An Adaptive Data Structure for Nearest Neighbors Search in an General Metric Space

Joseph Thomas
Thesis Adviser: John Kececioglu

June 1, 2010

Abstract

We consider the problem of computing nearest neighbors in an arbitrary metric space, particularly a metric space that cannot be easily embedded in $\mathbb{R}^n$. We present a data structure, the *Partition Tree*, that can be constructed in $O(n \log n)$ time, where n is the size of the set of points to be searched, and has been experimentally shown to have an average query time that is a sublinear function of n (roughly $O(\log(n)^\alpha)$ where $4 \leq \alpha \leq 5$). Our experiments show that this data structure could have applications in bioinformatics, particularly protein secondary structure prediction, where it can be used for similarity search among short sequences of proteins' primary structure.

1 Introduction

Similarity search is a problem that arises in many areas of computer science, and can be broadly described as follows: Suppose we possess a database of objects and receive another object as a query. For that query, we would like to find a collection of objects in the database that are the closest to the query.

An everyday example of similarity search involves the popular Internet radio service "Pandora." This application plays each user a personalized sequence of songs according to their tastes. Users input their favorite songs and musicians into the system, and Pandora in turn attempts to select songs from its library that are most similar to the user's input.

A common method for defining a similarity search problem is to select an objective function F that maps a pair of objects to a real number measuring their proximity. To search for objects similar to a given query q one attempts to find elements p of the database which maximize or minimize $F(p, q)$.

To determine the optimal database element quickly, one needs to know additional properties of the objective function. One property which arises frequently is that F is a *metric*. Let X denote the domain of objects under consideration, so that F is a function from $X \times X$ to $\mathbb{R}$. In order to be a metric, F must have three properties:

- *Positivity:* $F(x, y) \geq 0$ for all $x, y \in X$, with equality if and only if $x = y$.
- *Symmetry:* $F(y, x) = F(x, y)$ for all $x, y \in X$.
- *Triangle Inequality:* For any $x, y, z \in X$, $F(x, z) \leq F(x, y) + F(y, z)$.

Intuitively speaking, these three constraints ensure F has the properties expected of a distance. That is, the distance measure between any two points is positive, the distance from p to q is the same as the distance from q to p, and when we consider a "triangle" of three points, the triangle has the familiar property that the sum of any two sides' lengths upper bounds the length of the remaining side.

We define a *metric space* to be a pair (S, d), where S is a set of points and $d : S \times S \to \mathbb{R}$ is a metric. From the perspective of similarity search, smaller values of $d(x, y)$ indicate greater similarity between the points x and y. Given these mathematical objects we can define an important data structures problem:
The K Nearest Neighbors Problem

- Input: A metric space (S, d), a finite set of points $D \subseteq S$, a query point $q \in S$, and $k \in \mathbb{N}$.
- Output: The k points $x_1, x_2, \ldots x_k \in D$ for which the function $x \mapsto d(x, q)$ is smallest. These k points are called the k *nearest neighbors* of q.

Our original interest in similarity search arose as part of a project to predict proteins' secondary structure given its primary structure. From the computer scientist's perspective, a protein's primary structure can be viewed as a sequence of letters chosen from a 20 letter alphabet of amino acids. As part of our prediction procedure we defined a *word* to be an n-tuple of amino acids that appear consecutively within a primary structure. For a given word, we needed the ability to rapidly select the k most similar words from a database of observed words. In our application, similarity is measured by a metric defined on words.

This metric is specified by a 20×20 matrix M and a sequence of weights $\{w_i\}_{i=1}^n$ such that each $w_i \geq 0$ and $\sum_{i=1}^n w_i = 1$. For each pair of amino acids (a, b), M encodes a "substitution cost", $M[a, b]$. For two words $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$, we define:

$$d(A, B) = \sum_{i=1}^{n} w_i M[a_i, b_i]$$

Unlike many metric spaces which arise in applications, ours cannot be immediately embedded in $\mathbb{R}^m$ for some m.[1] This prevented us from directly utilizing many nearest neighbors data structures developed in computational geometry, like the *binary space partition tree*. We decided to invest the time required to develop our own data structure, the *Partition Tree*, that is capable of performing a nearest neighbors search in an arbitrary metric space.

The problem of nearest neighbors search bears some resemblance to problems involving searching a totally ordered set. For instance, in both problems it is natural to employ the algorithm design technique *divide-and-conquer* to organize the data in a way that allows it to be searched quickly. However, where searching a totally ordered set is a well solved problem for which one can make strong asymptotic guarantees about the amount of time and space required to perform a search, researchers have made less progress on the problem of nearest neighbors search.

A pathological example of a metric space may help to explain this state of affairs. For a given set S, the *trivial metric* $d : S \times S \to \mathbb{R}$ is defined to be the map such that $d(x, x) = 0$ for all $x \in S$ and $d(x, y) = 1$ for all $x, y \in S, x \neq y$. Though (S, d) is a metric space, in this case we cannot use the axioms d satisfies to rule out many candidate nearest neighbors in a single operation.

At this point in time, experts on the nearest neighbor search problem can mostly offer only heuristics that work well in practice. For a data set of n points, our data structure guarantees only that the tree can be constructed in $O(n \log n)$ time and a k nearest neighbors query can be solved in $O(n \log k)$ time, which is no better than a naive solution to the problem.

In practice, though, our data structure has been much more effective than the naive solution. We have empirically observed that the average amount of

[1] We considered generalizing the idea of a word, so that instead of an n-tuple of amino acids, one stores n convex combinations of the twenty amino acids at each position. This would have allowed us to embed the space of words in $\mathbb{R}^{20n}$, but the memory required to represent each word would have been too great for our purposes.

time needed to perform a k nearest neighbors query is a sublinear function of $|D|$, where D is the input data set.

We will later see that a search tree is a natural type of data structure to apply to the k nearest neighbors problem. Indeed, examining the work of past researchers, one sees that the data structures they have designed are based on a common strategy that they pursue by a variety of heuristics. Our data structure is distinguished by its *adaptive* heuristics. At tree-construction time, we perform an initial analysis of the data and customize the structure of the search tree accordingly.

The remainder of this thesis consists of two parts. In the first part, we define the partition tree and the heuristics which motivated our design. In the second part, we describe our procedures for testing the performance of the data structure and analyze the experimental results.

2 The Partition Tree Framework

We mentioned earlier that our data structure is developed using the algorithm design technique *divide-and-conquer.* Among nearest-neighbor data structures, the idea of recursively subdividing the data in a way that facilitates rapid search is prevalent; it appears, for instance, in techniques proposed in [2] and [3].

Given a metric space (M, d) and a point $p \in M$, a natural set to study in a divide-and-conquer approach is the *ball* centered at p. Given a positive $r \in \mathbb{R}$, the closed ball $B_r(p)$ of radius r centered at p is defined

$$B_r(p) = \{q \in M : d(p, q) \leq r\}$$

Because the distance function d satisfies the triangle inequality, there is a useful criterion for determining when the intersection of two balls is empty:

Given bounding balls $B_r(p)$ and $B_{r'}(p')$, if $d(p, p') > r + r'$, then $B_r(p) \cap B_{r'}(p') = \varnothing$.

To prove this fact, suppose by way of a contradiction there exists a point q in the intersection. Then $d(p, q) \leq r$ and $d(p', q) \leq r'$. But that implies $d(p, q) + d(p', q) \leq r + r' < d(p, p')$, which contradicts the fact d satisfies the triangle inequality.

Most divide-and-conquer strategies are based upon this criterion. These data structures subdivide the input data set into a collection of bounding balls (which are themselves be recursively subdivided). During a query, they maintain a dynamic bounding ball about the query point, where the radius of the ball is determined by the distance from the query point to the best known solution to the query. As better solutions are found, the radius of this bounding ball shrinks. When it can be shown that the bounding ball about the query and a bounding ball B in the data structure do not overlap, then B (and all its subsets) can be eliminated from the search.

We realize this organizing principle with a tree. Each node N in the tree represents a set of points S_N. Within the tree, a node N is the ancestor of a node N' if and only if $S_N \subseteq S_{N'}$. In addition, each leaf node represents a set containing a single point. Each node is also identified with a bounding ball. This entails storing, for each node N, the center of the bounding ball (a point c) and the radius of the ball, r. The pair (c, r) must be chosen such that $S_N \subseteq B_r(c)$, since we will use the bounding ball to determine whether we need to inspect the points in S_N for a given query.

Nearest neighbor data structures that are organized in a manner similar to ours tend to have a common fundamental search procedure which can be customized with a variety of heuristics to improve performance. We sketch it below:

Procedure 1. *K-Nearest-Neighbors Query Procedure (Sketch)*

- Input: A search tree T, a query point q, and $k \in \mathbb{N}$ the number of nearest neighbors to find.

- Output: The set of k points in T that are closest to q.

1. Initialize a queue S to hold the root of T and allocate an array P for k nearest neighbor candidates.

2. If S is empty, return the current nearest neighbors. Otherwise, let N be the node dequeued from S.

3. Let (c, r) be the point-radius pair specifying the bounding ball of N. Let τ be $\max_{p \in P} d(p, q)$, or ∞ if P is not full. If $d(c, q) \geq r + \tau$, return to 2; we do not need to inspect the points in this node.

4. Inspect $O(1)$ of the points in the set identified with N, to see if any can be used to improve the set of nearest neighbor candidates, P. Using S, enqueue each of node N's children and return to 2.

From the standpoint of a computer scientist who needs to perform nearest-neighbor searches quickly (and is less concerned about theoretically guaranteeing a sub-linear search time), the problem of developing a nearest neighbor data structure is to implement a collection of heuristics that make this query procedure run quickly in practice.

From the sketch of the query procedure, it is clear that the terms in the inequality $d(c, q) \geq r + \tau$ are important — when this inequality holds, we can ignore a subtree (and thereby a possibly large set of points). For this reason, we refer to the inequality as the *pruning inequality* and the terms in the inequality as the *pruning quantities*.

Our goal is to construct the tree in a way that causes this inequality to hold often and ensure that when the inequality holds, a nontrivial portion of the data set can be eliminated from the search.

Given this overview of the data structure and our goals, we must specify two procedures: a *construction procedure* that determines how the tree should be organized to represent the input points and a *query procedure* that specifies how the procedure should be searched for the k nearest neighbors to an input point q.

2.1 Partition Tree Construction

To construct a partition tree for a set of points S in a metric space (M, d), we require a *partition rule* that determines how the collection of points at a node should be partitioned into the l smaller collections of points that will be represented at the subtrees of its children. We also require a *center selection rule* that chooses, for a collection of points S, the element that will serve as the center of its bounding ball.

We mentioned earlier that the terms in the pruning inequality play an important role in determining how quickly a query can be performed. Unfortunately, these terms depend in a dynamic way on the particular q, the query, c, the current center, and the candidates inspected so far, which determines τ. The tree, in contrast, is static—we construct it once and then use it to solve many queries without changing its structure. Consequently, our first task is to obtain static estimates for the dynamic quantities in the pruning inequality.

To compute these estimates, we introduce the idea of a *sparse sample* that approximates the input set of points with a much smaller set.[2] We define the

[2]Later in this section, we will see that a very important application of the sparse sampling

procedure for selecting a sparse sample as follows:

Procedure 2. *The Sparse Sampling Procedure*

- Input: A set of points S, a metric d defined on S, and $k \in \mathbb{N}$.
- Output: A set of at most k points from S, the sparse sample.
- Time Complexity: $O(k|S|)$, by maintaining a few key sums.

1. If S has k or fewer points, return S.
2. Otherwise, let $K := |S|$, and let the points of S be numbered s_i, $i = 1 \dots K$. Let $P := \{s_1, s_2, \dots, s_k\}$ be an initial set of candidate sparse sample points.
3. For each $j := k+1, k+2, \dots, K$, do:
   (a) Let $p := s_j$.
   (b) Define a function $E : S \to \mathbb{R}$ by
   $$E(s) = \sum_{s' \in P \setminus \{s\}} d(s', p) - \sum_{s' \in P \setminus \{s\}} d(s', s)$$
   (c) Let $s^* = \operatorname{argmax}_{s \in P} E(s)$. If $E(s^*) > 0$, then exchange p for s^* in P.
4. Return P.

This greedy optimization selects a set of k points in which the average distance between any two points is large. When we use this procedure, k is a fixed constant so that the time complexity of the procedure is just $O(|S|)$.

We use the sparse sampling procedure to obtain a conservative estimate of the k-nearest-neighbor bounding ball radius (τ in the pruning inequality). To do this, we compute a sparse sample S for the input data set D, and then use a straightforward procedure to find:

$$\hat{\tau} := (1+\epsilon) \max_{p \in S} \min_{q \in D \setminus \{p\}} d(p, q)$$

procedure is to compute, for a given set of points S_N, a pair of points $p, q \in S_N$ for which $d(p, q)$ is large. This application was what initially motivated us to develop the sparse sampling procedure, since spending $\theta(|S_N|^2)$ time to compute the maximally distant pair was too costly for our purposes.

where ϵ is a small constant in $[0, 1]$ (say, 0.05). In words, $\hat{\tau}$ is a fixed percentage larger than the greatest 1-nearest-neighbor distance for each point in the sparse sample.

Once we have computed the parameter $\hat{\tau}$, we are ready to begin recursively partitioning the data set D. Given an internal node N in the tree, representing a set of points S_N, we partition S_N into three subsets: "left" (L), "middle" (M), and "right" (R). The subtrees we generate from each of these subsets will have a center selected from the set of points it represents.

The motivation for partitioning S_N in this way is to obtain two sets of points (L and R), which have disjoint bounding balls that are far apart. In this way, we try to ensure that when N is inspected we will be able to eliminate at least one of L or R from the search. The middle portion M serves as a kind of "slack variable." When S_N contains points which cannot be placed in L or R without undermining our goals, we can place these in the middle portion and refine them on the next partitioning step.

To obtain distant, disjoint bounding balls for L and R, we choose two distinguished "extremal" points x and y from the data set to be the centers of the left and right sets. To identify x and y we compute a sparse sample P for S_N and then select two points from the sample according to an objective function.

For a given point $p \in P$, let $\bar{d}_p$ denote the average distance from p to every other point in S_N. In keeping with the divide-and-conquer strategy, we would like to divide the points of S_N among L, M, R, without making any subtree's bounding ball radius large or putting all of the points in one subtree.

Suppose r is the radius of a subtree T with p at its center. Then if q is an "average" query, we can ignore T in our search if $d(p, q) \approx \bar{d}_p \geq r + \hat{\tau}$. In other words, provided the bounding ball has radius $r \leq \bar{d}_p - \hat{\tau}$, we are likely to be able to eliminate T often. Our heuristic, then, is to add as many points to T as possible, without causing r to exceed $\bar{d}_p - \hat{\tau}$.

To select x and y, then, we define a function ϕ by which maps a point p in the sparse sample to $|\{s \in S_N : d(p, s) \leq \bar{d}_p - \hat{\tau}\}|$. We choose x and y to be the two sparse sample points with the greatest values of ϕ. Thus, x and y have the following desirable properties:

- Among the points in S_N, they are far apart.
- They allow the most points of S_N to be added to their subtrees without exceeding our desired radius bound.

Having computed x and y, we are ready to partition the points in S_N into sets L, M, and R. In partitioning the points, we have two concerns:

- Assign as many points as possible to L and R, without exceeding their ideal bounding ball radii $\bar{d}_x - \hat{\tau}$ and $\bar{d}_y - \hat{\tau}$, respectively.
- Ensure that $L \cup R$ and M each contain at least a small fraction of the points in S_N.

The first goal tries to ensure that when when we prune away either the left or right subtree, this will eliminate a large number of the points from our search. The second goal guarantees that the tree has height $O(\log n)$, where n is the total number of points stored in the tree, which in turn will guarantee $O(n \log n)$ time to construct. In principle, if every time we inspected a node we pruned away at least one child, and this child had at least a nontrivial fraction of the tree's leaves, then a nearest neighbor query would require $\log(n)$ time to solve, which would be ideal.

With these goals in mind, we define the following partitioning procedure:

Procedure 3. *Partitioning S_N*

- Input: A collection of points S_N, the distance function d, extremal points x, y, and the partitioning parameters $\hat{\tau}, \bar{d}_x, \bar{d}_y, \eta$.
- Output: Sets L, M, R such that $S_N = L \cup M \cup R$ and L, M, R are pairwise disjoint.
- Time and Space Complexity: $O(|S_N|)$

Define the following predicates, for an input point $p \in S_N$:

$$\begin{aligned}
\texttt{CloserToX}(p) &:= d(p,x) < d(p,y) \\
\texttt{CloserToY}(p) &:= \neg\texttt{CloserToX}(p) \\
\texttt{InLeftBall}(p) &:= d(p,x) \leq \bar{d}_x - \hat{\tau} \\
\texttt{InRightBall}(p) &:= d(p,y) \leq \bar{d}_y - \hat{\tau} \\
\texttt{IsLeftPoint}(p) &:= \texttt{InLeftBall}(p) \wedge (\texttt{CloserToX}(p) \vee \neg\texttt{InRightBall}(p)) \\
\texttt{IsRightPoint}(p) &:= \texttt{InRightBall}(p) \wedge (\texttt{CloserToY}(p) \vee \neg\texttt{InLeftBall}(p))
\end{aligned}$$

1. Let $k = \lfloor \eta \cdot |S_N| \rfloor$. Remove points x and y from S_N.

2. If $0 < 2k \leq |S_N|$, then perform the following initial partitioning: Let $\phi : S_N \to \mathbb{R}$ be the function $\phi(p) := \min\{d(p, x), d(p, y)\}$. Define a comparison function $<_\phi$ on S_N, by specifying $p <_\phi q$ if $\phi(p) < \phi(q)$. Initialize A and B such that A consists of the k points in S_N under $<_\phi$ and B consists of the k greatest points under $<_\phi$.

3. Initialize $S := S_N \setminus (A \cup B)$. Now write:

$$
\begin{aligned}
L &:= \{x\} \cup \{p \in A : \texttt{CloserToX}(p)\} \cup \{p \in S : \texttt{IsLeftPoint}(p)\} \\
R &:= \{y\} \cup \{p \in A : \texttt{CloserToY}(p)\} \cup \{p \in S : \texttt{IsRightPoint}(p)\} \\
M &:= S_N \setminus (L \cup R)
\end{aligned}
$$

Clearly, further details must be supplied in order to ensure an efficient implementation of this procedure (both in terms of time and space), but these would obscure the main ideas behind our protocol. If the input set S_N is stored in an array, it is possible to carry out the algorithm in place. The solution makes use of a standard k-th order statistic finding algorithm, like the one found in [1].

We still need to specify the *center selection* rule. From the procedure above, it is clear that the centers of the L and R subsets are just x and y, respectively. For the center of the M, we select $\operatorname{argmin}_{m \in M} |d(m, x) - d(m, y)|$.

Having defined a partition rule and center selection rule, we construct the tree by the following recursive procedure:

Procedure 4. *Partition Tree Construction Procedure*

- Input: A set of points S, a metric d defined on S, and the static partitioning parameter $\hat{\tau}$.

- Output: A partition tree T.

- Time Complexity: $O(n \log n)$, where $n = |S|$. This follows from the linear time complexities we have established for the procedures above and an application of the Master theorem.

1. If $S = \varnothing$, return *NULL*.

2. If $S = \{p\}$ for some point p, return a leaf node with p as its center and 0 as its bounding ball radius.

3. At this point, we know $|S| \geq 2$. Compute a sparse sample for S, and select from it a pair of extremal points x, y. Compute $\bar{d}_x$ and $\bar{d}_y$.

4. Fixing η as a small constant in $[0, 1]$, say 0.05, apply the partitioning procedure to obtain sets L,M,R that partition S.

5. Recursively apply the construction procedure to L, M, and R to obtain subtrees T_L, T_M and T_R.

6. Apply the center selection rule to choose a center c for S. Compute the bounding ball radius r for this choice of center.

7. Return a node with c as its center, r as its bounding ball radius, and T_L, T_M, T_R as its subtrees.

2.2 Querying the Partition Tree:

We have customized the general procedure for searching the tree in two ways. Both heuristics relate to the radius of the dynamic bounding ball about the query, τ. Observe that if τ can be made small early in the query process, then when we inspect subsequent nodes it is easier for the pruning inequality $d(p, q) > \tau + r$ to be satisfied, so that more subtrees can be eliminated from the search.

Our first heuristic is called *best-first search.* In the sketch of the query procedure, when a node is inspected its children are placed in a queue for later examination. This does not take into account that by inspecting certain nodes earlier, it may be possible to prune away more subtrees. Therefore, in a best-first search we replace the queue with a heap which orders the nodes in a way we believe will cause τ to decrease rapidly.

The total ordering we define on the nodes is derived from the distance d, and is partially determined during the tree-construction phase. For each node N in the tree, we select a *representative* $R_N \in S_N$ as part of the construction procedure for N. Recall that when we partition S_N, we compute a sparse sample S for S_N. For each $p \in S$, we compute the average distance $\bar{d}_p$ from p to any other point in S_N. Using this information, we set $R_N := \operatorname{argmin}_{p \in S} \bar{d}_p$.

To order the nodes in the queue for a given query q, we specify that $N \leq N'$ if $d(R_N, q) \leq d(R_{N'}, q)$. In other words, we order our search such that nodes with representatives closer to the query are inspected first.

A second heuristic we have implemented, called *bootstrapping*, determines a good initial bound on the k-nearest-neighbors distance by solving a 1-nearest-neighbors problem and then an approximate k-nearest neighbors problem. The implementation is based on the observation that leaves which are close within the search tree (in the sense that they are contained in a common subtree of small height) correspond to points which are likely to be close to one another within the metric space. Consequently, we may be able to obtain a good bound on the final value of τ by first computing the 1 nearest neighbor to the query point, then naively computing k candidate nearest neighbors using the leaves of a small subtree around the 1 nearest neighbor.

To implement this heuristic, we add a "parent pointer" to each node in the tree and define the following procedure:

Procedure 5. *Bootstrap Bounding Procedure*

- Input: A search tree T, a query q, $k \in \mathbb{N}$ indicating the number of nearest neighbors desired.

- Output: A set of k nearest neighbor candidates.

1. Let p denote the nearest neighbor to q. Compute an initial bound b on $d(p, q)$ by computing the distance from q to each element of a sparse sample of the data set D (which as been saved as a field of the search tree).

2. Using b as an initial bound, perform a 1 nearest neighbor search of T to solve for p. Store the leaf node N which represents $\{p\}$.

3. Using the parent pointers, compute N's $\lfloor k/2 \rfloor$th ancestor, N'.

4. Compute a list L of all of the leaves which have N' as their ancestor. Compute the k nearest neighbors to q in L using a naive method. Return these candidates.

Given these two heuristics, our final query procedure is:

Procedure 6. *K-Nearest Neighbors Query Procedure*

Input: A search tree T, a query point q, and $k \in \mathbb{N}$ the number of nearest neighbors to find.

Output: The set of k points in T which are closest to q.

1. Use the Bootstrap Bounding Procedure to compute an initial set of k nearest neighbor candidates. Place these in a max-heap P, which orders nodes by their distance to the query.

2. Initialize a min-heap S to hold the root of T, ordered according to our best first order.

3. If S is empty, return the current nearest neighbors. Otherwise, let N be the node extracted from S.

4. Let (c, r) be the point-radius pair specifying the bounding ball of N. Let τ be $\max_{p \in P} d(p, q)$. If $d(c, q) > r + \tau$, return to 2; we do not need to inspect the points in this node.

5. If $d(c, q) < \tau$, add c to P and delete the largest element of P (the most distant nearest neighbor candidate).

6. Add each of node N's children to S and return to 2.

2.3 Summary

In this section, we considered the procedures for constructing and querying the partition tree. Surveying the field of nearest-neighbor data structures for general metric spaces, one can see that most are variations of a natural divide-and-conquer strategy. This strategy arises from the fact the points in the space satisfy the triangle inequality, which determines a useful criterion for when two bounding balls are disjoint.

Our partition tree data structure features three main heuristics. During tree construction, we approximate the quantities in the pruning inequality, and structure the tree in a way that facilitates search. At query time, we use best-first search and bootstrapping to decrease the size of query's bounding ball, which also decreases query time.

3 Experimental Results

3.1 Methodology

If we examine the published work in the area of nearest neighbor search, we see that among data structures developed for an arbitrary metric space it is common to test a new data structure on a variety of Euclidean and non-Euclidean metric spaces to determine its effectiveness. Yianilos, for example, tested his *Vantage Point Tree* construction on a metric space consisting of points in $\mathbb{R}^n$ and a metric space of black and white images [3].

We have pursued similar experimental procedures, selecting three metric spaces for testing. Nearest neighbors search has the desirable property that we can quantify a data structure's performance by the number of distance calculations it performs during a query. This statistic is easy to track (either explicitly or with the aid of profiling software) and is invariant under the experimenter's choice of programming language. For each metric space we considered, we selected a set of 10^3 points to serve as our *query set.* We also selected sets of size 10^3, 10^4, 10^5, and if possible 10^6 to serve as *data sets.*

Our testing procedure was as follows:

Procedure 7. *Partition Tree Performance Test*

- Input: A data set D, a query set Q, a metric d, and $k \in \mathbb{N}$, the number of nearest neighbors to find during each search.
- Output: A log of the search results.

1. Construct a partition tree from D and d.
2. For each $q \in Q$, use T to compute the k nearest neighbors in D to q and report how many distance calculations were performed during the query. (It may be useful to record other profiling data, such as the distances from the nearest neighbors to the query).

One of the principle aims of our data structure was to perform significantly fewer distance computations than a naive search. Consequently, for each query operation a natural way to measure performance is to examine the ratio

$$\frac{(\text{Total distance computations performed during a query})}{(\text{ Total points in the data set })}$$

since the denominator represents number of distance computations the naive approach would perform. Viewed as a percentage, we can say that the closer to 0% this statistic is, the better the search performance.

A second useful observation is that we can adequately measure the data structures' performance by looking at the average of this ratio over a large number of queries. Most applications of a nearest neighbor data structure (including our own) will run in batch mode, so we are more concerned with the total cost of many query operations than we are with the cost of any particular query.

In the introduction, we mentioned that we are particularly interested in efficiently solving nearest neighbor queries in a space of "protein-words," since this has applications in bioinformatics. We created a second set of tests to precisely determine the data structure's performance in this case. We also wanted to see how our data structure compares with another published nearest neighbors data structure for this application. To this end, we created data sets of size 2^k for $k = 9, 10, \ldots, 20$ and implemented the simplest "Vantage Point Tree" described by Yianilos. To compare the two data structures, we applied the test described above for each data set, using the same set of 10^3 query elements. To analyze the results, we compared the average number of distance computations per query for each test.

3.2 Metric Spaces Selected for Testing

Mathematics and computer science provide a variety of examples of metric spaces on which to test nearest neighbor data structures. Perhaps the most obvious metric space arises in multivariable calculus, $\mathbb{R}^n$ under the Euclidean (L_2) norm. As a test case, this space has the advantages that (for $1 \leq n \leq 3$) we easily produce diagrams detailing how a given data set of points is distributed in the space and how our data structure has selected bounding balls to partition those points. We constructed test data for this space by randomly selecting points within the unit n-dimensional hypercube (the Cartesian product of n copies of the set $[0, 1]$).

Because of our interest in metrics defined on words, we also considered a common metric from information theory, the *Levenshtein Distance*. This distance is defined on any two strings of symbols, for our purposes the standard 26 letters of the English alphabet. Given strings $V = \{v_i\}_{i=1}^n$ and $W = \{w_j\}_{j=1}^m$, we define $d(V, W)$ to be the minimum number of string operations required to transform V into W, where a valid string operation consists of deleting, replacing, or inserting a letter into V. This distance can be computed using dynamic programming. To construct test data for this space, we randomly selected sets of words from a dictionary of English words.

The final metric space we considered was the space of "protein-words" described in our introduction. In this case, we chose words of length 7. Recall that the distance defined on this space depends upon a table M that describes, for every pair of amino acids a, a', the (positive) substitution cost of replacing a with a'. We obtained M from a common matrix for protein sequence alignment, *BLOSUM62*. The distance also depends upon a sequence of weights $\{w_i\}_{i=1}^7$, all positive and summing to 1. We chose to make our weights symmetric about the middle position (so $w_1 = w_7$, $w_2 = w_6$, and $w_3 = w_5$), subject to the conditions that $w_3 = w_4/2$, $w_2 = w_3/2$, and $w_1 = w_2/2$. We constructed test data for this metric space by decomposing a set of proteins from the Protein Data Bank. These proteins were chosen to have less than 30% sequence identity by the researchers in [4].

3.3 Query Test Results

In this section, we provide the details of our experiments, for each of the three metric spaces we decided to consider. In each table entry, we record the average number of distance calculations per query, as a percentage of the total data set size for that test. Each test (determined by a data set size and a number of nearest neighbors) is performed twice, once with bootstrapping and once without. For tests that use bootstrapping, we report the average query to data set percentage for only the distance calculations performed during the *final* search of the tree, after we have established a bound on the k-nearest neighbors distance.

Table 1: Query Tests for Points in $\mathbb{R}^2$

	With Bootstrapping				Without			
$\lvert D\rvert$	1-NN	3-NN	5-NN	7-NN	1-NN	3-NN	5-NN	7-NN
10^3	4.861	5.843	6.093	6.409	5.797	6.482	6.924	7.315
10^4	0.310	0.310	0.420	0.450	0.480	0.480	0.530	0.530
10^5	0.063	0.180	0.132	0.138	0.177	0.180	0.190	0.200
10^6	0.010	0.010	0.010	0.010	0.010	0.010	0.010	0.010

There are several important details to be gleaned from this data. First, each column contains a descending sequence of values (as a function of the data set size $|D|$). This indicates that, if we view the average number of distance computations the data structure performs during a query as a function of the data set's size, this function is a sublinear function of $|D|$.

Second, the data structure solves queries very quickly for points in $\mathbb{R}^2$ in comparison with its performance on the other two metric spaces. This is not surprising, given that the points we selected are evenly distributed over the space (the unit square), the dimension of the space is low, and the metric on the space has many useful analytic properties. In short, this space is ideally suited to constructing nearest neighbor search trees.

Finally, we can see that although bootstrapping had a positive effect, that effect was not particularly substantial, particularly when one considers that two searches of the tree must be performed. The "1-NN with bootstrapping" column is particularly informative, since it tells us how long it takes to search the tree when we already know the distance from the query to the nearest neighbor.

Table 2: Query Tests for English Words

	With Bootstrapping				Without			
$\lvert D\rvert$	1-NN	3-NN	5-NN	7-NN	1-NN	3-NN	5-NN	7-NN
10^3	57.690	58.170	58.208	58.223	57.772	58.170	58.208	58.223
10^4	48.136	53.533	54.441	54.914	49.597	53.538	54.444	54.916
10^5	23.774	37.951	41.233	42.885	27.138	38.019	41.279	42.918

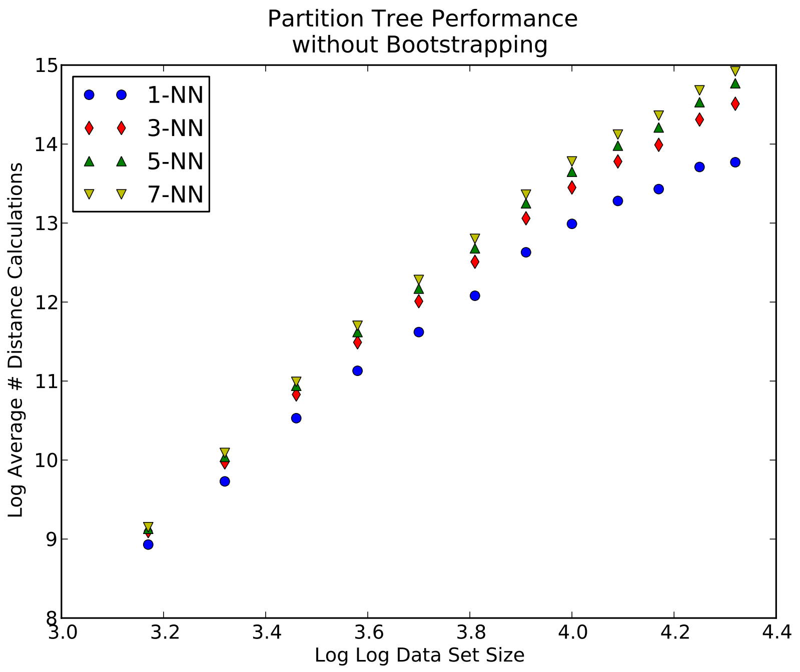
In Table 2, we can see that the data structure's performance is significantly worse for words under the Levenshtein distance than what we observed for points in $\mathbb{R}^2$. However, the data structure still manages to ensure the average query time remains a sublinear function of $|D|$. The metric space of English words under the Levenshtein distance differs considerably from the space of points in $\mathbb{R}^2$ under the Euclidean norm. Intuitively, if we choose a point p in the latter metric space, we can expect the function $d(p, \cdot)$ to attain a wide variety of values for points in the data set. In contrast, the range of the Levenshtein distance is constrained to $\mathbb{N}$ and the words in the English language are not uniformly distributed random strings. As a result, it may be that constructing a good search tree for these words is more difficult, particularly for small sets of words.

Table 3: Query Tests for Protein Words

	With Bootstrapping				Without			
$\lvert D\rvert$	1-NN	3-NN	5-NN	7-NN	1-NN	3-NN	5-NN	7-NN
10^3	39.947	50.949	53.119	54.447	45.024	51.253	53.381	54.714
10^4	12.245	23.444	25.919	27.720	19.494	25.156	28.120	30.162
10^5	2.292	5.376	6.012	6.544	4.590	6.320	7.295	8.044
10^6	0.185	0.955	1.077	1.204	0.756	1.226	1.460	1.633

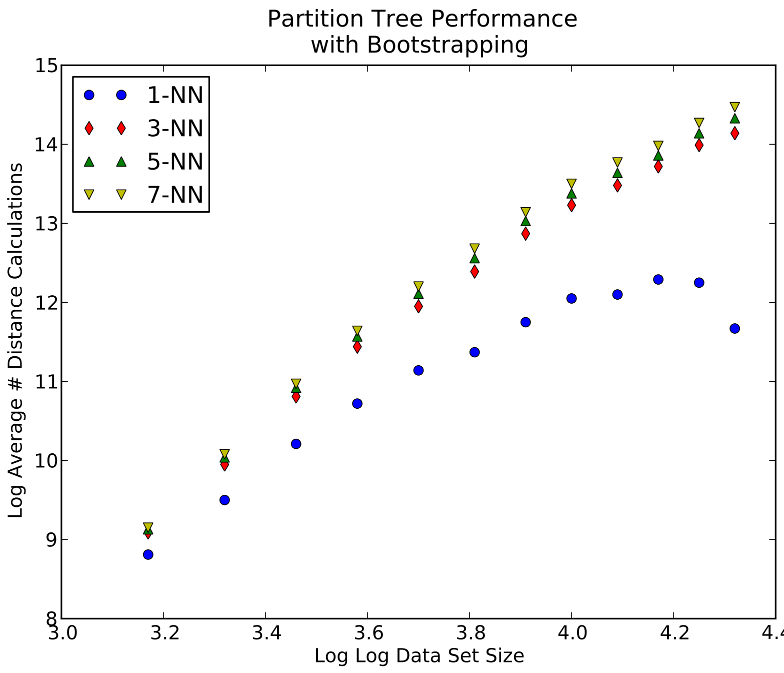
In Table 3, we can see that our data structure performs fairly well on the metric space we have defined on protein words. In particular, since we want to use this data structure to solve nearest neighbor queries for very large sets of words (between 1 and 3 million points), it is heartening to see that the data structure searches only a small fraction of the words when $|D| = 10^6$.

The second set of experiments further clarifies the impression that our data structure can solve queries in sublinear time and suggests that our data structure is competitive with Yianilos' Vantage Point Tree.



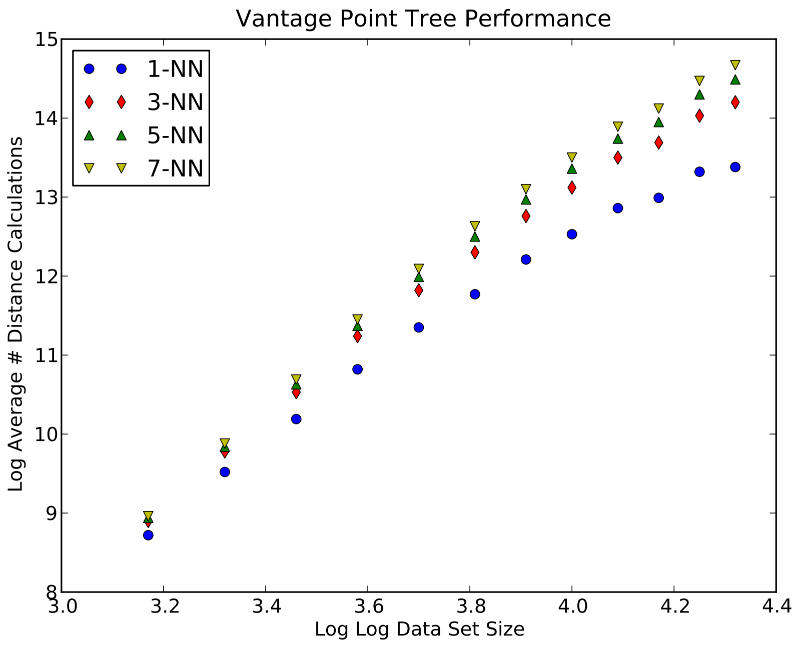
From the chart above, we can see that for a data set S, $\log\log|S|$ is roughly proportional to the logarithm of the average number of distance calculations required to solve a query. Further calculation reveals the slope of the lines above to be somewhere between 4 and 5, which suggests a query requires time $O(\log(|S|)^{\alpha})$ where $4 \leq \alpha \leq 5$.

Our second set of experiments also allows us to better characterize the effect of bootstrapping.



From the chart above, we can see that with the exception of the 1-NN test, each line has slope between 4 and 5. (The data structure's performance in the 1-NN case may be an artifact of the input data or the fact that in this case the bootstrap step provides a "perfect" bound for solving the query.) It appears that while bootstrapping can improve query time somewhat, it does not cause queries to run asymptotically faster than a standard query.

Our final graph, when compared with the first, demonstrates that the Vantage Point Tree and Partition Tree can solve nearest neighbor queries equally quickly. This is encouraging for two reasons. In a Vantage Point Tree, the elements in the data set are in one-to-one correspondence with the nodes of the tree. In contrast, in a Partition Tree the elements in the data set are in one-to-one correspondence with the leaves of the tree. In addition, during a query to a Vantage Point Tree, we inspect a node at the cost of one distance calculation, whereas in a Partition Tree, we inspect a node at the cost of two distance calculations. Thus the Partition tree is at a disadvantage in comparison to the Vantage Point Tree—it has more nodes (likely twice as many) and performs more distance calculations per node inspected. The fact the Partition Tree remains competitive with the Vantage Point Tree, despite these limitations, suggests that our data structure could be made faster than the Vantage Point Tree with some minor modifications.



Namely, the "center" and "representative" fields in the current tree nodes could be exchanged for a single field, and we could modify our rules for structuring the tree so that the data set elements are in one-to-one correspondence with the tree's nodes.

From the graph, some additional calculation shows that like the Partition Tree, for a data set S, a query to the Vantage Point tree generated from S requires time $O(\log(|S|)^{\alpha})$ where $4 \leq \alpha \leq 5$.

4 Conclusion

We have described a static data structure, the *Partition Tree*, which solves the k-nearest-neighbor problem. If n is the size of the input data set, the data structure can be constructed in $O(n \log n)$ time and queried in $O(n)$ time, although in practice we have seen that on average, the query operation takes time sublinear in n (approximately $O(\log(n)^\alpha)$, where $4 \leq \alpha \leq 5$).

Among nearest neighbor data structures, many take the form of a search tree based upon a criterion for deciding when two balls are disjoint. This criterion depends upon an important inequality, the *pruning inequality.* Our data structure is developed from an analysis of this inequality. At tree-construction time, we estimate the terms in this inequality and adaptively structure the tree in a way that makes it likely for this inequality to hold at query time. This substantially reduces the number of elements we need to inspect during a query.

We also introduced two query-time heuristics, *bootstrapping* and *best first search.* We observed that bootstrapping, at least as it is currently implemented, does not provide a significant enough improvement in the final search time to make up for its initial 1 nearest neighbor search. However, it seems quite plausible that one could modify the procedure to make this heuristic more effective.

Another area one could further analyze is the idea that some metric spaces intrinsically lend themselves to nearest-neighbors search. For instance, we saw experimentally that searching for nearest neighbors in $\mathbb{R}^2$ under the Euclidean norm is substantially “easier” than searching for nearest neighbors among English words under the Levenshtein distance. One might be able to characterize this by studying a histogram of all pairwise distances between any two points in a large set of representatives from the space. Well understood metric spaces, like the Euclidean plane and the trivial metric space, could then be compared with the resulting histogram to gain a better understanding of the space under consideration. In particular, one might be able to analyze these histograms at tree-construction time to better structure structure the tree for search.

5 Works Cited

[1] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford. *Introduction to Algorithms* (3rd ed.). MIT Press. 2009.

[2] Ullman, Jeffrey K. *Metric Trees. Applied Mathematics Letters.* Vol. 4, No. 5, pp. 61-62, 1991.

[3] Yianilos, Peter N. *Data Structures and Algorithms for Nearest Neighbors*

Search in General Metric Spaces. Proceedings of the Fourth ACM-SIAM Symposium on Discrete Algorithms, January 1993.

[4] Zhou, Tuping; Shu, Nanjiang; Hovmller, Sven. *A novel method for accurate one-dimensional protein structure prediction based on fragment matching. Bioinformatics.* 2010 Feb 15;26(4):470-7. Epub 2009 Dec 9.