F_{2r}(i)$ with that property
       if $d(s_i, s_k) < m$ then
         replace node in curlist that is farthest from $s_k$ with node $i$
         use curlist to make a new finger list
         set $m = \max(d(p, s_k), p \in$ curlist)
     else let $i = \max F_{2r}(i)$
  end.
```

3.3.2 Completing the ***FindK*** design

We made some design decisions during the implementation of the k-nearest neighbor procedure (introduced in Section 2.2.7).

Recall that this algorithm performs a series of ϵ-nearest neighbor queries, iteratively increasing the value of r used as the ϵ parameter. For setting the initial value of r, the original publication proposes a solution [1] that requires knowing the expansion rate c of the data set.

To avoid a situation where correctness depends on knowing c, we propose a different solution. We sample a few points from the beginning of the ordering and for each point we check the radius of its *last* finger list. For a point s_i near the beginning of the ordering, this tells us the value of ϵ for an ϵ-nearest neighbor query around s_i that would return approximately f points. We average the results from the first few points in the ordering, giving us an estimate on the radius of a ball containing f points. We use this radius estimate as the initial value of r in *FindK*.

3.3.3 Optimizing each search step

The *Find* algorithm as described in the original paper [1] dictates that a single search step from current point $s_i, d(q, s_i) = r$ should look in the finger list $F_{2r}(i)$ for a point j such that $d(q, s_j) \leq (r/2)$ or $d(q, s_j) < d(q, s_m)$. This means that we examine points sampled from $B_{2r}(s_i)$, hoping to find a "good" point, i.e. one that is inside $B_{r/2}(q)$ or $B_{r'}(q)$, $r' = d(q, s_m)$.

We are guaranteed that all the good points are contained in $B_{2r}(s_i)$, but this may be a larger ball than necessary. Because of the "or" in the criterion that determines if a point is good, we know that the "good" radius around q is in the interval $[r/2, d(q, s_m)]$. We can tighten the $B_{2r}(s_i)$ to $B_{r'}(s_i)$, $r' = (r + \max(d(q, s_m), r/2))$, and by reducing the radius of the outer ball to snugly fit around the "good" ball, we increase the probability of finding a good next point. We implemented this optimization for every function the structure supports.

3.4 A Modified Approach

The special case of building a neighborhood graph, or solving the all nearest neighbors problem, was introduced in Section 1.3.3. Here we introduce a modified version of the KR algorithm that is designed specifically to build a neighborhood graph. Two versions are introduced: one that builds an ϵ-neighborhood graph for a pre-specified ϵ, and another that builds a k-neighborhood graph for a pre-specified k.

3.4.1 ϵ-neighborhood variant

This variant utilizes the symmetry of the ϵ-nearest neighbor relation to simplify the building and querying into one phase. As it inserts each new point, it determines the ϵ-nearest neighbors to the new point among those following it in the ordering. Every time it finds an ϵ-nearest neighbor relationship, it notifies the other point as well. That way each point "knows" about ϵ-nearest neighbors that are inserted after it was inserted.

Implementing this variant required adding two classes: `EpsNbr` and `EpsCombo`. `EpsNbr` is a simple linked-list of nodes that are within distance ϵ. `EpsCombo` builds a structure with an `EpsNbr` object associated with every node. Details of both classes are provided below.

`EpsNbr` member variables

- pointers `first` and `last` that point to the beginning and end, respectively, of the linked list
- an integer `size` that represents the length of the linked list, which is the number of ϵ-nearest neighbors that have been found so far for the associated node

`EpsNbr` member functions

- a method `getFirst()` that returns the first element in the linked list
- a method `getSize()` that returns the length of the linked list

`EpsCombo` member variables

- an integer `numnodes` that represents the number of data points that the structure contains
- an array `nodes` of the `Node` objects that comprise the structure
- an array `epsHood` of the `EpsNbr` objects that store the adjacencies
- a decimal `eps` that stores the value of ϵ

`EpsCombo` member functions

- a constructor `EpsCombo(n, f, data,` ϵ`)` that builds the structure with `n` nodes and finger list size `f` on the data set `data` with each node connected to its ϵ-nearest neighbors
- a method `getEpsNbr(i)` that returns `epsHood[i]`

3.4.2 k-neighborhood variant

This variant is similar to the ϵ variant in that it combines building and querying into a single phase. But since the k-nearest neighbor relation is not symmetric, we cannot rely on nodes inserted later to notify nodes inserted earlier about k-nearest neighbor relationships. Because of this potential asymmetry we must build the data structure twice—once forwards and once backwards. That way each node can explore the nodes after as well as before it in the ordering to find all its k-nearest neighbors.

Implementing this variant required adding two classes: `KNbr` and `KCombo`. `KNbr` maintains an updateable list of k nodes that are close to a node. `KCombo` builds a structure with a `KNbr` object associated with every node. Details of both classes are provided below.

`KNbr` member variables

- an array `nbrs` of integers identifying (by rank) the nodes that are the k-nearest neighbors seen so far for the associated node
- an array `dists` of decimals containing the distances between the associated node and each of the nodes contained in `nbrs`
- an integer `k` that is equal to k, used to set the size of the arrays `nbrs` and `dists`

`KNbr` member functions

- a method `getNbrs()` that returns the array `nbrs`
- a method `getSize()` that returns the length of the array `nbrs`, which is equal to k
- a method `add(node, distance)` that adds `node` to the set of k if `dist` is one of the k smallest seen so far

`KCombo` **member variables**

- an integer `numnodes` that represents the number of data points that the structure contains
- an array `nodes` of the `Node` objects that comprise the structure
- an array `epsHood` of the `EpsNbr` objects that store the adjacencies

`KCombo` **member functions**

- a constructor `KCombo(n, f, data, k)` that builds the structure with `n` nodes and finger list size `f` on the data set `data` with each node connected to its `k`-nearest neighbors
- a method `getKNbr(i)` that returns `kHood[i]`

3.5 Omitted Optimizations

Some optimizations that provably reduce the time complexity of the KR algorithm were not implemented. Because this research stresses the proof-of-concept of the KR algorithm, it evaluates performance in terms of number of distance calculations, which facilitates analysis. The optimizations discussed in this section improve overall time bounds, but do not affect the number of distance calculations performed.

3.5.1 Finger list shortcut pointers

The finger list access method in our implementation is simple and slow. When looking up a finger list at a certain node, it looks through the linked list of finger lists until it finds the one of the correct radius. This takes time linear in the number of finger lists per node, which means $O(\log n)$ time per finger list access.

The original paper [1] proposes something faster—finger list pointers that span nodes. Specifically, for each $j \in F_r(i)$ we add a pointer to $F_r(j)$. Also for each finger list $F_r(i)$ we add a pointer to $F_{2r}(i)$. This allows for constant finger list lookup between search steps, since with each search step, r either decreases slightly or increases to at most $2r$.

3.5.2 Use of heaps

In some situations in our implementation we maintain a dynamic set of items by continually replacing the "worst" member. For example, in the *Insert* procedure, every time a new finger list is made we throw out the node that is farthest away from the node being inserted. In our implementation these operations are carried out with simple linear passes over arrays. These can be optimized using more advanced data structures. Binary and Fibonacci heaps, for example, support find-min in $O(1)$ time and delete-min in $O(\log n)$ time [13].

Chapter 4

Testing and Analysis

This chapter reviews the testing conducted with the KR algorithm as well as the analysis of some of our results. First we describe the variables in our testing framework. Then we show and comment on some sample test results. Finally we provide some empirical analysis to explain the behaviors we observed.

4.1 Variables

This section introduces the variables involved in testing. We split these into two classes: independent and dependent. Within each class, some of the variables we introduce are inherent to the data set, while others are parameters of the near neighbor data structure.

4.1.1 Independent variables

- n: The size, in number of data points, of the data set.

- D: The extrinsic dimensionality of the data set, i.e. the dimensionality of the input space.

- *dist*: The distance metric. Recall that it must obey symmetry and the triangle inequality. In all of our tests we used Euclidean distance.

- ***d*, *c* and *f***: These are listed together because they seem to be dependent on one another. d is the intrinsic dimensionality of the data set, i.e. the dimensionality of the manifold. $d \leq D$, and for dimensionality reduction, often $d << D$. The expansion rate c is a property of the data set, presumably equal to 2^d. The finger list size f, a parameter of the data structure, is likely a function of c.

- **ϵ or k**: This is the parameter we use for building the adjacencies in the neighborhood graph.

4.1.2 Observed variables

- **clock ticks**: This is one way of measuring time performance. It counts ticks on the system clock in order to estimate the amount of time taken by certain operations.

- **distance computations**: This is our preferred way of measuring time performance. It counts the number of times the `dist` function is invoked. It is more machine-independent, and it allows us to assess performance without worrying about less-important optimizations.

- **search step success rate**: This is the ratio of successful search steps to total search steps. Success is indicated by the search step finding a new current node that is within the desired distance of the query node.

4.2 Results

To test the KR structure we calculated neighborhood graphs on the swiss roll data set. Recall that a neighborhood graph is formed by connecting each data point to its ϵ- or k-nearest neighbors. So in order to compute the neighborhood graph on n data points, the structure must perform n insertions to build itself, and then n queries to determine the adjacencies.

Performance was measured in terms of distance calculations. The time bounds given in Section 2.3 were in terms of standard operations, but we did not implement

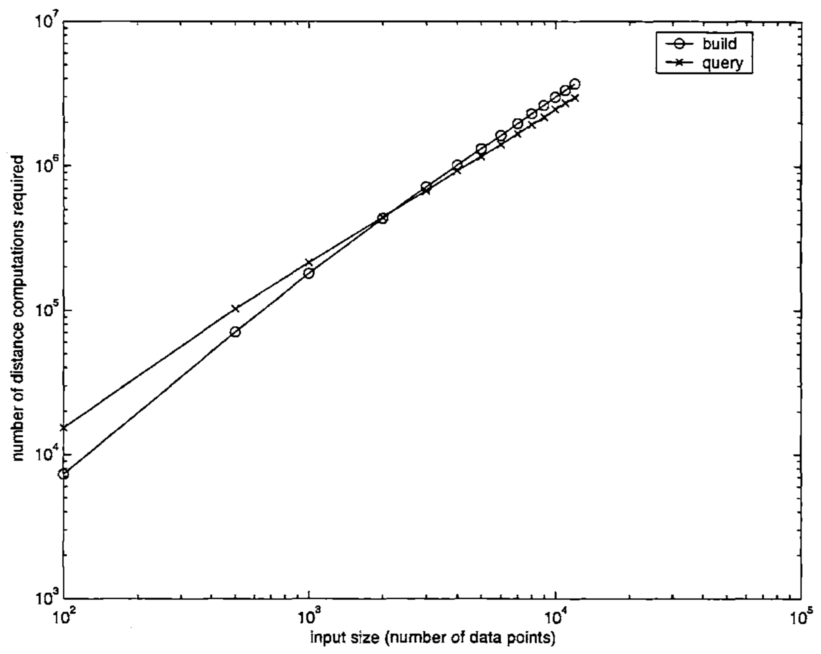

Figure 4-1: Processing time on the swiss roll data set with respect to input size. Finger list size = 3.

the optimizations necessary to achieve those. The theoretical bounds are the same for number of distance calculations as for standard operations.

4.2.1 Performance v. input size

As we see in Figure 4-1, the running time appears to be $O(n \log n)$ for the insertions as well as the queries. The extra factor of n appears because the times are aggregate over all of the insertions and all of the queries, respectively.

Notice that Figure 4-1 evaluates performance with finger list size of 3. This is very different from the $24c^3$ value derived in Section 2.2.1, which for the swiss data is equal to $24 * (2^2)^3 = 1536$. We show the plot for $f = 3$ because that value gave the best results (see Figure 4-2). This efficiency for small finger list sizes is a central discovery of this project; it is analyzed in Section 4.3.

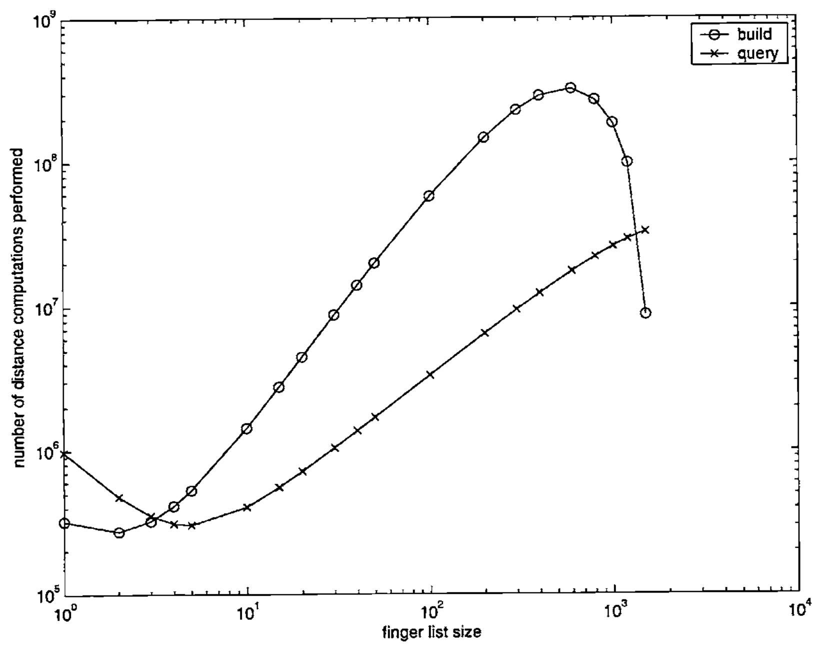

Figure 4-2: Processing time on the swiss roll data set with respect to finger list size. $n = 1600$, $\epsilon = 3.953$

4.2.2 Performance v. finger list size

Figure 4-2 considers how performance depends on the finger list size f. We see that the optimal f value is 2 for inserts (see the "build" curve, which represents the sum of all n inserts) and 5 for queries. The sum of build and query time is optimized at $f = 3$. The worst performance appears for medium-sized finger lists, and performance improves as the finger list size approaches n, which is 1600 in this case. This is probably because with finger lists of approximately n, the algorithm becomes brute-force, and search steps are very successful but the running time is quadratic.

Our test results, which show optimal performance for very small finger lists ($f = 3$) in the case of the swiss roll, require us to rethink the analysis that led to the theoretical suggestion of $f = 24c^3$.

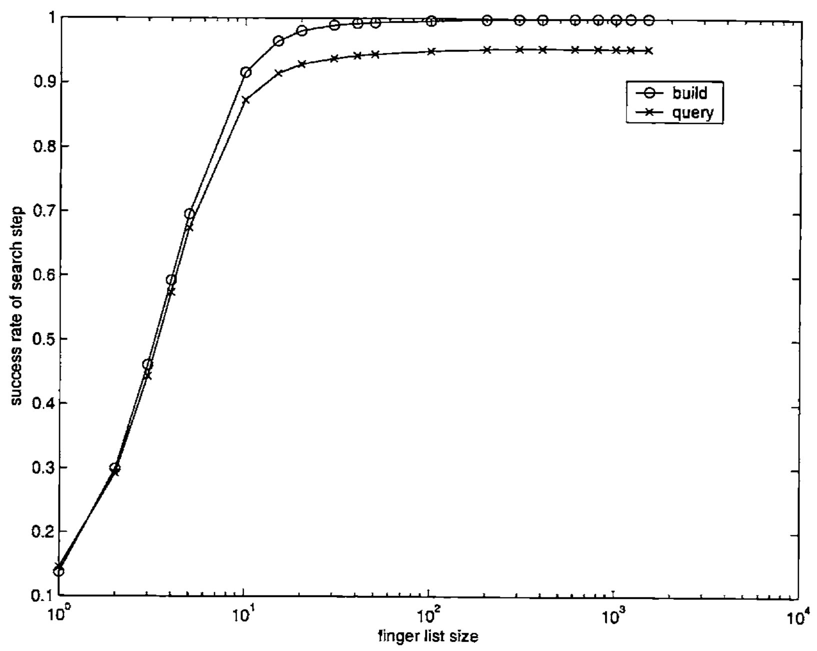

Figure 4-3: Search step success rate on the swiss roll data set with respect to finger list size. $n = 1600$, $\epsilon = 3.953$

4.3 Analysis

4.3.1 Why small finger lists work well

In examining the significance of finger list size, we first consider the benefit of small finger lists. Clearly a smaller finger list means each search step takes less time. Also the space complexity of the data structure is $O(nf \log n)$ (n nodes, each with $O(\log n)$ finger lists of length f), so for large data sets large finger lists make the structure difficult to store.

Now we examine the case for having large finger lists. The reason f was originally set to $24c^3$ was to give a good probability of success for each search step. But the analysis used to reach the value $24c^3$ appears conservative; as Figure 4-3 shows, very high search step success rates were found for quite low values of the finger list size f. We also see a region where the success rate scales proportionally to $\log f$, but there is little benefit to increasing the finger list size beyond $f = 10$.

One explanation for the optimality of small finger lists in our tests could lie in the nature of the neighborhood graph problem, for which we optimize the running time on exactly n inserts and n queries. In many other near neighbor applications we perform *one* build phase (of n inserts) followed by an unbounded number of queries. In those situations we prefer to optimize the query time; in our case, we give equal weight to inserts and queries. The best f for the build phase is less than that for the query phase (probably because *Insert* scales with f^2), and our overall optimal f value of 3 reflects the blend of the two individual optimal f values.

Another explanation considers the data sets we used for testing. Both the swiss roll and faces data sets consist of points sampled uniformly from a pre-defined manifold. This allows us to use a new model, one which assumes a uniform distribution of data points on the manifold, to re-analyze a search step.

Rethinking the success criterion

We re-examine the success criterion. In a search step starting at a current node that is distance r from the query node q, we are looking for a node within distance r' of q, where $(r/2 \leq r' < r)$. This means the success criterion is always somewhere in the spectrum between the "difficult" extreme of $r/2$ and the "easy" extreme of r. We consider these criteria in two separate cases, using a new context for both that is simpler than the one which led to the $24c^3$ value.

Given a point p at distance r from the query point q, we choose points randomly from a ball around p and assess the probability that at least one of them is within the desired distance of q. We consider the two extreme cases from the previous paragraph, and use two-dimensional examples to assist in the descriptions.

Case 1: $r' = r/2$

Figure 4-4 shows an inner circle of radius $r/2$ and an outer circle of radius $3r/2$. The radii differ by a factor of 3, so the areas of the two disks differ by a factor of $3^2 = 9$. If we randomly choose one point on the larger disk, with 8/9 probability it will not be on the smaller disk. So our probability of success with three randomly chosen points is:

$$1 - (8/9)^3 > .29$$

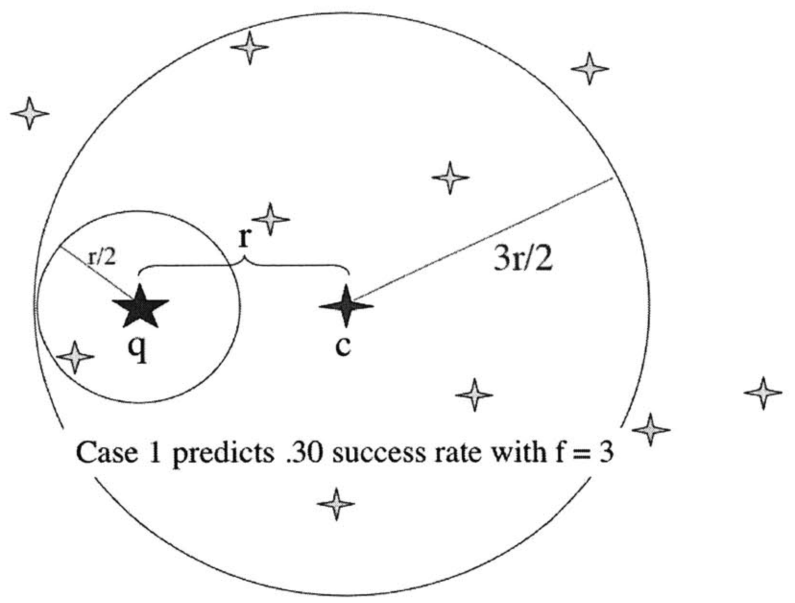

Figure 4-4: Sampling from $B_{3r/2}(c)$ to find something in $B_{r/2}(q)$.

Case 2: $r' = r$

Figure 4-5 shows an inner circle of radius r and an outer circle of radius $2r$. The radii differ by a factor of 2, so the areas of the two disks differ by a factor of $2^2 = 4$. If we randomly choose one point on the larger disk, with 3/4 probability it will not be on the smaller disk. So our probability of success with three randomly chosen points is:

$$1 - (3/4)^3 > .57$$

Blending the two cases

If we average the success rates of case 1 and case 2 for $f = 3$ we get approximately .43, which roughly matches the success rate observed for $f = 3$ in Figure 4-3. Thus we empirically verify our guess that the KR algorithm uses a blend of the two cases when defining its search success.

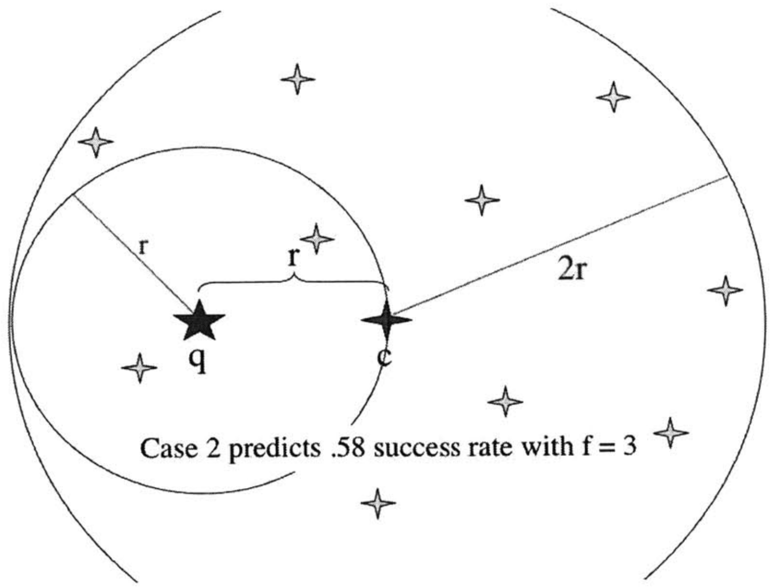

Figure 4-5: Sampling from $B_{2r}(c)$ to find something in $B_r(q)$.

A more general case

We can generalize our analysis to points uniformly distributed on d-dimensional manifolds. Given the ratio

$$x \equiv \frac{r}{r'} \text{ , } 1 \leq x \leq 2,$$

we know that the radius of the inner ball is r/x and the radius of the outer ball is $\frac{(1+x)r}{x}$. The ratio of outer radius to inner radius is $(1+x)$, so the ratio of the size of the outer ball to the size of the inner ball is $(1+x)^d$. Then

$$p(\text{success with one sample}) = (\frac{1}{1+x})^d, \text{ and}$$

$$p(\text{success with } f \text{ samples}) = 1 - (1 - (\frac{1}{1+x})^d)^f \text{ .}$$

We can verify the answers to Case 1 and Case 2 by computing with $d = 2, f = 3$, and $x = 2$ and 1, respectively.

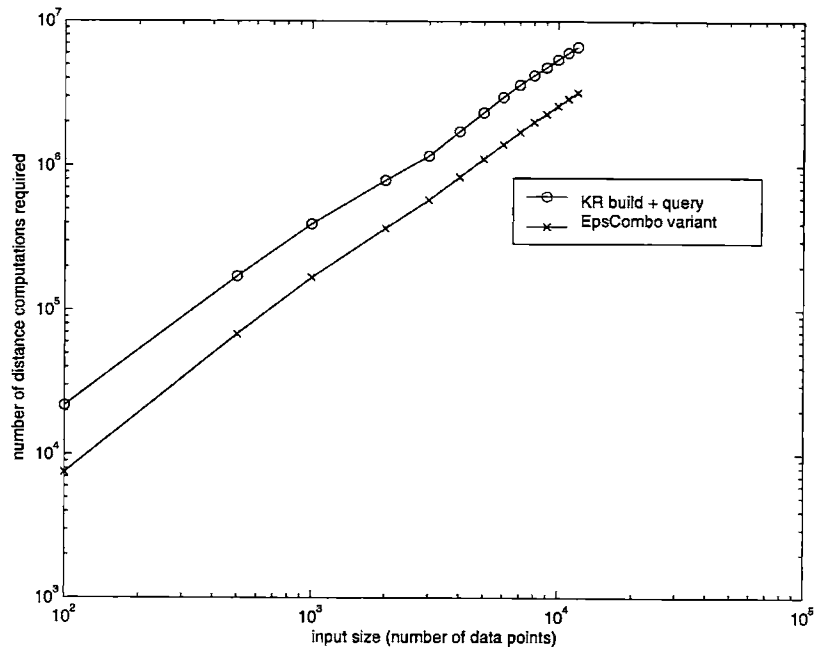

Figure 4-6: Processing time on the swiss roll data set with respect to input size. `EpsCombo` outperforms the regular KR structure. Finger list size = 3.

4.3.2 Assessment of my variants

Figure 4-6 shows the performance benefit of using my `EpsCombo` variant of the standard KR structure. `EpsCombo` uses approximately half as many distance calculations as *FindRange* when building neighborhood graphs. Though this doesn't offer asymptotic benefits, a factor of two could be useful for large data sets. My `KCombo` variant did not demonstrate a significant performance advantage over the standard *FindK* function.

Chapter 5

Conclusions and Further Research

5.1 Conclusions

The KR algorithm proved itself to be an efficient way of computing neighborhood graphs on low-dimensional data in high-dimensional embeddings. It performs searches correctly and quickly. My variants of the KR algorithm were marginally faster than the originals at neighborhood graph construction, but they require the prior specification of the ϵ or k parameter, and in many applications this parameter is not known beforehand.

The KR structure exceeded performance expectations by working best with very small finger lists. On the swiss roll data set, $f = 3$ resulted in the best performance. This is a very different result from the $24c^3$ value suggested in the original publication [1]. We have some intuitive feel for why this was the case. It appears that the original analysis (see Section 2.2.1) was conservative, and the swiss roll data set can be modeled as a two-dimensional surface containing points distributed uniformly at random. In preliminary testing, the optimality of small finger lists has extended to data sets of high extrinsic dimensionality (e.g. the face data set shown in Figure 1-2).

5.2 Further Research

5.2.1 Analysis

Theoretical analysis is still needed to prove the benefit of very small finger lists. It is plausible that the benefit we observed is a product of our testing limitations. The data sets we tested on were all uniformly sampled; this probably elevated the success rate of search steps. Data gathered in real-world contexts does not provide this uniformity. Perhaps along with more analysis of finger list size we will be able to determine the optimal list size as a function of some properties (e.g. n, c) of the data set.

5.2.2 Data sets

In order to truly prove its utility, the KR algorithm should be used to allow ISOMAP to operate on data sets that were prohibitively large before. Finding large real-world data sets for dimensionality reduction has been difficult. Some good candidates are in the MNIST [14] database, which contains an archive of images of various handwritten characters. More "toy" dimensionality reduction data sets like the swiss roll can also be generated by random functions, but the results are less gratifying, and could produce artificial performance gains due to properties such as uniform distribution of data points.

5.2.3 Optimizations

This research invites many continuations and extensions. Our implementation omitted several useful optimizations (see Section 3.5) that can reduce real-world running times. Also if the dynamic version (see Section 2.2.8) of the KR algorithm were implemented, the structure would become accessible to a new set of applications.

5.2.4 Comparison to other near neighbor schemes

Before it can be claimed as the "best choice" for purposes such as dimensionality reduction, the KR structure should be compared to other near neighbor schemes. Even though other schemes (such as k-d trees [15]) do not promise competitive theoretical time bounds on low-dimensional data embedded in high-dimensional spaces, many seem to work well in practice.

Appendix A

Source Code

This appendix shows all the C++ code for my implementation of the KR algorithm.

A.1 Header file

```
#include <math.h>
#include "matlab.hpp"
#include <stdlib.h>
#include <iostream.h>
#include <time.h>
#include <stdio.h>

class Point{
  int d;
  double* coords;
 public:
  Point(int dim, double embedding[]);
  Point();
  ~Point();
  int getD();
  double getCoord(int i);
  double dist(Point* p1);
  static int usage;
  static void resetUsage();
  static int getUsage();
  static void incrUsage();
};
class FingerList{
  int length;
  int* fingers;
  FingerList** shortcuts;
  FingerList* nextList;
  FingerList* prevList;
  double r;

 public:
  FingerList(double maxr, int size, int fings[]);
  ~FingerList();
  double getR();
  FingerList* getNext();
  void setNext(FingerList* f);
  FingerList* getPrev();
  void setPrev(FingerList* f);
  int getListElt(int i);  // doesn't check if i is in bounds

  // looks in this fingerlist for the node j with the
  // smallest index s.t. (d(j,q) <= r).  if there is no
  // such j, then it returns the maximum j seen.
  int findJ(Point* q, double r, Point* data[]);
  int getSize();

  // returns the index of fList containing the
  // index of the node most distant from Point q.
  int farthestIndex(int q, Point* data[]);

  // returns a random int in the closed interval of params
  static int randRange(int lowest_number, int highest_number);
};
class Node{  // basically just a linked list of FingerList items
  FingerList* firstList;
  FingerList* lastList;
 public:
  Node(FingerList* firstL);
  ~Node();
  void addFingerList(FingerList* f);

  // returns the finger list with the largest radius
  // that is less than or equal to r
  FingerList* getFingerList(double r);

  FingerList* getFirstList();
  FingerList* getLastList();
};
class KNbr{
  int* nbrs;
  double* dists;
  int k;
  int size;
  double maxR;
  int worstInd;
 public:
  KNbr(int kVal);
  ~KNbr();
  int* getNbrs(); // returns nbrs array directly
  int getSize();
  bool add(int indx, double dist);
};
```

```
class EpsNbr{
  FingerList* first;
  FingerList* last;
  int size;
 public:
  EpsNbr();
  ~EpsNbr();
  FingerList* getFirst();
  bool add(int indx);  // doesn't check or verify distance
  int getSize();
};
class NNStruct{
  int numnodes, fingListSize, numSteps, numGoodSteps;
  Node** nodes;
 public:
  NNStruct(int n, int listSize);
  ~NNStruct();
  Node* getNode(int k);
  int getSteps();
  int getGoodSteps();
  bool resetSteps();
  double getKBallR();
  double augment(double rad, int* curBall, int sizeBall, Point** dat);
  static double median(double* arr, int len);

  // insert a point at the beginning of the structure
  void insert(int q, Point** data);

  // find single nearest-neighbor
  int find(Point* p, Point** data);
  int findRange(Point* p, Point** data, double range, int** output);
  int findK(Point* p, Point** data, int k, int** output);
  int findKOld(Point* p, Point** data, int k, int** output);
};
class KCombo{
  int n, k, numSteps, numGoodSteps;
  KNbr** kHood;
 public:
  KCombo(int numnodes, int listSize, Point** data0, int kVal);
  ~KCombo();
  KNbr* getKNbr(int k);
  int getSteps();
  int getGoodSteps();
  bool resetSteps();
};
class EpsCombo{
  int numnodes, numSteps, numGoodSteps;
  Node** nodes;
  EpsNbr** epsHood;
  double eps;
 public:
  EpsCombo(int n, int listSize, Point** data, double epsVal);
  ~EpsCombo();
  Node* getNode(int k);
  EpsNbr* getEpsNbr(int k);
  int getSteps();
  int getGoodSteps();
  bool resetSteps();
};
class OldNN{
  double** distances;
  int numnodes;
 public:
  OldNN(int n, Point** data);
  ~OldNN();
  int singleNN(int indx, Point** data);  // NN to pt IN DATA SET
  int epsilonNN(int indx, Point** data, double range, int** output);
  int kNN(int indx, Point** data, int k, int** output);
};

int compare(int nsize, bool epsmode, double param, int fListLen,
            Point** points, Point** kPoints, int* derand, int kRand);
int compare2(int nsize, bool epsmode, double param, int fListLen,
             Point** points, Point** kPoints, int* derand, int kRand);
int loadData(bool swiss, int nsize, Point*** points,
             Point*** kPoints, int** derand);

#endif
```

A.2 Class Point

```
Point::Point(int dim, double embedding[]){
  d = dim;
  coords = new double[d];
  int i;
  for (i=0;i<d;i++)
    coords[i] = embedding[i];
}

Point::~Point(){
  delete[] coords;
}

int Point::getD(){
  return d;
}

double Point::getCoord(int i){ // requires in-bounds i
  return coords[i];
}

double Point::dist(Point* p){  // requires matched dimensionality
  double dist=0;
  int i;
  for (i=0;i<d;i++)
```

```
    dist = dist + pow(p->getCoord(i) - coords[i], (double)2);
  incrUsage();
  return sqrt(dist);
}

int Point::usage = 0;

void Point::resetUsage(){
  usage = 0;
  return;
}

int Point::getUsage(){
  return usage;
}

void Point::incrUsage(){
  usage++;
  return;
}
```

A.3 Class `FingerList`

```
FingerList::FingerList(double maxr, int size, int fings[]){
  r = maxr;
  nextList = NULL;
  prevList = NULL;
  fingers = new int[size];
  length = size;

  for (int i=0;i<length;i++)
    fingers[i] = fings[i];
}

FingerList::~FingerList(){
  delete[] fingers;
}

double FingerList::getR(){
  return r;
}

FingerList* FingerList::getNext(){
  return nextList;
}

void FingerList::setNext(FingerList* f){
  nextList = f;
}
```

```
FingerList* FingerList::getPrev(){
  return prevList;
}

void FingerList::setPrev(FingerList* f){
  prevList = f;
}

int FingerList::getListElt(int i){
  return fingers[i];
}

int FingerList::getSize(){
  return length;
}

int FingerList::findJ(Point* q, double rad, Point* data[]){
  int maxNode=0;
  int minGoodNode=0;
  bool foundGoodNode=false;
  int curNode;
  for (int i=0; i<length; i++){
    curNode = fingers[i];
    if ( (q->dist(data[curNode]) < rad) &&
         ((curNode < minGoodNode) || !foundGoodNode) ){
      minGoodNode = curNode;
      foundGoodNode = true;
    }
    if (curNode > maxNode) { maxNode = curNode; }
  }
  if (foundGoodNode) { return minGoodNode; }
  else { return maxNode; }
}

// see random_range() at http://www.cpp-home.com/tutorial.php?319_1
int FingerList::randRange(int lowest_number, int highest_number){
  int range = highest_number - lowest_number + 1;
  int i = random()%RAND_MAX;
  return lowest_number + int(range * i/(RAND_MAX + 1.0));
}
```

A.4 Class Node

```
Node::Node(FingerList* firstL){
  firstList = firstL;
  lastList = firstList;
}

Node::~Node(){
  FingerList* f = getFirstList();
  FingerList* f2;
  while (f->getNext() != NULL){
    f2 = f->getNext();
    delete f;
    f = f2;
  }
  delete f;
}

void Node::addFingerList(FingerList* f){
  lastList->setNext(f);
  f->setPrev(lastList);
  lastList = lastList->getNext();
  return;
}

FingerList* Node::getFingerList(double r){
  FingerList* f = getFirstList();
  while (f->getNext() != NULL){
    if (f->getR() <= r)
      return f;
    f = f->getNext();
  }
  return f;  // return a suboptimal list; better than nothing.
}

FingerList* Node::getFirstList(){
  return firstList;
}

FingerList* Node::getLastList(){
  return lastList;
}
```

A.5 Class NNStruct

```
NNStruct::NNStruct(int n, int listSize){
  numnodes = n;
  fingListSize = listSize;
  nodes = new Node*[numnodes];
  for (int i=0;i<numnodes;i++)
    nodes[i] = NULL;
  numSteps = 0;
  numGoodSteps = 0;
}

NNStruct::~NNStruct(){
  for (int i=0;i<numnodes;i++)
    if (nodes[i] != NULL) { delete nodes[i]; }
  delete[] nodes;
}
```

```
int NNStruct::getSteps(){
  return numSteps;
}

int NNStruct::getGoodSteps(){
  return numGoodSteps;
}

bool NNStruct::resetSteps(){
  numSteps = 0;
  numGoodSteps = 0;
  return true;
}

Node* NNStruct::getNode(int k){
  return nodes[k];
}

void NNStruct::insert(int q, Point** data){
  int* curList = new int[fingListSize];
  double* curDists = new double[fingListSize];
  int curListSize = 0; // indicates the next-open index in curList
  double curR;
  int curListWorstIndex = 0;
  int curPos = q;

  // loop to discover the first (and possibly only) finger list
  while (curListSize < fingListSize){
    curPos++;
    curList[curListSize] = curPos;
    curR = data[q]->dist(data[curPos]);
    curDists[curListSize] = curR;
    if (curR > curDists[curListWorstIndex]){
      curListWorstIndex = curListSize;
      curDists[curListWorstIndex] = curR;
    }
    curListSize++;
    if (curPos > numnodes-2){
      nodes[q] = new Node
        (new FingerList(curDists[curListWorstIndex], curListSize, curList));
      return;
    }
  }

  // if we reach this point, we know curListSize == fingListSize,
  // so we construct the new node and its first fingerList:
  nodes[q] = new Node
    (new FingerList(curDists[curListWorstIndex], fingListSize, curList));

  double queryR;

  while (curPos < numnodes-1){
    if (curDists[curListWorstIndex] < (curR/(double)2))
      queryR = (curR/(double)2);
      else queryR = curDists[curListWorstIndex];
      curPos = nodes[curPos]->getFingerList(queryR+curR)->
        findJ(data[q], queryR, data);
      curR = data[q]->dist(data[curPos]);
      numSteps++;
      if (curR <= queryR)
        numGoodSteps++;
      if (curR < curDists[curListWorstIndex]){
        // substitute to form the new list
        curList[curListWorstIndex] = curPos;
        curDists[curListWorstIndex] = curR;
        // find the r for the new list
        for (int i=0;i<fingListSize;i++){
          if (curDists[i] > curDists[curListWorstIndex])
            curListWorstIndex = i;
        }
        nodes[q]->addFingerList
          (new FingerList(curDists[curListWorstIndex], fingListSize, curList));
      }
    }
    delete[] curList;
    delete[] curDists;
    return;
  }

  int NNStruct::find(Point* p, Point** data){
    int curNode = 0;
    int minNode = 0;
    double curDist = p->dist(data[curNode]);
    double minDist = curDist;
    double queryR;

    while (curNode < numnodes-1){
      if (minDist < (curDist/(double)2))
        queryR = (curDist/(double)2);
      else queryR = minDist;
      curNode = nodes[curNode]->getFingerList
        (curDist+queryR)->findJ(p, queryR, data);
      curDist = p->dist(data[curNode]);
      numSteps++;
      if (curDist <= queryR)
        numGoodSteps++;
      if (curDist < minDist){
        minDist = curDist;
        minNode = curNode;
      }
    }
    return minNode;
  }

  int NNStruct::findRange(Point* p, Point** data, double range, int** output){
    int curNode = 0;
    double curDist = p->dist(data[curNode]);
    double minDist = curDist;
```

```
double queryR;
int numOut = 0;
FingerList* firstOut=NULL;
FingerList* lastOut=NULL;
int curOut[1];

while (curNode < numnodes-1){
  // establish variables based on current search position:
  if (curDist < minDist){ minDist = curDist; }
  if (minDist < range) { minDist = range; }

  // add curNode to output if needed:
  if (curDist <= range){
    numOut++;
    curOut[0] = curNode;
    if (firstOut == NULL){
      firstOut = new FingerList(curDist, 1, curOut);
      lastOut = firstOut;
    }
    else {
      lastOut->setNext(new FingerList(curDist, 1, curOut));
      lastOut = lastOut->getNext();
    }
  }
  // jump to the next node:
  if (minDist < (curDist/(double)2))
    queryR = (curDist/(double)2);
  else { queryR = minDist; }
  curNode = nodes[curNode]->
    getFingerList(curDist+queryR)->findJ(p, queryR, data);
  curDist = p->dist(data[curNode]);
  numSteps++;
  if (curDist <= queryR)
    numGoodSteps++;
}
// check the last node before returning:
if (curDist <= range){
  numOut++;
  curOut[0] = curNode;
  if (firstOut == NULL) { firstOut = new FingerList(curDist, 1, curOut); }
  else { lastOut->setNext(new FingerList(curDist, 1, curOut)); }
}

// dump all the answers into the output array, and delete the FingerLists
if (numOut > 0){
  *output = new int[numOut];
  FingerList* fl = firstOut;
  FingerList* tmp;
  for (int i=0;i<numOut;i++){
    (*output)[i] = fl->getListElt(0);
    tmp = fl;
    fl = fl->getNext();
    delete tmp;
  }
```

```
    if (fl != NULL) { cout << "error making array" << endl; }
  }
  else { *output = NULL; }
  return numOut;
}

int NNStruct::findK(Point* q, Point** data, int k, int** output){
  if (k > numnodes) {
    cout << "invalid k-n-n query-- k is too high." << endl;
    *output = NULL;
    return 0;
  }
  double r = getKBallR();
  if (r <= 0){
    cout << "getKBallR() is broken, using r=1.0" << endl;
    r = 1.0;
  }
  int* ball;
  int ballSize = findRange(q,data,r,&ball);
  while (ballSize < 1){
    cout << "rescuing the ball!" << endl;
    r = 2.0*r;
    delete[] ball;
    ballSize = findRange(q,data,r,&ball);
  }
  while (ballSize < k){
    r = augment(r,ball,ballSize,data);
    delete[] ball;
    ballSize = findRange(q,data,r,&ball);
  }
  KNbr* kn = new KNbr(k);
  for (int i=0;i<ballSize;i++){
    kn->add(ball[i],q->dist(data[ball[i]]));
  }
  delete[] ball;
  *output = new int[k];
  int* nbrs = kn->getNbrs();
  for (int i=0;i<k;i++){
    (*output)[i] = nbrs[i];
  }
  delete kn;
  return k;
}

double NNStruct::augment(double rad, int* curBall, int sizeBall, Point** dat){
  double* rads = new double[sizeBall];
  for (int i=0;i<sizeBall;i++){
    rads[i] = DBL_MAX;        // return infinity by default
    double minR = DBL_MAX;   // initial value for 'best' choice
    if (curBall[i] == numnodes-1)   // the last node has no finger lists,
      continue;                    // so we can't pursue it.
    FingerList* fl = nodes[curBall[i]]->getFingerList(2.0*rad);
    if ((fl->getPrev() != NULL) && (fl->getR() < (2.0*rad))){
      fl = fl->getPrev();
    }
    for (int j=0;j<fl->getSize();j++){
      double curR = dat[curBall[i]]->dist(dat[fl->getListElt(j)]);
      if ((curR > (2.0*rad)) && (curR < minR)){
        rads[i] = curR;
        minR = curR;
      }
    }
  }
  double out = median(rads, sizeBall);
  delete[] rads;
  return out;
}

double NNStruct::median(double* arr, int len){
  for (int i=0;i<len;i++){
    double candidate = arr[i];
    int numSmaller = 0;
    int numLarger = 0;
    for (int j=0;j<len;j++){
      if (arr[j] < candidate)
        numSmaller++;
      if (arr[j] > candidate)
        numLarger++;
    }
    if (abs(numSmaller-numLarger) < 2)
      return candidate;
  }
}

double NNStruct::getKBallR(){
  if (nodes[0] == NULL){
    cout << "struct not fully built-- getKBallR is returning 0.0" << endl;
    return 0.0;
  }
  int numSamples;
  if (numnodes > 150)
    numSamples = (int) log((double)numnodes);
  else if (numnodes < 5)
    numSamples = numnodes;
  else numSamples = 5;
  double output = 0;
  for (int i=0; i<numSamples; i++)
    output = output + nodes[i]->getLastList()->getR();
  return output/((double)numSamples);
}

int NNStruct::findKOld(Point* p, Point** data, int k, int** output){
  if (k > numnodes) {
    cout << "invalid k-n-n query-- k is too high." << endl;
    *output = NULL;
    return 0;
  }
  *output = new int[k];
  double curR;
  double curListMaxR = 0;
  int curListWorstIndex;

  // start by setting output to the first k nodes.
  for (int curPos=0;curPos<k;curPos++){
    (*output)[curPos] = curPos;
    curR = p->dist(data[curPos]);
    if (curR > curListMaxR){
      curListMaxR = curR;
      curListWorstIndex = curPos;
    }
  }
  int curPos = k-1;
  curR = p->dist(data[curPos]);
  double queryR;

  while (curPos < numnodes-1){
    if (curListMaxR < (curR/(double)2))
      queryR = (curR/(double)2);
    else queryR = curListMaxR;
    curPos = nodes[curPos]->getFingerList(queryR+curR)->
      findJ(p, queryR, data);
    curR = p->dist(data[curPos]);
    numSteps++;
    if (curR <= queryR)
      numGoodSteps++;
    if (curR < curListMaxR){
      // substitute to improve the list
      (*output)[curListWorstIndex] = curPos;
      // find the max-r for the altered list
      curListMaxR = curR;
      for (int i=0;i<k;i++){
        double tempR = p->dist(data[(*output)[i]]);
        if (tempR > curListMaxR){
          curListMaxR = tempR;
          curListWorstIndex = i;
        }
      }
    }
  }
  return k;
}
```

A.6 Classes EpsNbr and EpsCombo

```
EpsNbr::EpsNbr(){
  first = NULL;
  last = NULL;
  size = 0;
}
```

```
EpsNbr::~EpsNbr(){}  // getNbrs(int) deallocates the fingerlists.

FingerList* EpsNbr::getFirst(){
  return first;
}

bool EpsNbr::add(int indx){
  int tmp[] = { indx };
  if (first == NULL){
    first = new FingerList(0,1,tmp);
    last = first;
    size++;
    return true;
  }
  last->setNext(new FingerList(0,1,tmp));
  last = last->getNext();
  size++;
  return true;
}

int EpsNbr::getSize(){
  return size;
}

EpsCombo::EpsCombo(int n, int listSize, Point** data, double epsVal){
  numnodes = n;
  numSteps=0;
  numGoodSteps=0;
  nodes = new Node*[n-1];
  epsHood = new EpsNbr*[n];
  epsHood[n-1] = new EpsNbr();
  epsHood[n-1]->add(n-1);
  eps = epsVal;
  for (int q=n-2; q>-1; q--){
    epsHood[q] = new EpsNbr();
    epsHood[q]->add(q);
    int* curList = new int[listSize];
    double* curDists = new double[listSize];
    int curListSize = 0; // indicates the next-open index in curList
    double curR;
    int worstIndex=0;
    int curPos = q;
    bool breakFlag = false;

    // loop to discover the first (and possibly only) finger list
    while (curListSize < listSize){
      curPos++;
      curList[curListSize] = curPos;
      curR = data[q]->dist(data[curPos]);
      curDists[curListSize] = curR;
      if (curR <= eps){
        epsHood[q]->add(curPos);
        epsHood[curPos]->add(q);
    }
    if (curR > curDists[worstIndex])
      worstIndex = curListSize;
    curListSize++;
    if (curPos > n−2){
      breakFlag = true;
      break;
    }
  }

  // construct the node and its first (largest radius) finger list.
  nodes[q] = new Node(new FingerList(curDists[worstIndex], curListSize, curList));

  // stop constructing this node if the first fingerlist was the only one.
  if (breakFlag){
    delete[] curList;
    delete[] curDists;
    continue;
  }

  // loop to construct the remaining finger lists.
  double queryR;
  while (curPos < n−1){
    if (curDists[worstIndex] < (curR/(double)2))
      queryR = (curR/(double)2);
    else queryR = curDists[worstIndex];
    if (queryR < eps) { queryR = eps; }
    curPos = nodes[curPos]−>getFingerList(queryR+curR)−>
      findJ(data[q], queryR, data);
    curR = data[q]−>dist(data[curPos]);
    numSteps++;
    if (curR <= queryR)
      numGoodSteps++;
    if (curR <= eps){
      epsHood[q]−>add(curPos);
      epsHood[curPos]−>add(q);
    }
    if (curR < curDists[worstIndex]){
      // substitute to form the new list
      curList[worstIndex] = curPos;
      curDists[worstIndex] = curR;
      // find the r for the new list
      for (int i=0;i<listSize;i++){
        double tempR = curDists[i];
        if (tempR > curDists[worstIndex])
          worstIndex = i;
      }
      nodes[q]−>addFingerList
        (new FingerList(curDists[worstIndex], listSize, curList));
    }
  }
  delete[] curList;
  delete[] curDists;
}
```

```
  return;
}

EpsCombo::~EpsCombo(){
  for (int i=0;i<numnodes-1;i++){
    delete nodes[i];
    delete epsHood[i];
  }
  delete epsHood[numnodes-1];
  delete[] nodes;
  delete[] epsHood;
}

int EpsCombo::getSteps(){
  return numSteps;
}

int EpsCombo::getGoodSteps(){
  return numGoodSteps;
}

bool EpsCombo::resetSteps(){
  numSteps = 0;
  numGoodSteps = 0;
  return true;
}

EpsNbr* EpsCombo::getEpsNbr(int k){
  return epsHood[k];
}
```

A.7 Classes KNbr and KCombo

```
KNbr::KNbr(int kVal){
  k = kVal;
  nbrs = new int[k];
  dists = new double[k];
  size = 0;  // next open slot in nbrs array
  maxR = 0;
  worstInd = 0;
}

KNbr::~KNbr(){
  delete[] nbrs;
  delete[] dists;
}

int* KNbr::getNbrs(){
  return nbrs;
}
```

```
int KNbr::getSize(){
  return size;
}

bool KNbr::add(int indx, double dist){
  if (size < k){
    nbrs[size] = indx;
    dists[size] = dist;
    if (dist > maxR){
      maxR = dist;
      worstInd = size;
    }
    size++;
    return true;
  }
  if (dist < maxR){
    nbrs[worstInd] = indx;
    dists[worstInd] = dist;
    maxR = dist;
    for (int i=0; i<k; i++){
      double temp = dists[i];
      if (temp > maxR){
        maxR = temp;
        worstInd = i;
      }
    }
    return true;
  }
  return false;
}

KCombo::KCombo(int numnodes, int listSize, Point** data0, int kVal){
  n = numnodes;
  numSteps=0;
  numGoodSteps=0;
  k = kVal;
  // initialize KNbr output objects
  kHood = new KNbr*[n];
  for (int i=0;i<n;i++){
    kHood[i] = new KNbr(k);
    kHood[i]->add(i, (double)0);
  }
  // construct backward array of Points for second pass
  Point** data1 = new Point*[n];
  for (int i=0;i<n;i++){
    data1[i] = data0[n-i-1];
  }

  for (int pass=0;pass<2;pass++){
    Node** nodes = new Node*[n-1];
    Point** dat;
    if (pass == 0) { dat = data0; }
    else { dat = data1; }
```

```
// loop that builds structure on "dat" into "kH"
for (int q=n−2; q>−1; q−−){
  int* curList = new int[listSize];
  double* curDists = new double[listSize];
  double curR;
  int worstIndex=0;
  int curPos = q;

  // loop to discover the first (and possibly only) finger list for this q
  int curListSize = 0;                    // next open index in curList
  bool breakFlag = false;
  while (curListSize < listSize){
    curPos++;
    curList[curListSize] = curPos;
    curR = dat[q]−>dist(dat[curPos]);
    curDists[curListSize] = curR;
    if (curR > curDists[worstIndex])
      worstIndex = curListSize;
    curListSize++;
    if (curPos > n−2){
      if (pass == 0)
        for (int i=0; i<curListSize; i++)
          kHood[q]−>add(curList[i],curDists[i]);
      else
        for (int i=0; i<curListSize; i++)
          kHood[n−q−1]−>add(n−curList[i]−1,curDists[i]);
      breakFlag = true;
      break;
    }
  }

  // construct the new node and its first fingerList:
  nodes[q] = new Node
    (new FingerList(curDists[worstIndex], curListSize, curList));

  // stop working on this node if first fingerlist was only one.
  if (breakFlag) {
    delete[] curList;
    delete[] curDists;
    continue;
  }

  // loop to discover the rest of the finger lists.
  double queryR;
  while (curPos < n−1){
    if (curDists[worstIndex] < (curR/(double)2))
      queryR = (curR/(double)2);
    else queryR = curDists[worstIndex];
    curPos = nodes[curPos]−>getFingerList(queryR+curR)−>
      findJ(dat[q], queryR, dat);
    curR = dat[q]−>dist(dat[curPos]);
    numSteps++;
    if (curR <= queryR)
      numGoodSteps++;
```

```
        if (curR < curDists[worstIndex]){
          // substitute to form the new list
          curList[worstIndex] = curPos;
          curDists[worstIndex] = curR;
          // find the r for the new list
          for (int i=0;i<listSize;i++)
            if (curDists[i] > curDists[worstIndex])
              worstIndex = i;
          // make new list, add it to node
          nodes[q]->addFingerList
            (new FingerList(curDists[worstIndex], listSize, curList));
        }
      }
      // update the KNbr object for q
      if (pass == 0)
        for (int i=0; i<listSize; i++)
          kHood[q]->add(curList[i],curDists[i]);
      else
        for (int i=0; i<listSize; i++){
          kHood[n-q-1]->add(n-curList[i]-1,curDists[i]);
        }
      delete[] curList;
      delete[] curDists;
    }

    // throw out the nodes array at the end of each pass
    for (int i=0;i<n-1;i++)
      delete nodes[i];
    delete[] nodes;
  }

  // deallocate backwards array, retain input (data0) and output(kHood0).
  delete[] data1;

  return;
}

KCombo::~KCombo(){
  for (int i=0;i<n;i++){
    delete kHood[i];
  }
  delete[] kHood;
}

int KCombo::getSteps(){
  return numSteps;
}

int KCombo::getGoodSteps(){
  return numGoodSteps;
}

bool KCombo::resetSteps(){
  numSteps = 0;
  numGoodSteps = 0;
  return true;
}

KNbr* KCombo::getKNbr(int i){
  return kHood[i];
}
```

A.8 Testing code

```
OldNN::OldNN(int n, Point** data){
  numnodes = n;
  distances = new double*[n];
  distances[0] = NULL;
  int i,j;
  // fill in distances[i][j] for all i > j
  for (i=1; i<n; i++){
    distances[i] = new double[i];
    for (j=0; j<i; j++)
      distances[i][j] = data[i]->dist(data[j]);
  }
}

OldNN::~OldNN(){
  for (int i=1;i<numnodes;i++)
    delete[] distances[i];
  delete[] distances;
}

int OldNN::singleNN(int indx, Point** data){
  int bestIndex = (indx+1)%numnodes;  // chosen arbitrarily
  double minDist;
  if (indx < bestIndex) { minDist = distances[bestIndex][indx]; }
  else { minDist = distances[indx][bestIndex]; }

  for (int i=0;i<numnodes;i++){
    double curDist;
    if (i == indx) { curDist = (double) 0; }
    else {
      if (i > indx) { curDist = distances[i][indx]; }
      else { curDist = distances[indx][i]; }
    }
    if (curDist < minDist){
      minDist = curDist;
      bestIndex = i;
    }
  }
  return bestIndex;
}

int OldNN::epsilonNN(int indx, Point** data, double range, int** output){
  FingerList* firstOut = NULL;
  FingerList* lastOut = NULL;
  int curOut[1];
  int numOut = 0;
  for (int i=0;i<numnodes;i++){
    double curDist;
    if (i == indx) { curDist = (double) 0; }
    else {
      if (i > indx) { curDist = distances[i][indx]; }
      else { curDist = distances[indx][i]; }
    }
    if (curDist < range){
      curOut[0] = i;
      numOut++;
      if (firstOut == NULL){
        firstOut = new FingerList(curDist, 1, curOut);
        lastOut = firstOut;
      }
      else {
        lastOut->setNext(new FingerList(curDist, 1, curOut));
        lastOut = lastOut->getNext();
      }
    }
  }
  // dump all the answers into the output array, and delete the FingerLists
  if (numOut > 0){
    *output = new int[numOut];
    FingerList* fl = firstOut;
    FingerList* tmp;
    for (int i=0;i<numOut;i++){
      (*output)[i] = fl->getListElt(0);
      tmp = fl;
      fl = fl->getNext();
      delete tmp;
    }
    if (fl != NULL) { cout << "error making array" << endl; }
  }
  else { *output = NULL; }
  return numOut;
}

int OldNN::kNN(int indx, Point** data, int k, int** output){
  if (k > numnodes) {
    cout << "invalid k-n-n query-- k is too high." << endl;
    *output = NULL;
    return 0;
  }
  *output = new int[k];
  double maxR = 0;
  int worstIndex;
  for (int i=0;i<numnodes;i++){
    double curDist;
    if (i == indx) { curDist = (double) 0; }
    else {
      if (i > indx) { curDist = distances[i][indx]; }
      else { curDist = distances[indx][i]; }
    }
    if (i<k){
      (*output)[i] = i;
      if (curDist > maxR){
        maxR = curDist;
        worstIndex = i;
      }
    }
    else if (curDist < maxR){
      (*output)[worstIndex] = i;
      // find the max-r for the altered list
      maxR = curDist;
      for (int j=0;j<k;j++){
        double tempR;
        if ((*output)[j] == indx) { tempR = (double) 0; }
        else {
          if ((*output)[j] < indx) { tempR = distances[indx][(*output)[j]]; }
          else { tempR = distances[(*output)[j]][indx]; }
        }
        if (tempR > maxR){
          maxR = tempR;
          worstIndex = j;
        }
      }
    }
  }
  return k;
}

int loadData(bool swiss, int nsize, Point*** points,
             Point*** kPoints, int** derand){
  int d;
  if (swiss)
    d = 3;
  else d = 4096;
  int n = nsize;

  cout << "n is " << n << ", d is " << d << endl;

  if (n-1 > RAND_MAX)
    cout << "warning, RAND_MAX is " << RAND_MAX
         << ", which is too small for n!" << endl;
  if (n < 2)
    cout << "n is too small for KR alg to work correctly!" << endl;

  // initialize seed for random generator.
  srandom((unsigned)time(0));

  // we will use t1 and t2 to calculated elapsed times.
  clock_t t1, t2;
  mwArray matpts;
  double* thedata;
```

```
if (swiss) {
  load("swiss_roll_data.mat", "X_data", &matpts);
  thedata = new double[3*20000];
}
else {
  load("face_data.mat", "images", &matpts);
  thedata = new double[4096*698];
}
matpts.ExtractData(thedata);

// allocate the input array from the heap
double** input = new double*[n];
for (int i=0;i<n;i++)
  input[i] = new double[d];

// fill the input array with the first n points
// of the imported (.mat) data set.
for (int i=0;i<n;i++)
  for (int j=0;j<d;j++)
    input [i][j] = thedata[(d*i)+j];

delete[] thedata;

// create an array containing the data points
// and create the Points inside

*points = new Point*[n];
double* coords = new double[d];
for (int i=0;i<n;i++){
  for (int j=0;j<d;j++)
    coords[j] = input[i][j];
  (*points)[i] = new Point(d,coords);
}
delete[] coords;

for (int i=0;i<n;i++)
  delete[] input[i];
delete[] input;

// construct kPoints, a copy of Points that
// the karger alg will shuffle and use.

*kPoints = new Point*[n];
for (int i=0;i<n;i++)
  (*kPoints)[i] = (*points)[i];

// allocate "derand," the mapping
// from 'kPoints' index to 'Points' index

(*derand) = new int[n];
for (int i=0;i<n;i++)
  (*derand)[i]=i;

cout << "randomizing data set for Karger alg. . .";
```

```
  t1=clock();
  // randomize the order of the data pts in the array,
  // while maintaining correctness of the mapping.
  cout << " (rand seq: " << random() << " " << random() << ") ";
  for (int i=0;i<n;i++){
    int j = FingerList::randRange(0,n-1);  // assume n < (compiler's max)

    Point* temp = (*kPoints)[i];
    (*kPoints)[i] = (*kPoints)[j];
    (*kPoints)[j] = temp;

    int k = (*derand)[i];
    (*derand)[i] = (*derand)[j];
    (*derand)[j] = k;
  }
  t2=clock();
  int kRand = t2-t1;
  cout << "took " << kRand << " ticks." << endl;
  return kRand;
}

int compare2(int nsize, bool epsmode, double param, int fListLen,
             Point** points, Point** kPoints, int* derand, int kRand){
  int n = nsize;
  bool modeIsEpsilon = epsmode;
  double epsilon = param;   // parameter for epsilon-nearest-neighbors
  int kParameter = (int) param;
  int fingerListLength = fListLen;
  if (fingerListLength > n){
    fingerListLength = n;
    cout << "optimization: fingerList size reduced to "
         << n << ", which is input size." << endl;
  }

  if (modeIsEpsilon)
    cout << "epsilon mode; epsilon is " << epsilon << endl;
  else {
    cout << "k mode; k is " << kParameter << endl;
    if (kParameter > fingerListLength)
      cout << "fListSize < k!  KR-Nbr will have to raise fListSize later."
           << endl;
  }

  // initialize seed for random generator.
  srandom((unsigned)time(0));

  // we will use t1 and t2 to calculated elapsed times.
  clock_t t1, t2;

  // allocate the Karger data structure
  NNStruct* nns = new NNStruct(n, fingerListLength);
  cout << "KR structure allocated with fingerlists of size "
       << fingerListLength << endl;
```

```
cout << "Building KR data structure. . . ";

Point::resetUsage();
t1 = clock();
for (int i=n−2;i>−1;i−−)
  nns−>insert(i, kPoints);
t2 = clock();
int kBuild = t2−t1;
int kBuildUsage = Point::getUsage();
cout << "took " << kBuild << " ticks and used "
     << kBuildUsage << " dist calcs." << endl;

int kBuildGoodSteps = nns−>getGoodSteps();
int kBuildSteps = nns−>getSteps();
nns−>resetSteps();

cout << "Querying with KR method, and derandomizing. . . ";
Point::resetUsage();
t1 = clock();
int** kOut = new int*[n];
int* kSizes = new int[n];
for (int i=0;i<n;i++){
  if (modeIsEpsilon)
    kSizes[derand[i]] =
      nns−>findRange(kPoints[i],kPoints,epsilon,&(kOut[derand[i]]));
  else
    kSizes[derand[i]] =
      nns−>findK(kPoints[i],kPoints,kParameter,&(kOut[derand[i]]));
  for (int j=0;j<kSizes[derand[i]];j++)
    kOut[derand[i]][j] = derand[kOut[derand[i]][j]];
}
t2 = clock();
int kQuery = t2−t1;
int kQueryUsage = Point::getUsage();
cout << "took " << kQuery << " ticks and used "
     << kQueryUsage << " dist calcs." << endl;

int kQueryGoodSteps = nns−>getGoodSteps();
int kQuerySteps = nns−>getSteps();
delete nns;

// construct my variant of karger structure
cout << "building/querying karger nbr structure with fingerlists of "
     << fingerListLength << ". . . " ;
Point::resetUsage();
t1 = clock();
int** myOut = new int*[n];
int* mySizes = new int[n];
int mySteps, myGoodSteps;
if (!modeIsEpsilon){
  if (fingerListLength < kParameter){
    fingerListLength = kParameter;
    cout << "KR-Nbr is raising fListSize to " << kParameter << endl;
  }
  KCombo* myNns = new KCombo(n, fingerListLength, kPoints, kParameter);
  for (int i=0;i<n;i++){
    KNbr* nbr = myNns->getKNbr(i);
    if (nbr->getSize() != kParameter)
      cout << "size of " << i << " neq k!" << endl;
    mySizes[derand[i]] = nbr->getSize();
    myOut[derand[i]] = new int[mySizes[derand[i]]];
    int* tmp = nbr->getNbrs();
    for (int j=0;j<mySizes[derand[i]]; j++)
      myOut[derand[i]][j] = derand[tmp[j]];
  }
  mySteps = myNns->getSteps();
  myGoodSteps = myNns->getGoodSteps();
  delete myNns;
}
else {
  EpsCombo* myNns = new EpsCombo(n, fingerListLength, kPoints, epsilon);
  for (int i=0;i<n;i++){
    EpsNbr* nbr = myNns->getEpsNbr(i);
    mySizes[derand[i]] = nbr->getSize();
    myOut[derand[i]] = new int[mySizes[derand[i]]];
    FingerList* tmp = nbr->getFirst();
    for (int j=0; j<mySizes[derand[i]]; j++){
      myOut[derand[i]][j] = derand[tmp->getListElt(0)];
      tmp = tmp->getNext();
    }
  }
  mySteps = myNns->getSteps();
  myGoodSteps = myNns->getGoodSteps();
  delete myNns;
}
t2 = clock();
int myTime = t2-t1;
int myUsage = Point::getUsage();
cout << "took " << myTime << " ticks and used "
     << myUsage << " dist calcs." << endl;

cout << "Verifying consistency of results. . . ";
t1 = clock();
int mismatches = 0;
for (int i=0;i<n;i++){
  if (kSizes[i] != mySizes[i]){
    cout << "mismatch for point " << i << ": kSize= ";
    cout << kSizes[i] << " and kNbrSize= " << mySizes[i] << ". ";
  }
  if (!modeIsEpsilon)
    if ((kSizes[i] != kParameter) || (mySizes[i] != kParameter))
      cout << "exactly k not found for point " << i << endl;
  // first match kOut against myOut.
  for (int j=0;j<mySizes[i];j++){
    bool match = false;
    for (int k=0;k<kSizes[i];k++)
      if (kOut[i][k] == myOut[i][j]) { match = true; }
    if (!match) {
        mismatches++;
        cout << i << ", " << myOut[i][j] << endl;
      }
    }
    // then match myOut against kOut.
    for (int j=0;j<kSizes[i];j++){
      bool match = false;
      for (int k=0;k<mySizes[i];k++)
        if (myOut[i][k] == kOut[i][j]) { match = true; }
      if (!match) {
        mismatches++;
        cout << i << ", " << kOut[i][j] << endl;
      }
    }
  }
  t2 = clock();
  cout << "took " << t2-t1 << " ticks." << endl;
  if (mismatches==0)
    cout << "Results match!" << endl;
  else {
    cout << "Results DO NOT MATCH-- there were " <<
      mismatches << " mismatches, " << endl;
    cout<< " for a failure rate of " <<
      (double)mismatches/(double)(n*n) << endl;
  }

  clock_t tFinish = clock();
  cout << endl;
  cout << "time for KR was: " << endl
       << kRand << " rand + " << kBuild << " build + "
       << kQuery << " query. Total = " << kRand+kBuild+kQuery << endl;
  cout << "dist calc usage for KR was:" << endl
       << kBuildUsage << " build + " << kQueryUsage << " query."
       << " Total = " << kBuildUsage+kQueryUsage << endl;
  cout << "step success for build was " << kBuildGoodSteps
       << " / " << kBuildSteps << " = "
       << (double)kBuildGoodSteps / (double)kBuildSteps
       << endl;
  cout << "step success for query was " << kQueryGoodSteps
       << " / " << kQuerySteps << " = "
       << (double)kQueryGoodSteps / (double)kQuerySteps << endl;
  cout << endl;
  cout << "time for KR-nbr method was:" << endl
       << kRand << "rand + " << myTime << " build/query. Total = "
       << kRand+myTime << endl;
  cout << "dist calc usage was " << myUsage << endl;
  cout << "step success was " << myGoodSteps << " / " << mySteps << " = "
       << (double)myGoodSteps / (double)mySteps << endl;
  cout << endl;
  cout << "total time taken was......... " << tFinish << endl;
  int oldUsage = (int)((pow(n, (double)2)-n)/2.0);
  cout << "KR-nbr perf indices (over KR) are "
       << (double)(kRand+kBuild+kQuery) / (double)(kRand+myTime)
       << " procTicks and "
      << (double)(kBuildUsage+kQueryUsage) / (double)(myUsage)
      << " distCalcs." << endl;
  cout << "perf indices (over Old) for dist calcs are:" << endl;
  cout << (double)oldUsage / (double)(kBuildUsage+kQueryUsage)
      << " for KR and "
      << (double)oldUsage / (double)(myUsage)
      << " for KR-Nbr." << endl;


  // CLEANUP
  // first deallocate the inner pointers for some 2-d arrays.
  for (int i=0;i<n;i++){
    delete[] kOut[i];   // don't need to check if these are null
    delete[] myOut[i];
    delete points[i];
  }
  // then deallocate remaining dynamic arrays
  delete[] kOut;
  delete[] myOut;
  delete[] kSizes;
  delete[] mySizes;

  return 0;
}

int compare(int nsize, bool epsmode, double param, int fListLen,
            Point** points, Point** kPoints, int* derand, int kRand){
  int n = nsize;
  bool modeIsEpsilon = epsmode;
  double epsilon = param;   // parameter for epsilon-nearest-neighbors
  int kParameter = (int) param;
  int fingerListLength = fListLen;
  if (fingerListLength > n){
    fingerListLength = n;
    cout << "optimization: fingerList size reduced to "
         << n << ", which is input size." << endl;
  }
  if (modeIsEpsilon)
    cout << "epsilon mode; epsilon is " << epsilon << endl;
  else {
    cout << "k mode; k is " << kParameter << endl;
    if (kParameter > fingerListLength)
      cout << "fListSize < k!  KR-Nbr will have to raise fListSize later."
           << endl;
  }

  // we will use t1 and t2 to calculated elapsed times.
  clock_t t1, t2;

  // allocate the Karger data structure
  NNStruct* nns = new NNStruct(n, fingerListLength);
  cout << "Karger structure allocated with fingerlists of size "
      << fingerListLength << endl;
```

```
cout << "Building Karger data structure... ";

Point::resetUsage();
t1 = clock();
for (int i=n-2;i>-1;i--)
  nns->insert(i, kPoints);
t2 = clock();
int kBuild = t2-t1;
int kBuildUsage = Point::getUsage();
cout << "took " << kBuild << " ticks and used "
     << kBuildUsage << " dist calcs." << endl;

int kBuildGoodSteps = nns->getGoodSteps();
int kBuildSteps = nns->getSteps();
nns->resetSteps();

cout << "Querying with KARGER method, and derandomizing... ";
Point::resetUsage();
t1 = clock();
int** kOut = new int*[n];
int* kSizes = new int[n];
for (int i=0;i<n;i++){
  if (modeIsEpsilon)
    kSizes[derand[i]] =
      nns->findRange(kPoints[i],kPoints,epsilon,&(kOut[derand[i]]));
  else
    kSizes[derand[i]] =
      nns->findK(kPoints[i],kPoints,kParameter,&(kOut[derand[i]]));
  for (int j=0;j<kSizes[derand[i]];j++)
    kOut[derand[i]][j] = derand[kOut[derand[i]][j]];
}
t2 = clock();
int kQuery = t2-t1;
int kQueryUsage = Point::getUsage();
cout << "took " << kQuery << " ticks and used "
     << kQueryUsage << " dist calcs." << endl;

int kQueryGoodSteps = nns->getGoodSteps();
int kQuerySteps = nns->getSteps();
delete nns;

// construct my variant of karger structure
cout << "building/querying karger nbr structure with fingerlists of "
     << fingerListLength << "... " ;
Point::resetUsage();
t1 = clock();
int** myOut = new int*[n];
int* mySizes = new int[n];
int mySteps, myGoodSteps;
if (!modeIsEpsilon){
  if (fingerListLength < kParameter){
    fingerListLength = kParameter;
    cout << "KR-Nbr is raising fListSize to " << kParameter << endl;
  }
  KCombo* myNns = new KCombo(n, fingerListLength, kPoints, kParameter);
  for (int i=0;i<n;i++){
    KNbr* nbr = myNns->getKNbr(i);
    if (nbr->getSize() != kParameter)
      cout << "size of " << i << " neq k!" << endl;
    mySizes[derand[i]] = nbr->getSize();
    myOut[derand[i]] = new int[mySizes[derand[i]]];
    int* tmp = nbr->getNbrs();
    for (int j=0;j<mySizes[derand[i]]; j++)
      myOut[derand[i]][j] = derand[tmp[j]];
  }
  mySteps = myNns->getSteps();
  myGoodSteps = myNns->getGoodSteps();
  delete myNns;
}
else {
  EpsCombo* myNns = new EpsCombo(n, fingerListLength, kPoints, epsilon);
  for (int i=0;i<n;i++){
    EpsNbr* nbr = myNns->getEpsNbr(i);
    mySizes[derand[i]] = nbr->getSize();
    myOut[derand[i]] = new int[mySizes[derand[i]]];
    FingerList* tmp = nbr->getFirst();
    for (int j=0; j<mySizes[derand[i]]; j++){
      myOut[derand[i]][j] = derand[tmp->getListElt(0)];
      tmp = tmp->getNext();
    }
  }
  mySteps = myNns->getSteps();
  myGoodSteps = myNns->getGoodSteps();
  delete myNns;
}
t2 = clock();
int myTime = t2-t1;
int myUsage = Point::getUsage();
cout << "took " << myTime << " ticks and used "
     << myUsage << " dist calcs." << endl;

cout << "Building old nn structure. . . ";
Point::resetUsage();
t1 = clock();
OldNN* ns = new OldNN(n, points);
t2 = clock();
int oldBuild = t2-t1;
int oldBuildUsage = Point::getUsage();
cout << "took " << oldBuild << " ticks and used "
     << oldBuildUsage << " dist calcs." << endl;

cout << "Querying with Old method. . . ";
Point::resetUsage();
t1 = clock();
int** oldOut = new int*[n];
int* oldSizes = new int[n];
for (int i=0;i<n;i++){
  if (modeIsEpsilon)
```

```
      oldSizes[i] = ns->epsilonNN(i,points,epsilon,&(oldOut[i]));
    else
      oldSizes[i] = ns->kNN(i,points,kParameter,&(oldOut[i]));
  }
  t2 = clock();
  int oldQuery = t2-t1;
  int oldQueryUsage = Point::getUsage();
  cout << "took " << oldQuery << " ticks and used "
       << oldQueryUsage << " dist calcs." << endl;

  delete ns;

  cout << "Verifying consistency of results... ";
  t1 = clock();
  int mismatches = 0;
  for (int i=0;i<n;i++){
    if (kSizes[i] != oldSizes[i]){
      cout << "mismatch for point " << i << ": kSize= ";
      cout << kSizes[i] << " and oldSize= " << oldSizes[i] << ". ";
    }
    if (mySizes[i] != oldSizes[i]){
      cout << "mismatch for point " << i << ": mySize= ";
      cout << mySizes[i] << " and oldSize= " << oldSizes[i] << ". ";
    }
    if (!modeIsEpsilon)
      if ((kSizes[i] != kParameter) ||
          (oldSizes[i] != kParameter) ||
          (mySizes[i] != kParameter))
        cout << "exactly k not found for point " << i << endl;
    // first match kOut against oldOut.
    for (int j=0;j<oldSizes[i];j++){
      bool match = false;
      for (int k=0;k<kSizes[i];k++)
        if (kOut[i][k] == oldOut[i][j]) { match = true; }
      if (!match) {
        mismatches++;
        cout << i << ", " << oldOut[i][j] << endl;
      }
    }
    // then match myOut against kOut.
    for (int j=0;j<kSizes[i];j++){
      bool match = false;
      for (int k=0;k<mySizes[i];k++)
        if (myOut[i][k] == kOut[i][j]) { match = true; }
      if (!match) {
        mismatches++;
        cout << i << ", " << kOut[i][j] << endl;
      }
    }
    // last match oldOut against myOut.
    for (int j=0;j<mySizes[i];j++){
      bool match = false;
      for (int k=0;k<oldSizes[i];k++)
        if (oldOut[i][k] == myOut[i][j]) { match = true; }
      if (!match) {
        mismatches++;
        cout << i << ", " << myOut[i][j] << endl;
      }
    }
  }
  t2 = clock();
  cout << "took " << t2-t1 << " ticks." << endl;
  if (mismatches==0)
    cout << "Results match!" << endl;
  else {
    cout << "Results DO NOT MATCH-- there were " <<
      mismatches << " mismatches, " << endl;
    cout<< " for a failure rate of " <<
      (double)mismatches/(double)(n*n) << endl;
  }

  clock_t tFinish = clock();
  cout << endl;
  cout << "time for KR was: " << endl
       << kRand << " rand + " << kBuild << " build + "
       << kQuery << " query. Total = " << kRand+kBuild+kQuery << endl;
  cout << "dist calc usage for KR was:" << endl
       << kBuildUsage << " build + " << kQueryUsage << " query."
       << " Total = " << kBuildUsage+kQueryUsage << endl;
  cout << "step success for build was " << kBuildGoodSteps
       << " / " << kBuildSteps << " = "
       << (double)kBuildGoodSteps / (double)kBuildSteps << endl;
  cout << "step success for query was " << kQueryGoodSteps
       << " / " << kQuerySteps << " = "
       << (double)kQueryGoodSteps / (double)kQuerySteps << endl;
  cout << endl;
  cout << "time for KR-nbr method was:" << endl
       << kRand << "rand + " << myTime << " build/query. Total = "
       << kRand+myTime << endl;
  cout << "dist calc usage was " << myUsage << endl;
  cout << "step success was " << myGoodSteps << " / " << mySteps << " = "
       << (double)myGoodSteps / (double)mySteps << endl;
  cout << endl;
  cout << "time for old method was " << oldBuild+oldQuery << " procTicks,"
       << " dist calc usage was " << oldBuildUsage+oldQueryUsage << endl;
  cout << endl;
  cout << "total time taken was. . . . . . . . . " << tFinish << endl;
  cout << "KR perf indices are     "
       << ((double)(oldBuild+oldQuery)) / ((double)(kRand+kBuild+kQuery))
       << " procTicks and "
       << ((double)(oldBuildUsage+oldQueryUsage)) / ((double)(kBuildUsage+kQueryUsage))
       << " distCalcs." << endl;
  cout << "KR-nbr perf indices are "
       << ((double)(oldBuild+oldQuery)) / ((double)myTime) << " procTicks and "
       << ((double)(oldBuildUsage+oldQueryUsage)) / ((double)(myUsage))
       << " distCalcs." << endl;

  // CLEANUP
  // first deallocate the inner pointers for some 2-d arrays.
  for (int i=0;i<n;i++){
    delete[] kOut[i];   // don't need to check if these are null
    delete[] oldOut[i];
    delete[] myOut[i];
    delete points[i];
  }
  // then deallocate remaining dynamic arrays
  delete[] kOut;
  delete[] oldOut;
  delete[] myOut;
  delete[] kSizes;
  delete[] oldSizes;
  delete[] mySizes;

  return 0;
}

int main(int argc, char** argv){
  if (argc < 6)
    cout << "not enough args." << endl;
  char* dSet = argv[1];
  int inpSize = atoi(argv[2]);
  char* mode = argv[3];
  double param = atof(argv[4]);
  int fingerListSize = atoi(argv[5]);

  bool swissDs=true;  //defaults
  bool epsMd=false;

  if (dSet[0] == 's'){
    swissDs = true;
    cout << "using swiss." << endl;
  }
  else if (dSet[0] == 'f'){
    swissDs = false;
    cout << "using faces." << endl;
  }
  else
    cout << "datset not found, using swiss." << endl;

  if (mode[0] == 'e'){
    epsMd = true;
    cout << "using epsilon." << endl;
  }
  else if (mode[0] == 'k'){
    epsMd = false;
    cout << "using k." << endl;
  }
  else
    cout << "mode not found, using k." << endl;

  Point** datpts;
  Point** kDatpts;
  int* dernd;

  int randTime = loadData(swissDs, inpSize, &datpts, &kDatpts, &dernd);
  compare2(inpSize, epsMd, param, fingerListSize,
           datpts, kDatpts, dernd, randTime);

  delete[] datpts;
  delete[] kDatpts;
  delete[] dernd;

  return 0;
}
```

Bibliography

[1] D. Karger and M. Ruhl. Finding Nearest Neighbors in Growth-Restricted Metrics. STOC 2002.

[2] J. Tenenbaum, V. de Silva, and J. Langford. A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science* 2000.

[3] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A. Wu. An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions. 1999.

[4] M. Belkin and P. Niyogi. Using Manifold Structure for Partially Labeled Classification. NIPS 2002.

[5] K. Clarkson. A Randomized Algorithm for Closest-Point Queries. 1987.

[6] J. Friedman, J. Bentley, and R. Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. 1977.

[7] L. Saul and S. Roweis. Think Globally, Fit Locally: Unsupervised Learning of Nonlinear Manifolds. Univ. of Pennsylvania, 2002.

[8] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. ACM SIGCOMM 2001.

[9] K.V. Mardia, J.T. Kent, and J.M. Bibby. Multivariate Analysis. Academic Press, 1979.

[10] S. Skiena. The Algorithm Design Manual. 1997.

[11] M. Dickerson and D. Eppstein. Algorithms for Proximity Problems in Higher Dimensions.

[12] K. Clarkson. Fast Algorithms for the All-Nearest-Neighbors Problem. FOCS 1983.

[13] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. Introduction to Algorithms, Second Edition. 2001.

[14] The MNIST database of handwritten digits. Available from http://yann.lecun.com/exdb/mnist/ .

[15] H. Samet. Applications of Spatial Data Structures. Addison-Wesley, 1990.