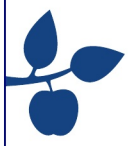


SYDDANSK UNIVERSITET
UNIVERSITY OF SOUTHERN DENMARK

Neighbor finding in high dimensional spaces

Master's Thesis by

Jesper Skovgård Nielsen
jesper@runlevel0.dk

Advisor: Kim Skak Larsen

Department of Mathematics and Computer Science
University of Southern Denmark, Odense

May 1, 2011

Abstract

Modern technologies have made it possible to collect vast amounts of data. When data have been collected they must be ordered and/or searchable in order to be useful. Having unorganized data is pointless in most applications. A common problem with large data sets is to identify similarities in the data and organize data according to these similarities. Thus enabling searches after objects with the same features.

To provide these kinds of searches, three kinds of methods can be applied: Brute force search which is implemented as a linear search, space partitioning trees, which are often implemented as KdTrees or R-Trees, and hashing often implemented as Locality Sensitive Hashing. We have implemented at least one method from each class and examined them from a practical standpoint.

More specifically, we have tested the methods on uniform data with uniform query points in a high-dimensional space. This turned out to be the hardest class of data sets. Because natural data is often clustered, we have tested the methods on synthetically generated clustered data, in order to make it evident which parameters in the clustering slows the search.

Our results confirm results from earlier papers, such as [WSB98]: In general, space partitioning methods do not perform well in high-dimensional spaces. We also show that when a query point is placed close to a data point, space partitioning methods can perform well, even in high dimensions.

Locality Sensitive Hashing performs well on clustered data where data and query points come from the same distribution. On uniform data with uniform query points, Locality Sensitive Hashing does not seem to be much of an improvement over a linear search as it, in general, struggles to get the examined part of the data set below 75%.

Our conclusion is that one needs to choose the right tool for the job. It seems there is not one method which dominates the others. Thus, we provide a method recommendation in our conclusion. It is neither exhaustive nor set in stone, but provides guidelines based on our findings.

Acknowledgments

The people mentioned in this section all deserve a big thank you. Without you this thesis would never have come together.

First and foremost Ole K. Neckelmann from TriVision needs a big thank you for providing me with the opportunity to write this thesis. Second for giving me a fun tour of your company. It was truly inspiring to see the things you are working on!

Kim Skak Larsen for providing me with good ideas and helpful discussions.

Anne Lauge for her big help with the proofreading.

My supportive and patient girlfriend, Lisbeth, for her great support along the way when things got tough.

My family for always inspiring me.

Contents

1 Introduction

Modern technologies have made it possible to collect vast amounts of data. When data have been collected they must be ordered and/or searchable in order to be useful. Having unorganized data is pointless in most applications. A common problem with large data sets is to identify similarities in the data and organize data according to these similarities. Thus, enabling searches after objects with the same features. An example could be: Given an image, find images with some of the same features.

Lets say we save a set of feature vectors in a database stored as points. We are now faced with the following problem: Given a query point, q, and a data set, P, find the point in P which is closest to q, by some measure. This process is usually referred to as nearest neighbor search.

To obtain a feature vector from an image, we can gray-scale the image and make a color histogram. This provides us with a 256 dimensional signature of the image. Other methods such as face recognition could be employed.

Variations of the nearest neighbor problem includes finding the k points which are closest to q as well as finding all the points which have at most distance R to q. More precise definitions of the near neighbor problems will be given in the preliminaries in Chapter 2.

Near neighbor search in general has a wide range of use-cases in real life:

Biometric access control: Access control methods based on face recognition, finger prints, and iris (eye) scans, all rely on nearest neighbor search to match people trying to gain access with trusted individuals in a database.

Content based retrieval: The online service `tineye.com` is in the words of its creators: "a reverse image search engine. It finds out where an image came from, how it is being used, if modified versions of the image exist, or if there is a higher resolution version". We have already sketched how this application utilizes near neighbor search.

The iPhone app `listen.app` does exactly the same, just with music.

Recommendation/add systems: Shopping sites like `amazon.com` save the shopping habits of their users, in oder to present products of interest to their users. This is done by looking at a users recently bought items and matching them to similar products.

Google adds does something similar. By analyzing searches and websites visited, they analyze behavior and present adds which may have interest to a given user.

The list is not exhaustive but it gives some idea as to how applicable near neighbor search is.

A well-studied class of the near neighbor problems is those in Euclidean space where points reside in a d-dimensional space and distances are given by the Euclidean distance function. For many practical problems, dimensionality may rise to several hundreds, even thousands.

For lower dimensions, $d < 10$, efficient algorithms exists. These are mainly tree-based space partitioning algorithms. Despite much effort, however, for high enough d the space partitioning algorithms will degrade to a linear search [WSB98]. This phenomenon is referred to as "the curse of dimensionality".

In high dimensions things get more tricky, but modern hashing methods can achieve good query times under the right conditions.

1 Motivation

Throughout discussions with the danish company TriVision, the following problem arose: They were building a face recognition system and needed fast similarity search in high dimensions. A face scan would result in a 84 dimensional point. To recognize someone, they needed to find the best match in a database of 84 dimensional points.

TriVision has developed a quite impressive product. They first photograph a person. Then the face is extracted from the photo. Facial features are identified and the features of the face are fitted to a parametric model. The 84 dimensional vector represents the parameters of the model.

2 Preliminaries

This chapter will contain a short primer in the notation and terminology used throughout this thesis.

1 Notation

This section presents notation used throughout the thesis.

1.1 Distances

For two points x and y we say that the distances between them is $\|x - y\|$. This terminology is used when no specific distance measure is needed.

In general we say a distance function over a p-norm is given by

$$\|x - y\|_p = \left(\sum_{i=1}^{d} (x_i - y_i)^p \right)^{1/p} \tag{2.1}$$

where d is the number of dimensions of the points.

Thus, the Euclidean distance between x and y is given by $\|x - y\|_2$.

1.2 Recall

We define recall as

$$recall = \frac{\#\text{nearest neighbors found}}{\#\text{queries}} \tag{2.2}$$

Thus, if make 10 queries and the algorithm being tested finds the nearest neighbor 7 times, we say the recall is 0.7. We say that $\%recall = recall \cdot 100$.

We use this as a measure for the quality of the probabilistic algorithms. A good algorithm should have a high recall, thus find the nearest neighbor most of the time.

1.3 Sets

We use the notation $\{1, \ldots, 5\}$ for the discrete set, containing the numbers $1, 2, 3, 4, 5$.

The notation $[0 : 1]$ is used for the continuous set of all numbers between zero and one, such as 0.42.

2 Near Neighbor Problems

We now define the variants of the near neighbor problem in more detail. We give a strict mathematical definition when necessary. We will also start using the notation used throughout the thesis.

We call this class of problems the near neighbor problems, this includes the nearest neighbor problem.

2.1 Nearest neighbor

The nearest neighbor (NN) problem is the simplest of the near neighbor problems. The problem is defined as follows: Given a set of points, P, and a query point, q, both in d dimensions, find the point $r \in P$ for which it holds that $\forall p \in P : \|p - q\| \geq \|r - q\|$. By using the general distance notation we imply that the problem is well-defined for all distance measures.

If $\exists x, y \in P : \|x - q\| = \|y - q\|$, the solution is no longer well-defined. We assume throughout this thesis that the nearest neighbor is always well-defined. As an implication, we also assume that points are unique in the sense that they at least differ in one coordinate.

The nearest neighbor problem is often referred to as the NN problem.

2.2 All pairs nearest neighbor

This is a special case of the nearest neighbor problem. Say we have a data set, P, for each point $x \in P$ we want to find a point, $r \in P$, where it holds that $\forall p \in P : \|p - x\| \geq \|r - x\|$ and $r \neq x$.

In other words, we want to pair all points in P with its nearest neighbor. Thus, we need to find n nearest neighbors.

Note that if x is y's nearest neighbor, it does not imply that y is x's nearest neighbor.

2.3 K-Nearest neighbor

The k-nearest neighbor problem is defined as finding the k nearest neighbors to q. The most naive algorithm for this problem would be to simply solve the nearest neighbor problem k times, each time removing the point found from the data set.

The k nearest neighbor problem is often referred to as the kNN problem.

2.4 R-Near neighbor

The R-near neighbor problem is the following: Given a set of points, P, and a query point, q, report all points $p \in P$ satisfying $\|p - q\| \leq R$.

The R-near neighbor problem is often referred to as the RNN problem.

2.5 Approximation and probabilistic algorithms

All of the near neighbor problems have approximation variants. These may not find the exact nearest neighbor, but something close.

Probabilistic algorithms exists as well. These solve a given neighbor problem with a given success probability $1 - \delta$.

These variants are interesting because some use-cases may not require an exact answer and the algorithms for solving these problems are often much faster than those applied to the exact variants.

3 Algorithms

Three types of algorithms can be applied to the near neighbor problems:

Linear Search where the data set is searched from one end to the other. If all points are touched in the search we call this algorithm brute force. Different optimizations can be applied such as parallelization and approximation.

Space partitioning where space is split into subspaces recursively. This technique is used by KdTrees, all variants of the R-Trees and many others. KdTrees use hyper-planes to split space. R-Trees encapsulate points in minimum bounding rectangles and thus split the space into rectangles. In general we call this class of trees SP-trees.

Hashing where points with similar features are hashed into the same bucket. Locality Sensitive Hashing does this by sketching vectors and using the sketch as the key in the hash table.

All three approaches will be tested in this thesis.

We assume that all algorithms will be able to fit all data into main memory. No algorithms optimized for secondary storage will be presented.

3

Linear Search

The brute force linear search is the most straight forward way of solving the nearest neighbor problem. The time complexity is $O(dn)$.

For small data sets this approach may be a good solution, with ease of implementation as its primary force.

Even though the linear search is a naive approach, speedups can be made which make the strategy scale to some extend. In general, two approaches can be employed: parallelization or approximation. The linear search is very easy to parallelize. This may be a good solution for a small speedup.

The VA-File [WSB98] is an approximation algorithm utilizing heuristic measures to prune the search. In this way, it saves some of the expensive distance calculations. We have not implemented this approach due to its limited dissemination.

Data domain knowledge can also be used to prune the search. By utilizing e.g. cluster analysis the linear search can gain speedups.

1 Brute Force

The brute force approach is very simple: Calculate all distances to the query point, q, and find the point with the smallest distance to q. This approach has complexity $O(dn)$.

The k-nearest neighbor problem has a complexity of $O(dn+kn)$: calculate n distances and for all distances, check if the distance just seen is among the k smallest seen so far. However, as k approaches n, the time complexity approaches $O\left(n^2\right)$. By sorting the distances we get a complexity of $O(dn + n \log n + k)$: calculate n distances, sort them and output the k first. For large k this will lead to better search times. It is easy to calculate which approach to use.

If we set

$$dn + kn = dn + n \log n + k \tag{3.1}$$

and solve with respect to k we get

$$k = \frac{n \log n}{n - 1} \tag{3.2}$$

If k is smaller than Equation 3.2 the naive approach is better than the sorting approach and visa versa.

2 Parallel Brute Force

The serial brute force approach has an obvious parallel implementation: Given c cores, and n points, let each core calculate n/c distances and let a core pick the point from the c candidates which is closest to the query point. This method has complexity $O(n/c+c)$.

If c is large we want to do better. We cannot do any better in the first step, every core must calculate at least n/c distances. But in the second step we can improve: Let the cores be enumerated $w_1, \ldots, w_c$. If i is odd, w_i sends its sub-result, r_i, to w_{i+1} which holds the sub-result r_{i+1}. w_{i+1} picks the better sub-result and remembers this. We now re-enumerate all cores holding a sub-result and repeat the process. The process is repeated until their is only one result.

In the first round we have c sub-results, in the next $\lceil c/2 \rceil$ and so on. This approach is equivalent to building a balanced binary tree of height $\log_2 c$, with each layer in the tree counting the number of cores working. Thus the method has complexity $O(n/c + \log c)$.

If we have an odd number of cores, the last core simply waits for the next round.

This approach can easily be blocked to optimize transfer time between cores.

3 Implementation

Because CPU scheduling is beyond the scope of Java, it is not possible to assign tasks directly to a core. Instead we can create c threads and assume that if c cores are present, the c threads will be scheduled to their own core by the operating system.

In the parallel strategy each thread calculates $t_s = (t_{id} - 1)(n/c)$ as its starting index and $t_e = t_{id}(n/c)$ as its end index where t_{id} is the thread id and c is the number of cores. Each thread calculate distances for all points $p_i \in P$ with $t_s < i \leq t_e$, where P is the data set. The optimization mentioned above has not been implemented, because our test machine only has two cores.

4 Parallel speedup

Due to limitations on the available hardware, we only test the parallel version of the linear search on two cores. When using two cores compared to one, we would expect performance to double. There is, however, some overhead in the process of spawning threads and combining the sub-results.

The test is conducted by creating a data set of size $n = 100000$, for a set of dimensions $d \in \{1, \ldots, 500\}$ and let the serial and parallel linear algorithms search for 100 nearest

neighbors. We save the average time of the 100 runs. If a_s is the execution time of the serial algorithm and a_p is the execution time of the parallel algorithm, we expect $\frac{a_s}{a_p}$ to converge towards ~ 2 as the number of dimensions rise and the two threads get more work.

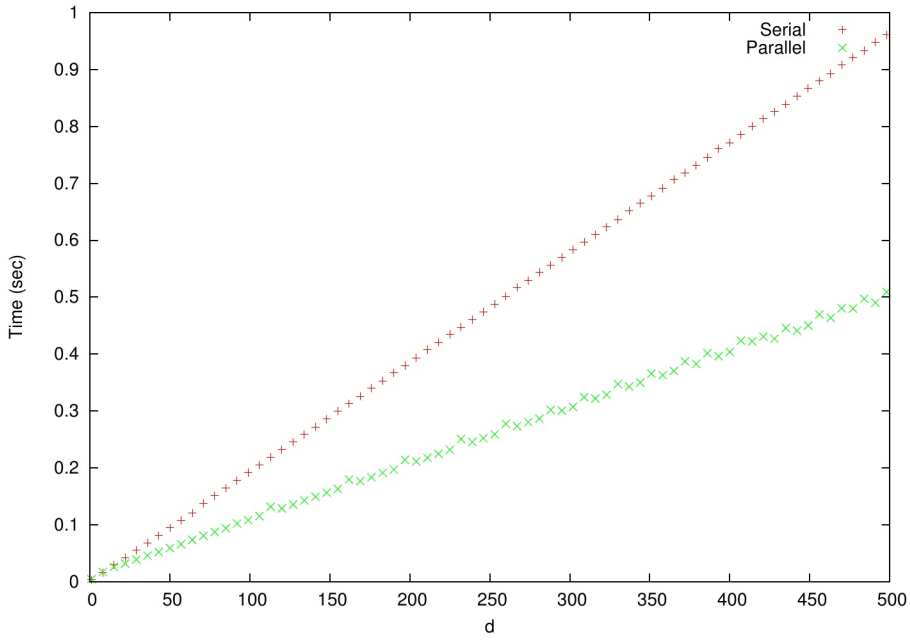


Figure 3.1: The average time to find the nearest neighbor in a d dimensional space with 100000 points. 'Serial' uses one core, 'Parallel' uses two.

The tests have been run on the Mac Mini, described in Appendix A.1.

4.1 Conclusion

When considering Figure 3.1 and 3.2 it is clear that the ratio converges towards ~ 2 as dimensionality grows, as expected.

How good the ratio is, depends on how computationally expensive the task of finding the nearest neighbor is for every core. The harder it is, the better the ratio gets because more time is spend in the worker-threads and less is spend on the overhead of spawning threads and combining sub-results.

5 Cluster analysis

Another way of optimizing the nearest neighbor search, is to utilize data domain knowledge. If we know that data is clustered and we are only interested in data points close to our query point, the following method will speedup the linear nearest neighbor search.

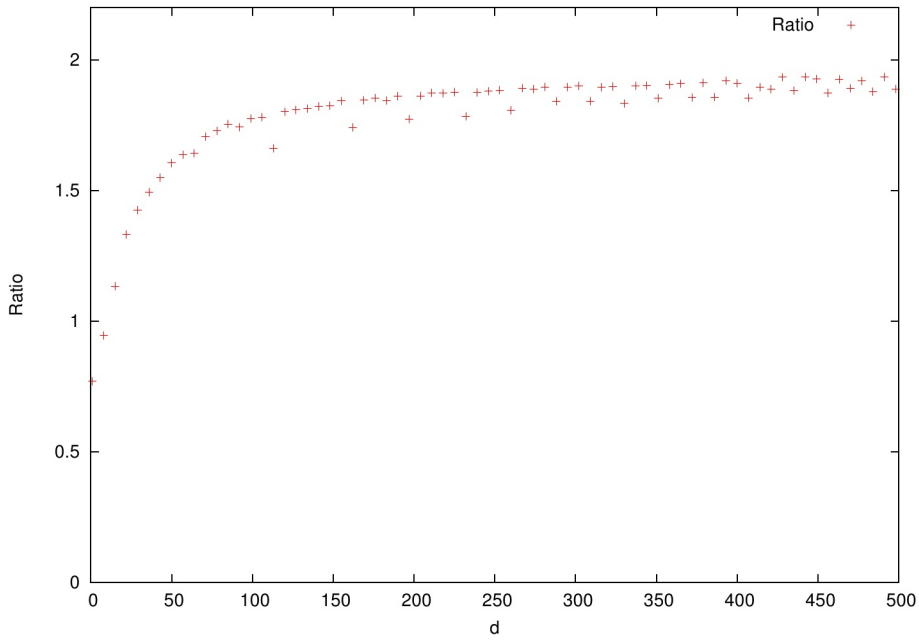


Figure 3.2: The ratio between a single core algorithm versus a dual core algorithm, when dimensionality grows.

5.1 The method

We build our data structure by first identifying each cluster. Then a representative is chosen for each cluster. An obvious choice would be the median on each axis, thus inducing a new point. We save which cluster each point belongs to as well as the representatives.

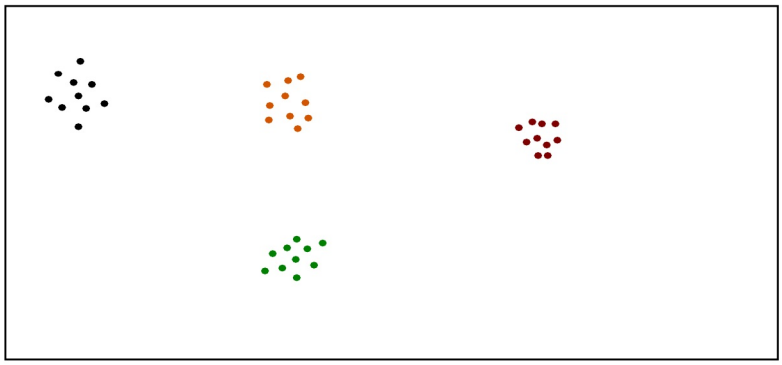


Figure 3.3: Four clusters identified, each with their own color.

Finding the clusters can be done in several ways. If we know that two points in different clusters are separated by at least some constant, s, it is easy to devise an $O(n^2)$ algorithm which calculates distances between all points and splits them into clusters.

Locality Sensitive Hashing can also be used for splitting points into clusters. We present the general algorithm in Chapter 6.

When we search for a nearest neighbor, we first search in the list of cluster representatives. When we have found the representative which is closest to our query point we search through the cluster it represents.

When query points are close to the cluster, compared to the distance between clusters, this method will return the correct result.

If data consists of n_c clusters with roughly n_p points in each, we would expect to examine $O(n_c + n_p)$ data points. This is an improvement over the $O(n_c n_p)$ brute force search.

5.2 Conclusion

This section is merely a demonstration of how data domain knowledge can help to improve search times. This is not in any way a sophisticated technique for nearest neighbor search in general, but for some types of data an efficient search can be devised simply by exploiting properties in the data set.

In a later chapter we will examine how this method stacks up, compared to more sophisticated search methods.

4

KdTree

The KdTree [FBF77] is the natural extension of the binary search tree. It uses space partition for organizing data, by having each node represent a splitting plane, separating points into two subsets and recursively building the structure.

KdTrees was originally presented as an efficient data structure for similarity search, and over time the structure has proven itself in other areas, such as range queries. Currently, however, the KdTree is considered a slow structure for nearest neighbor search in higher dimensions (> 10). Still, compared to more recent structures, it holds an advantage in ease of implementation.

The basic structure of the KdTree has remained the same as presented in the original article. Search methods have been diversified with different heuristics.

1 Structure

The KdTree is a Binary Space Partitioning (BSP) tree, with every non-leaf node, v, consisting of a d-dimensional point, v_p, an axis $1 \leq v_a \leq d$ and two child pointers v_{left} and v_{right}. Every non-leaf node can be thought of as a hyperplane, splitting the d-dimensional space, S, into two subspaces S_{left} and S_{right}, with respect to v_p along the axis v_a.

When creating nodes, the point v_p is chosen so that it is the median in the set S, with respect to the v_a'th coordinate. For the sake of simplicity, we assume that all coordinates are unique. In this way the median is always well-defined. We give a solution to the unrestricted problem in Section 4.2.

From the definition of the nodes, it is clear that every node splits S into two subspaces which induce a split on the body of points, which differ with at most one point in size.

When a subspace only contains one point, a leaf is created.

2 Construction

The construction is carried out recursively as shown in Algorithm 4.1.

Algorithm 4.1: buildKdTree(*input*, *depth*)

if size of *input* = *1* **then**
  $l \leftarrow$ leaf containing last point in *input*;
  return l;
end

$axis \leftarrow depth \mod (dimensions + 1)$;
select *median* with respect to *axis* from *input*;

$v_p \leftarrow median$;
$v_a \leftarrow axis$;
$v_{left} \leftarrow$ `buildKdTree` $(\{e \in input \mid e \leq_{axis} median\}, depth + 1)$;
$v_{right} \leftarrow$ `buildKdTree` $(\{e \in input \mid e >_{axis} median\}, depth + 1)$;
return v;

For two d-dimensional points x and y, the notation $x >_{axis} y$ should be read as: x larger than y with respect to coordinate $axis$. We assume the root is at depth one.

The next thing to be addressed is how to choose the dimension in which the hyperplane splits the space. This is done by cycling through the d possibilities as the depth of the tree increases.

Figure 4.1 represents a KdTree, by drawing the points and splitting planes. The tree is constructed by splitting on the median of the x-axis, then on the median of the y-axis, recursively. When a subspace contains only one point, it is considered a leaf.

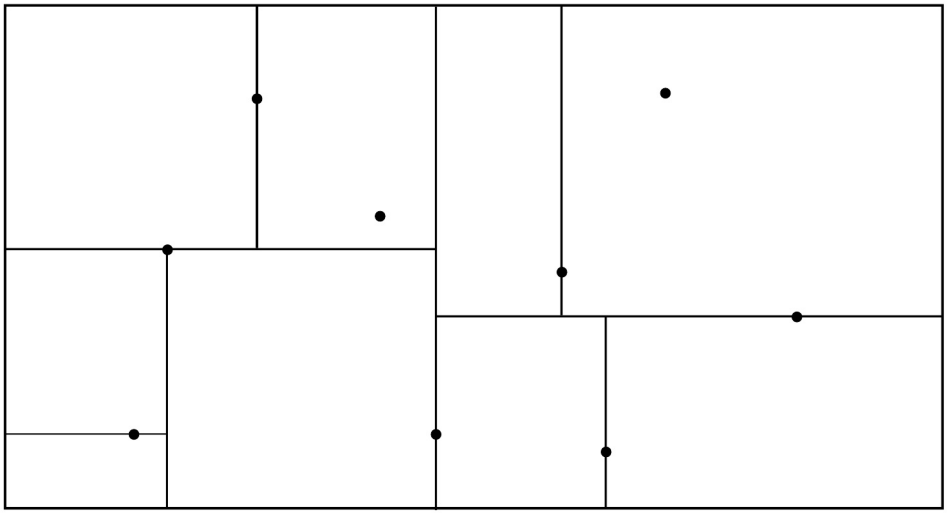


Figure 4.1: Sample KdTree, by drawing the points and splitting planes.

Because the tree is perfectly balanced, the construction time complexity is given by

$$\begin{aligned} T(n) &\in O(n) + 2 \cdot T(n/2) \\ &\in O(n \log n) \end{aligned}$$

2.1 Finding the median

An essential part of building the KdTree is being able to find the median in linear time. This can be done either deterministically or randomized. The Median of Medians algorithm provides a deterministic solution. The Quick Select algorithm will use a randomized approach to the problem by letting us select the $(n/2)$'th element. Both algorithms have linear time complexity [CSRL01].

Both algorithms have been implemented and tested. Section 4.3 covers the details of choice of algorithm. Because the randomized approach was the fastest in our test, it is the algorithm we present. The algorithm is sketched in Algorithm 4.2.

Algorithm 4.2: quickSelect($input, k$)

$p \leftarrow$ a random pivot element from *input*;
split *input* into two subsets *lesser* and *greater* by comparing elements to p;
$l \leftarrow$ size of *less*;
if $l = k - 1$ **then return** p;
else if $l > k - 1$ **then return** `quickSelect`($lesser, k$);
else if $l < k - 1$ **then return** `quickSelect`($greater, k - l - 1$);

The algorithm can be explained as follows:

1. If we have $k-1$ elements that are smaller than p, then p must be the k'th element and we are done.

2. If we have more than $k-1$ elements that are smaller than p, then the k'th element has to be smaller than p, and we look for it in the *lesser* group.

3. If we have fewer than $k-1$ elements that are smaller than p, the k'th element must be larger than p. Thus, we look for the $(k-l-1)$'th element in the *greater* group, because we already have l elements that are smaller than p in the *lesser* group.

The time complexity analysis can be found in [CSRL01, Chapter 9] and [Blu, Thm 4.1].

3 Neighbor Finding

This section will cover the algorithm for nearest neighbor finding and an approximation algorithm for finding a near neighbor.

3.1 Exact search

Nearest neighbor search is carried out in the following manner: Search down through the tree until a leaf is found, while maintaining the best nearest neighbor candidate among the nodes on the search path. On backtracking an implicit hypersphere is drawn, with center in the query point and the best nearest neighbor on the periphery. If the hypersphere intersects the splitting plane of a node, the other branch needs to be checked. See Algorithm 4.3 for the details.

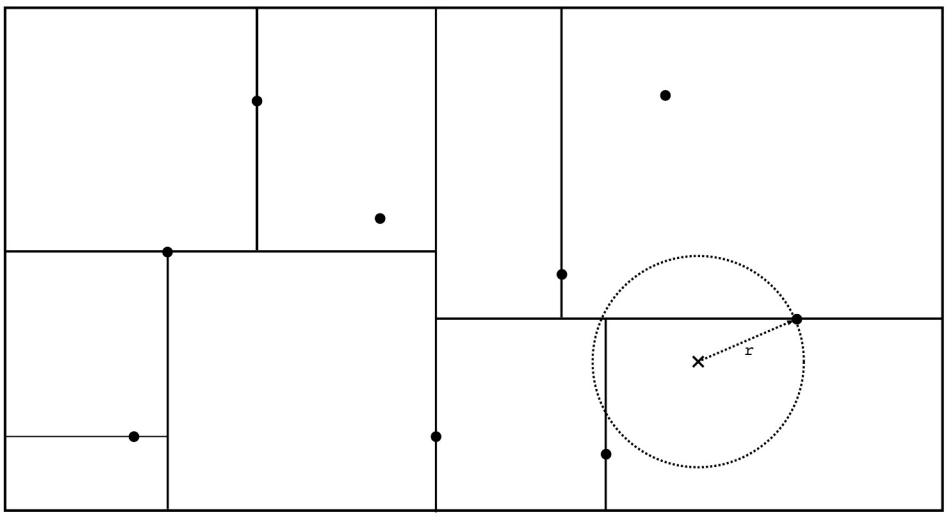


Figure 4.2: Sample KdTree, with a query point resulting in branching to 2 other subtrees.

The example in Figure 4.2 will lead to branching in the search because the bound, r, obtained in the ascend of the search is too large to bound all other branches in the tree. This, however, will not lead to a better solution than one found in the ascend.

The algorithm has a worst-case complexity of $O(n)$, which is reached when every point in the KdTree is placed on a hypersphere with the query point in the center.

The more interesting result is the average-case complexity. The analysis is complex [FBF77], and yields a complexity of $O(\log n)$.

Empirical data from [Moo91], however, show that the efficiency of the nearest neighbor search in a KdTree depends on the number of dimensions. It has been shown theoretically that from some dimension d and on any space partitioning algorithm will degrade to a linear search [WSB98, Conclusion 1].

These two conclusions may seem contradictory, but both results hold. The time complexity of a nearest neighbor search is given by $O(f(d) \log n)$, where $f(d)$ is constant for a fixed d. What [WSB98, Conclusion 1] tells us, is there exists some d where $f(d) \geq n$.

3.2 Approximation

The KdTree can be used for approximate near neighbor search as well. By turning off backtracking, we get an approximation of the nearest neighbor. Other strategies, such

Algorithm 4.3: nnKdTree($v, q, best$)

```
if v = null then return best;
if best = null then best ← v;

distHere ← distance(v, q);
distBest ← distance(best, q);
if distHere < distBest then best ← v

// Search most promising child
axis ← depth mod (dimensions + 1);
child ← (q <_axis v_p ? v_left : v_right);
best ← nnKdTree(child, point, best);

// Search other branch, if necessary
distBest ← distance (best, q);
distSp ← distance to the splitting plane from q;
if distBest > distSp then
    child ← (q ≥_axis v_p ? v_left : v_right);
    best ← nnKdTree(child, q, best);
end
return best;
```

as limiting the search to $c \log n$ nodes for some constant c, can be employed.

When backtracking, the other branches are explored bottom-up, as a natural part of the backtracking. This is not necessarily the best way of doing it. When we search down the tree, a priority queue of other-branches could be maintained. The other branches would be scored according to how promising they seem, according to some objective function. This could be how close the query point is to the splitting plane the other-branch represents. This might lead to a better bound faster and thus a quicker pruning of the search tree.

4 Implementation

4.1 General considerations

The implementation is done in Java 1.5.

The data set is allocated with the type `ArrayList<double[]>`, meaning we have a list object, containing arrays of doubles. This will induce some overhead for the list object. When building the tree, a new list is created for each split (to partition data), leading to n lists at the end of the build. Note that it is only the lists that are replicated, not the arrays they contain, thus the space requirement is kept linear.

No type casting is used. If an object is defined as type A, it will stay that way. Thus, type checking is done at compile time and induces no extra time penalty at runtime.

Nodes in the tree are implemented as objects, containing a point, an axis, 2 children, a boolean deciding if the node is a leaf, and a counter which is used in the tests.

4.2 Dealing with non-unique coordinates

Until now we have assumed that every coordinate in every point was unique. This provided us with the luxury of being able to discriminate points by only looking at one coordinate. This is, however, is not a very realistic scenario.

Say we want to order the points according to coordinate k. If we meet two points with the same values in coordinate k, we simply compare them by coordinate $(k+1) \mod (d+1)$, where d is the number of dimensions of the points. If they do not differ at coordinate $(k+1) \mod (d+1)$ we move to $(k+2) \mod (d+1)$ and so forth.

In other words, we use the lexicographic ordering, starting at coordinate k.

4.3 Choosing a median algorithm

In Section 2.1 we were presented with a choice: Should we use the deterministic Median of Medians algorithm or the randomized Quick Select algorithm?

The test consisted of generating a corpus of n points in 100 dimensions. For each dimension the median was found. Points where generated by assigning each coordinate a value between 0 and 1 at uniform random. In Figure 4.3 the average times for finding the median is presented. The randomized approach seems fastest.

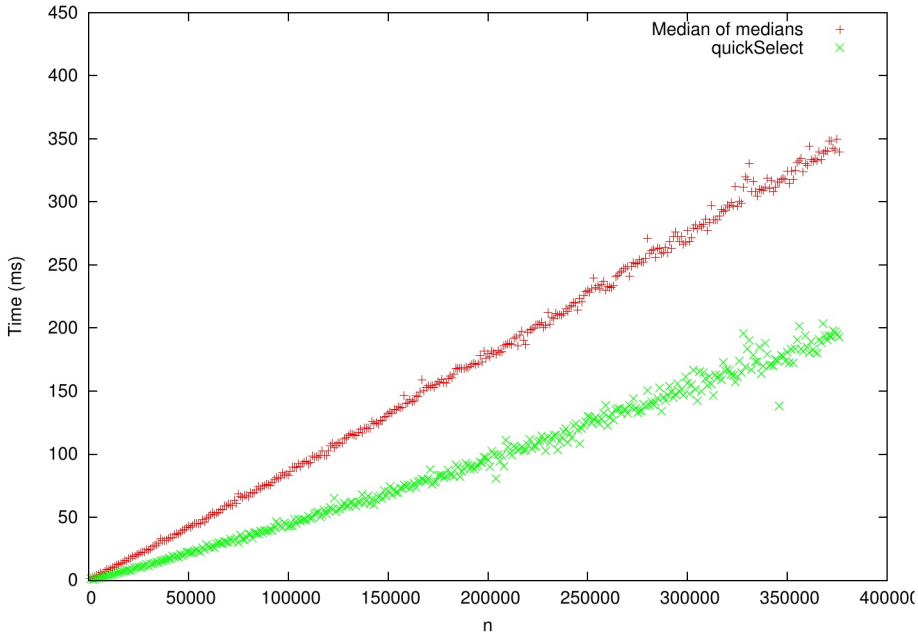


Figure 4.3: The average time to find the median for a given corpus size, in 100 dimensions, on uniform random points.

5 The Curse of Dimensionality

KdTrees suffer from the curse of dimensionality[WSB98]. Intuitively we reckon that as the number of dimensions grow, the need for backtracking in the nearest neighbor search increases, the question is how much.

5.1 Methodology

We test how many nodes we visit in the tree when performing nearest neighbor search. This is proportional to the time it takes to perform the nearest neighbor search.

A collection of data sets has been used:

Uniform: A synthetically generated set of 1000000 points in dimensions $\{1, \ldots, 40\}$, with coordinates in $\{0, \ldots, 1000000\}$. Query points are generated exactly as the data set it self.

Gaussian: A synthetically generated set of 1000000 points in dimensions $\{1, \ldots, 40\}$, with coordinates chosen from $\mathcal{N}(0, 1)$. Query points are generated exactly as the data set it self.

Uniform, close query: A synthetically generated set of 1000000 points in dimensions $\{1, \ldots, 40\}$, with coordinates in $\{0, \ldots, 1000000\}$. Query points are generated so that they are close to some point in the data set, e.i., for a query point, q, there exists some point, p, in the data set so that $\|q - p\|_2 \leq k\sqrt{d}$ for some $k \in \{10, 100, 1000\}$.

In practice this is done by taking a point from the data set and adding a random number $v \in \{0, \ldots, k\}$ to each coordinate, choosing a new v for every coordinate.

For each dimension, we build the tree, query it for 10 points while counting how many nodes are visited by the nearest neighbor search, and save the average number of nodes visited by the 10 searches.

The plots of the results show the percentage of nodes in the tree visited. Thus, if a tree stores n points, it will have a total of $n + (n - 1)$ nodes. If v nodes are visited on average, the graph will show $\frac{v}{n+(n-1)} \cdot 100$.

5.2 Results

We now present the results of the test runs. It should be self evident from the captions what the figures show. The results will be discussed below.

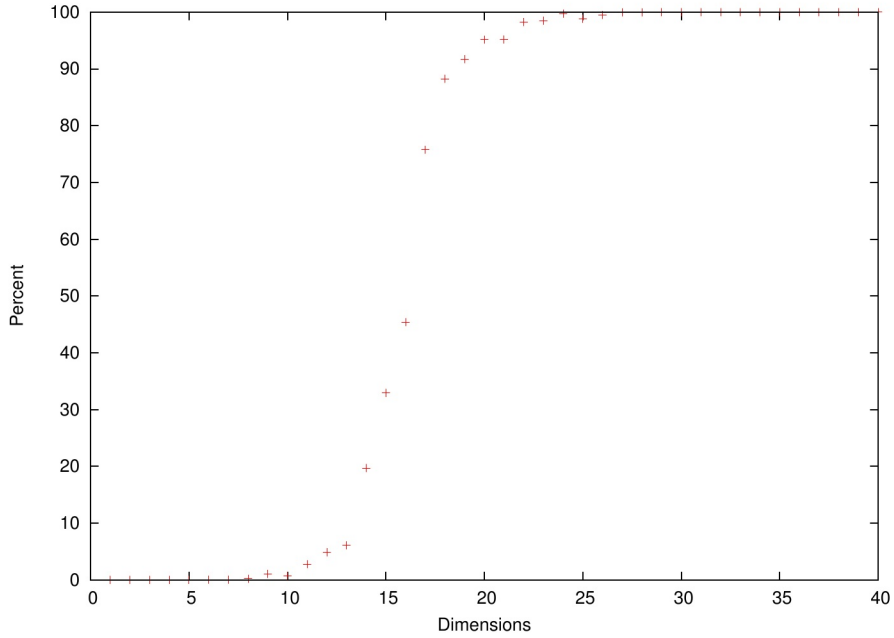


Figure 4.4: The percentage of the tree visited in a nearest neighbor search with data and query points chosen at uniform random in a d dimensional space with 1000000 points.

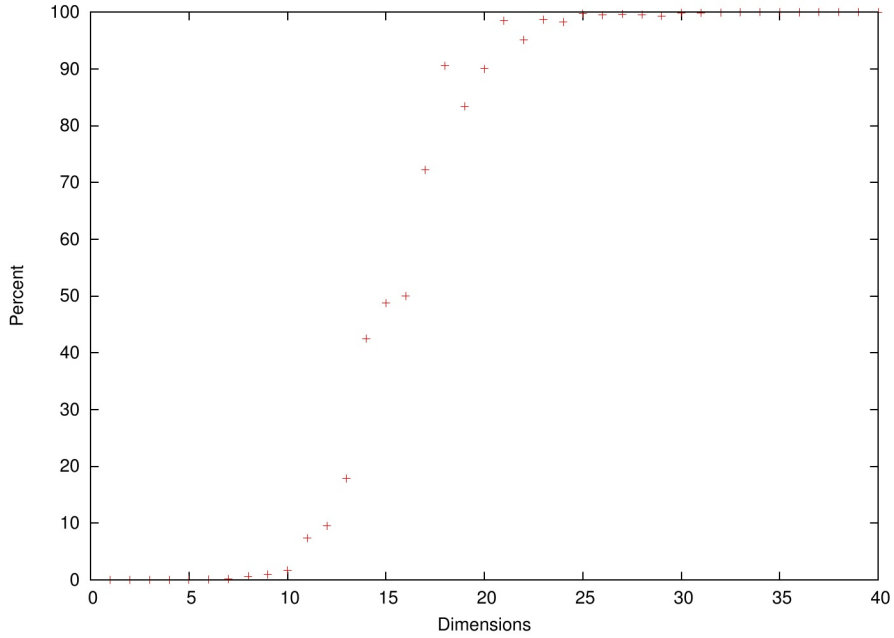


Figure 4.5: The percentage of the tree visited in a nearest neighbor search. Data and query points have coordinates chosen from $\mathcal{N}(0, 1)$.

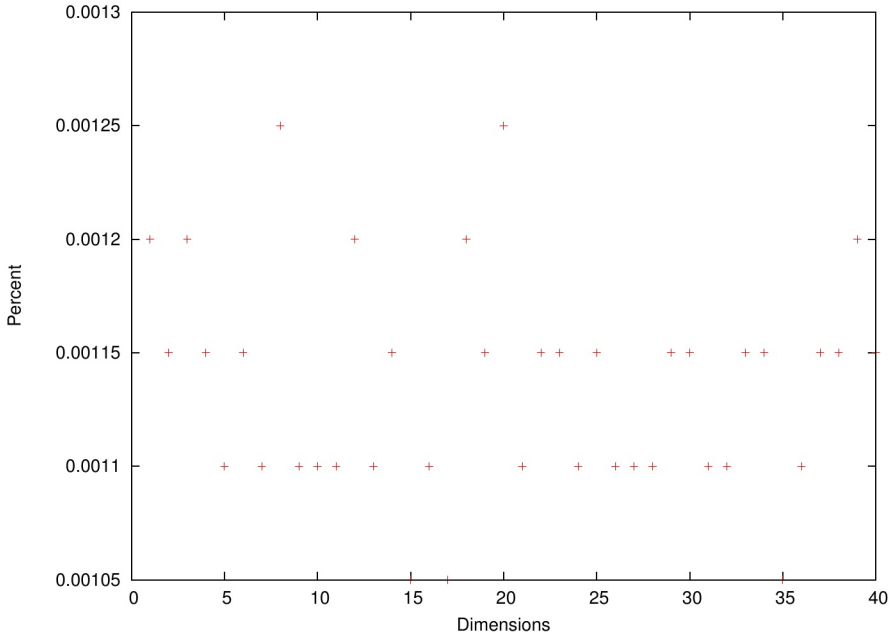


Figure 4.6: The percentage of the tree visited in a nearest neighbor search on uniform random data with query points within a distance of $\sqrt{d} * 10$ to the real nearest neighbor in a d dimensional space with 1000000 points.

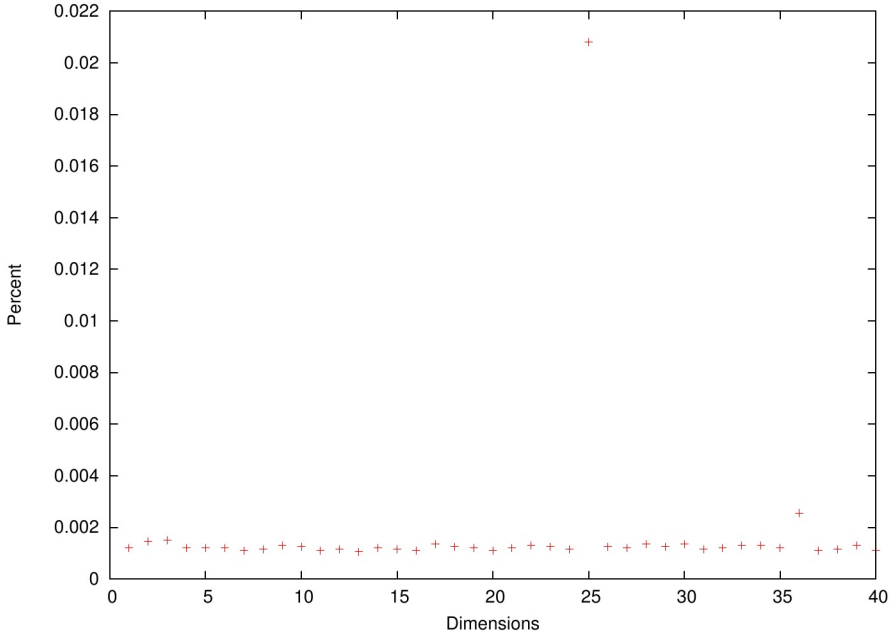


Figure 4.7: The percentage of the tree visited in a nearest neighbor search on uniform random data with query points within a distance of $\sqrt{d} * 1000$ to the real nearest neighbor in a d dimensional space with 1000000 points.

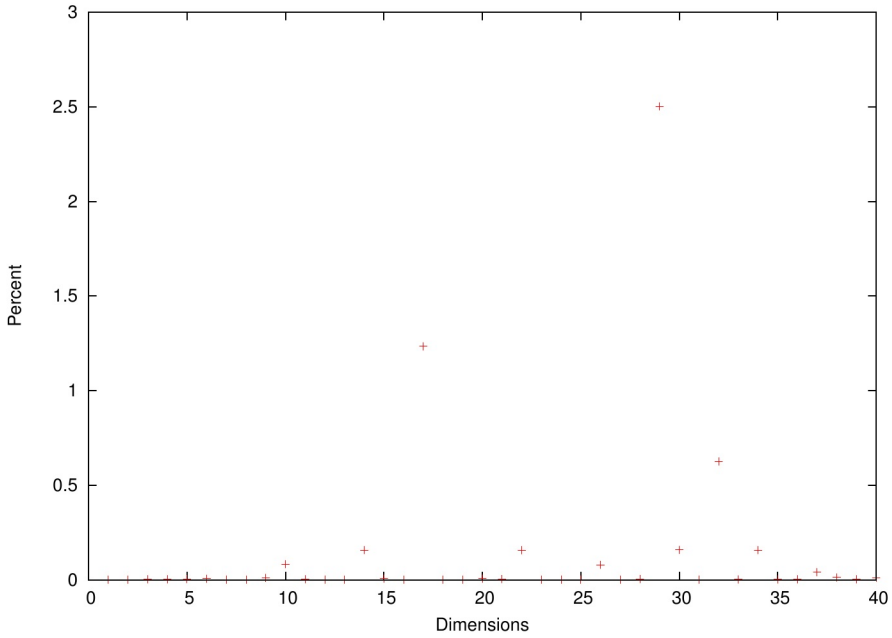


Figure 4.8: The percentage of the tree visited in a nearest neighbor search on uniform random data with query points within a distance of $\sqrt{d} * 10000$ to the real nearest neighbor in a d dimensional space with 1000000 points.

5.3 Conclusion

From Figure 4.4 and 4.5, it is easy to see that when no guarantees are given, KdTrees are useless over 10 dimensions. This result has been shown before in both [WSB98] and [Moo91], we discovered this result had been shown after the tests where performed.

With the promise of query points being within a radius of $k\sqrt{d}$, things seem to change. When query points are close to their nearest neighbor, strong bounds for the backtracking are attained very quickly and backtracking is kept down. This is supported by Figures 4.6, 4.7, and 4.8.

6 Approximation, No Backtracking

To accommodate searches in higher dimensions an approximation strategy is employed. We simply search down the tree while keeping track of which node is closest to the query point. When a leaf is reached, we return the best point found, without any backtracking. This ensures $O(\log n)$ query time, but gives no guarantee on result quality.

6.1 Methodology

The test is performed on a synthetically generated set of 1000000 points in dimensions $\{1, \ldots, 400\}$, with coordinates in $[0 : 1000000]$. Query points are generated exactly as the data set itself. For each dimension, we query for 10 different points and save the

average distance to the points found. This is compared to the distance to the exact nearest neighbor.

When testing approximation algorithms, we defined the ratio as

$$\rho = \frac{alg}{opt}$$

If the algorithm alg is optimal, then $\rho = 1$.

If q is the query point, a is the result found by the KdTree without backtracking and p is the exact nearest neighbor then

$$\rho = \frac{||a - q||_2}{||p - q||_2}$$

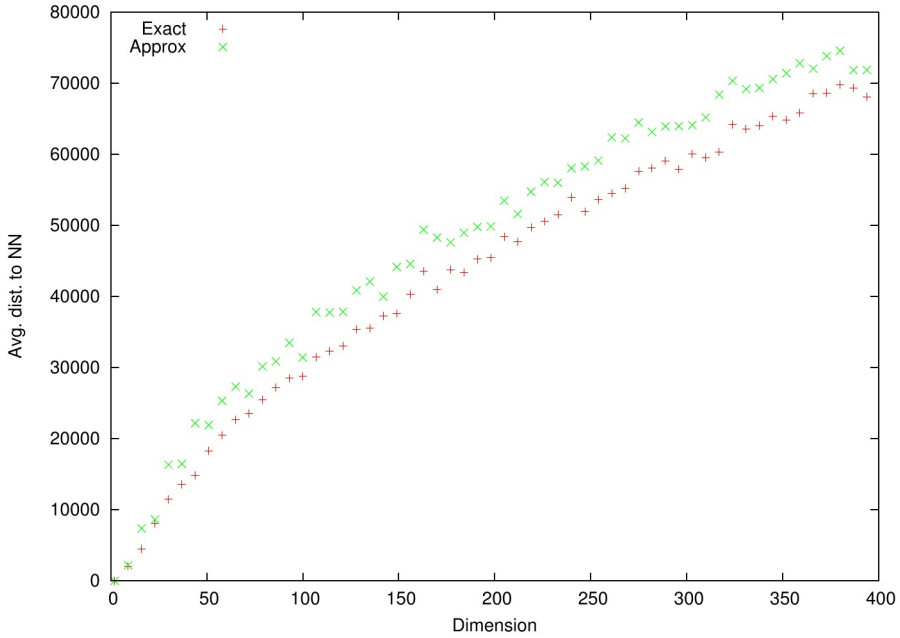


Figure 4.9: Comparison of a KdTree without backtracking (Approx) and a linear search (Exact).

6.2 Conclusion

Contrary to intuition, the approximation sticks close to the exact nearest neighbor. Figure 4.9 shows this.

When looking at the ratio between the approximate and exact result, we see a slight drop in ratio, meaning that the approximation actually improves with dimensionality, in terms of result ratio. Figure 4.10 shows this.

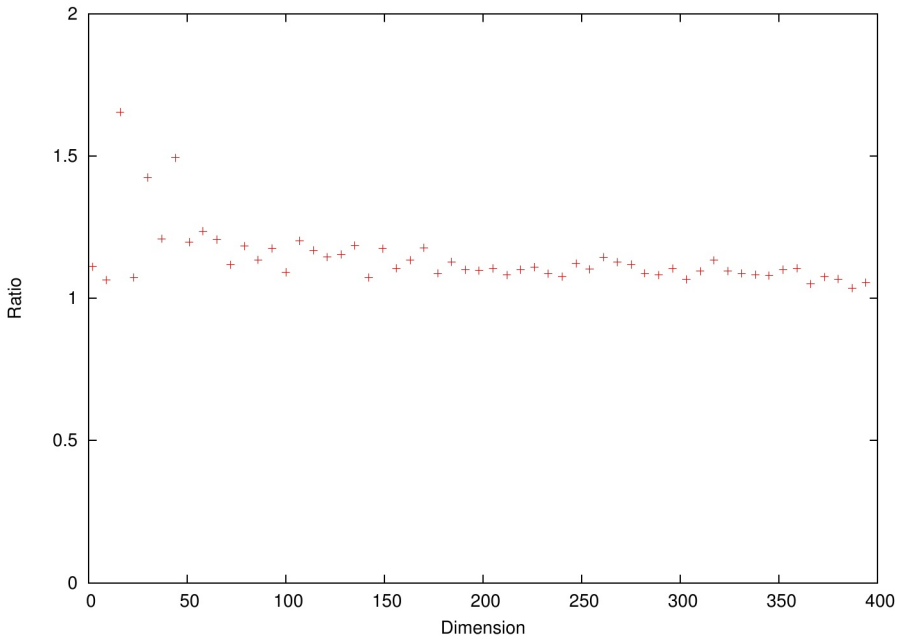


Figure 4.10: The ratio between the result found by the KdTree without backtracking and the exact nearest neighbor.

7 Approximation, Limited Backtracking

The natural extension of the previous section is to see what happens when we limit the backtracking to a number of nodes. Without backtracking we limit the search to $\log n$ nodes. We now test what happens if we limit the search to $c \log n$ nodes, for a given c.

7.1 Methodology

The test is performed on a synthetically generated set of 100000 points in dimensions $\{1, \ldots, 500\}$, with coordinates in $[0 : 100000]$. Query points are generated exactly as the data set itself. For each dimension, we query for 100 different points and save the average distance to the points found. This is compared to the distance to the exact nearest neighbor. We test $c \in \{5, 10, 100, 200, 300, 500\}$.

We only plot the ratio, as it will show us if we get significantly better results. We define ratio as in the previous section.

The test has been performed on the Mac Mini described in Appendix A.1.

7.2 Conclusion

From Figure 4.11 it is clear that the extra backtracking helps keep the ratio stable, but the results are not significantly better.

From Figure 4.12 we see that the query times are exactly proportional to d, as dimensionality grows. This is expected, as c can be considered a constant for a given

tree.

Because the distance between points in general grow faster than the distance from query point to nearest neighbor, we get a drop in ratio as dimensionality grows.

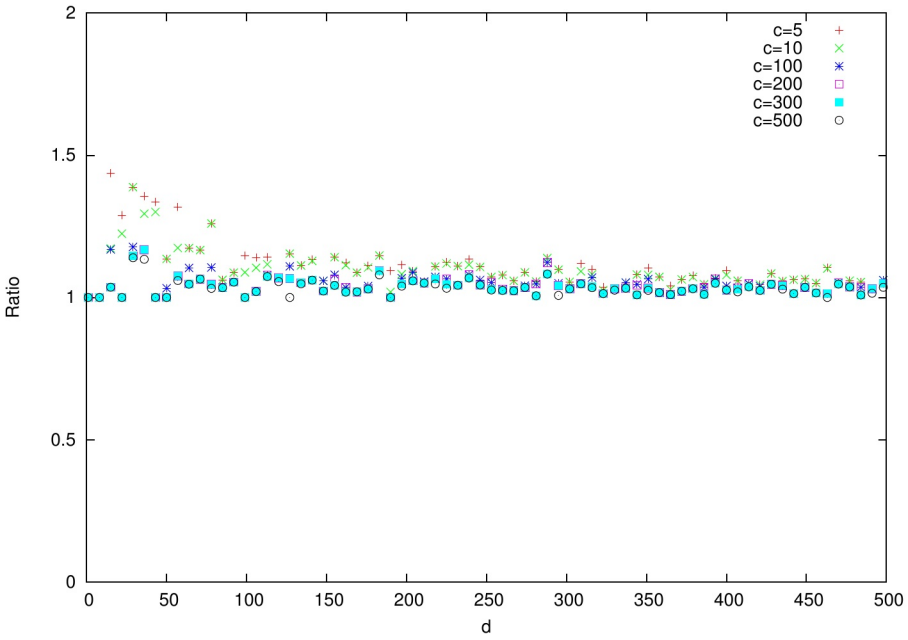


Figure 4.11: The ratio between the result found by the KdTree with backtracking limited to $c \log n$ nodes and the exact nearest neighbor.

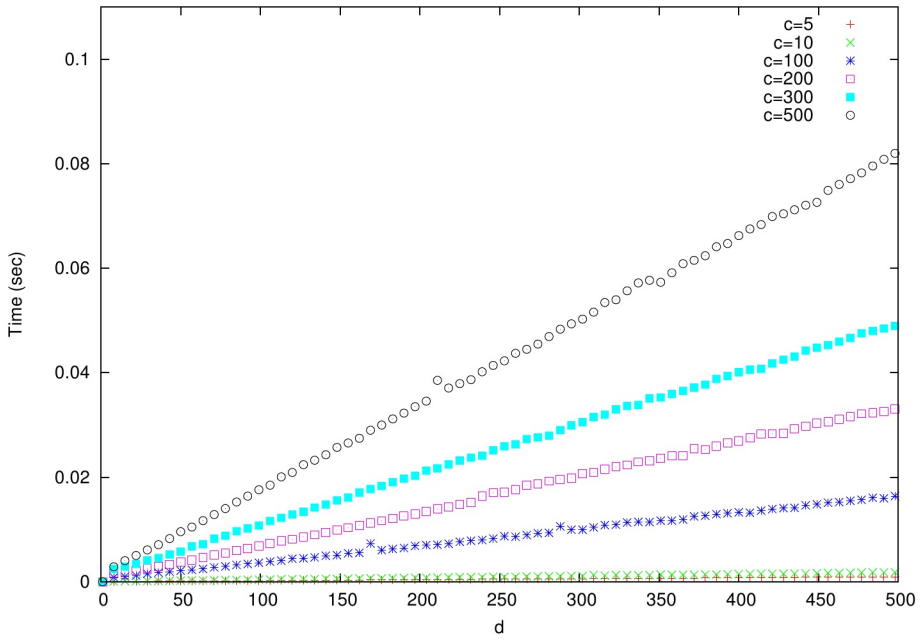


Figure 4.12: Comparison of the search times for a KdTree with backtracking limited to $c \log n$ nodes.

5

R-Tree

The R-Tree[Gut84] is the preferred choice of data structure for spatial access methods in many applications. A number of variants with different properties has been developed to accommodate various needs, primarily better insertion properties.

All R-Tree variants have a minimum bounding rectangle (MBR) in every internal node which encloses a set of child rectangles. Points are stored in the leaves.

R*-Trees support all the same operations as a regular R-Tree with a slightly higher cost for insertion. The more aggressive insertion algorithm results in less overlap between MBRs and thus better space utilization.

Hilbert R-Trees optimize the R-Tree for spatial objects such as lines, regions or high dimensional feature objects. In this R-Tree variant, the Hilbert curve is used to impose a linear ordering on the data objects.

Nearest neighbor search has not changed much over time. The first algorithm was presented in [RKV95]. It utilized the optimistic (MINDIST) and pessimistic (MINMAXDIST) distance measure from a query point to an MBR in order to prune the nearest neighbor search in the tree. In [KL97], a simplified version of the search algorithm was presented. The simplified algorithm avoided the MINMAXDIST distance calculations both simplifying the algorithm and improving query time ($\sim 10\%$), in terms of nodes touched in the search.

Insertion of nodes can either be considered an online problem where nodes have to be inserted one by one or as an offline problem where all nodes are considered as a bulk. We call the offline version bulk loading. bulk loading can improve the quality of the tree, where the quality is measured by some given evaluation function. This function could reflect the area covered by the MBRs, overlap of MBRs, amongst others. The most well known algorithms for bulk loading R-Trees are Sort Tile Recursive (STR) [LEL97] and Top-Down Greedy Split (TGS) [GLL98].

1 Structure

Every internal node in the R-Tree contains an MBR enclosing all child MBRs and a set of children. Like B-Trees, the R-Tree can have an arbitrary large outdegree in the nodes, which remains constant for a specific tree. The MBR is minimal in the sense that its area cannot be reduced without leaving out some of the area covered by a child MBR. Leaves contain an MBR as well as pointers to the points contained in the node.

MBRs may overlap, even though every point only exists in one sub-tree. Overlap occurs when MBRs cannot be split into equi-sized, non-overlapping partitions. An example of this could be splitting the MBRs in Figure 5.1 into two equi-sized partitions. No matter how we partition the MBRs, the resulting parent MBRs will overlap.

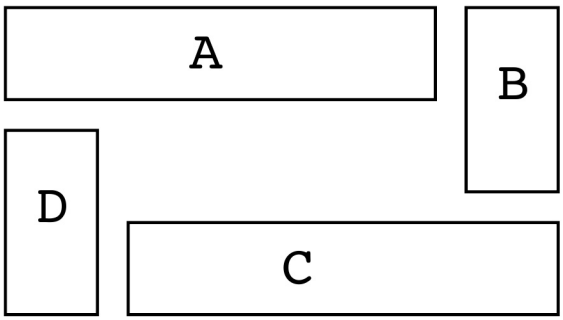


Figure 5.1: Four MBRs that cannot be split into 2 equi-sized partitions, without overlapping parent MBRs.

Consider Figure 5.2 for a sample R-Tree structure with outdegree three and depth one.

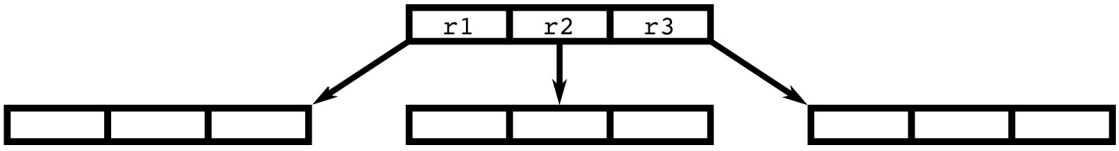


Figure 5.2: The R-Tree tree structure width outdegree three.

The root contains an MBR enclosing the three leaves and pointers to the leaves. Each leaf contains an MBR and pointers to the data points (not depicted). The MBRs could be placed as shown in Figure 5.3.

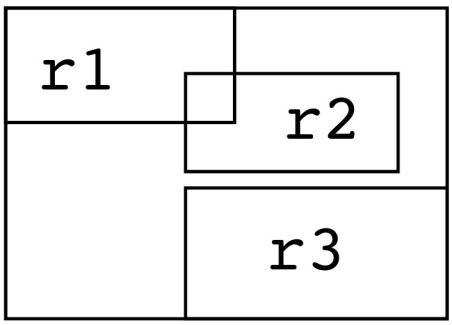


Figure 5.3: Structure of the rectangles in the plane.

2 Nearest-Neighbor Search

R-Trees support both exact and approximate nearest neighbor search. In this section an algorithm for exact nearest neighbor search will be presented.

There are two exact algorithms for nearest neighbor search in R-Trees. The first was presented in [RKV95] and will not be covered here. We cover the simplified version, as presented in [KL97].

2.1 Simplified nearest neighbor search

First, a definition of the optimistic distance measure between a query point, q, and an MBR, R, is needed.

Definition 5.1 (MINDIST) *The MINDIST between a query point, q, and an MBR, R, is defined by*

$$MINDIST(q, R) = \sqrt{\sum_{i=1}^{d} (q_i - r_i)^2} \tag{5.1}$$

where d is the number of dimensions of q and R, q_i is the coordinate of the point q in the i'th dimension and

$$r_i = \begin{cases} s_i & \text{if } q_i < s_i \\ t_i & \text{if } q_i > t_i \\ q_i & \text{else} \end{cases} \tag{5.2}$$

where s_i and t_i are the minimum and maximum values of the i'th dimension of the rectangle, R.

The MINDIST is the smallest possible distance between a point, q, and an MBR, R. There is no child rectangle or point in R that can have a smaller distance to q than the MINDIST. If the point q is inside R, the MINDIST is 0. Note that the original definition

in [KL97] did not include the square root on the distance, as one in practice would not do this.

The MINDIST can be summed up by Figure 5.4.

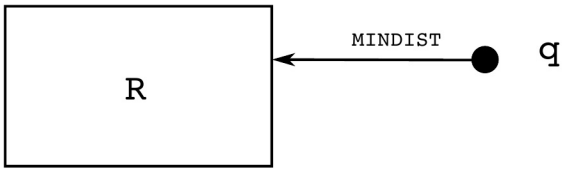


Figure 5.4: An example of the MINDIST measure. The rectangle cannot contain anything which is closer to q than this.

The search for a nearest neighbor is carried out as presented in Algorithm 5.1.

Algorithm 5.1: RTree-NN-Search(q, *root*)

plist $\leftarrow$ all children of *root*;
while *plist not empty* **do**
  expand $R \in$ *plist* with smallest $MINDIST(q, R)$;
  if *a leaf is reached* **then**
    update *best* if needed;
  end
  prune away every $R \in$ *plist* with $MINDIST(q, R) > \|q - best\|_2$;
end
return *best*;

First, all children of the root is added to a priority list. Now, as long as the list contains more rectangles, which has MINDIST less than the closest point we have found so far, add the children of the rectangle with the smallest MINDIST to the priority list. As we find leaves, update the best point found so far. In practice it is faster not to prune the priority list, but simply stop the search when there are no rectangle in the list with smaller MINDIST than the best point found.

If many rectangles overlap, rectangles will be expanded even though they may not contain a nearer neighbor. This problem, however, is attributed to the construction of the tree, not the nearest neighbor search.

To speed up the search, a greedy depth first search can be used to get a bound, thus starting the search with a reasonable bound.

The greedy depth first search strategy can also be used if the goal is to find an approximate nearest neighbor.

3 Bulk loading

Because the quality of the tree affects the query time, bulk loading can induce speed-ups. In general, bulk loading algorithms have the following scheme for a tree consisting of n rectangles with an outdegree of c:

1. Pre-process data so that rectangles are ordered in $\lceil n/c \rceil$ consecutive groups of c rectangles. Each group is to be placed under the same node.

2. Build the $\lceil n/c \rceil$ nodes with their children.

3. Recursively build a tree of the nodes just created.

The rectangles can be ordered in a number of ways:

Hilbert Sort: The ordering is imposed by the distance along the Hilbert curve from the origin of the curve to the center of the rectangle.

Sort Tile Recursive: The ordering is imposed by tiling where rectangles are sorted on the center coordinate.

There are, however, other options. The top-down greedy splitting algorithm attacks the problem top-down, in a more aggressive way.

3.1 Top-down greedy splitting

The top-down greedy splitting (TGS) algorithm[GLL98] partitions input data in a top-down fashion, as partitions close to the root are likely to have the most impact on the performance of the nearest neighbor search. At each level, the algorithm considers all cuts orthogonal to the coordinate axis, which results in a balanced tree and greedily selects the ones that optimize a user supplied cost function.

TGS is based on the basic split step which partitions a set of n rectangles into two sets by using a cut orthogonal to an axis, so that the sum of costs, with respect to the cost function, is minimal. Candidate cuts are constrained so that one side of the cut has a multiple of c rectangles to guarantee that the resulting subtrees are packed. The process is then applied to both parts until the desired number of partitions have been created.

The setup is as follows: n is the number of rectangles to be split. Let $M = c^{\lceil log_c n \rceil - 1}$ be the number of rectangles in each of the c sub-trees. $f(r_1, r_2)$ is the user supplied cost function measuring the cost of the split (r_1, r_2).

We are now ready to consider Algorithm 5.2, the basic split step.

Orderings considered could be { min. coord., max. coord., both, center coord. }.

In this manner we split the data points into c sets in every internal node.

Algorithm 5.2: TGS-BasicSplitStep(*input*)

```
if n < M then Stop
foreach dimension d do
    foreach ordering considered in this dimension do
        for i ← 1 to (⌈n/M⌉ − 1) do
            Remember i if a cut in the current order after position i · M results in
            a better value of f(B0, B1), where B0, B1 is the MBRs of the
            rectangles before and after the cut in question.
        end
    end
end
```

4 Implementation

The implementation is done in Java 1.5.

The data set is allocated with the type `ArrayList<double[]>`, meaning we have a list object, containing arrays of doubles. This will induce some overhead for the list object. When building the tree, a new list is created for each split (to partition data), leading to n/c lists at the end of the build. Note that it is only the lists that are replicated, not the arrays they contain, so the space requirement is kept linear.

No type casting is used. If an object is defined as type A, it will stay that way. Thus, type checking is done at compile time and induce no extra time penalty at runtime.

Nodes in the tree are implemented as objects containing a list of children, an MBR, a boolean deciding if it is a leaf and a point (if it is a leaf) along with a bit of metadata about the point.

We let TGS consider the orderings $\{min, max\}$ as considering both and center coordinate did not lead to better query times.

5 The Curse of Dimensionality

Like KdTrees, we know that the R-Tree suffers the curse of dimensionality [WSB98]. We now set out to test just how vulnerable the R-Tree is to rising dimensionality.

All tests have been performed on the Mac Mini described in Appendix A.1.

5.1 Methodology

We used a synthetically generated data set of 100000 points in dimensions $\{1, \ldots, 50\}$, with coordinates in $\{0, \ldots, 100000\}$. Query points are generated exactly as the data set itself.

For each dimension, we build the tree with TGS, query it for 10 points while timing the nearest neighbor search, and save the average time of the 10 searches.

We test different outdegrees for the R-Tree, $c \in \{2, 5, 7, 10\}$, in order to see if it makes any difference in performance. When data reside on disk, outdegree can have a huge impact, but it is not clear if it is as important when data reside in main memory.

With data residing on disk, outdegrees can range up to 200 in order to match the block size, but with data in main memory it is the cache line size we want to match. Thus, a much smaller outdegree makes sense.

For each dimension, we have used a linear search to find the nearest neighbor in exactly the same data set. We have plotted the time for the linear search as well, for comparison.

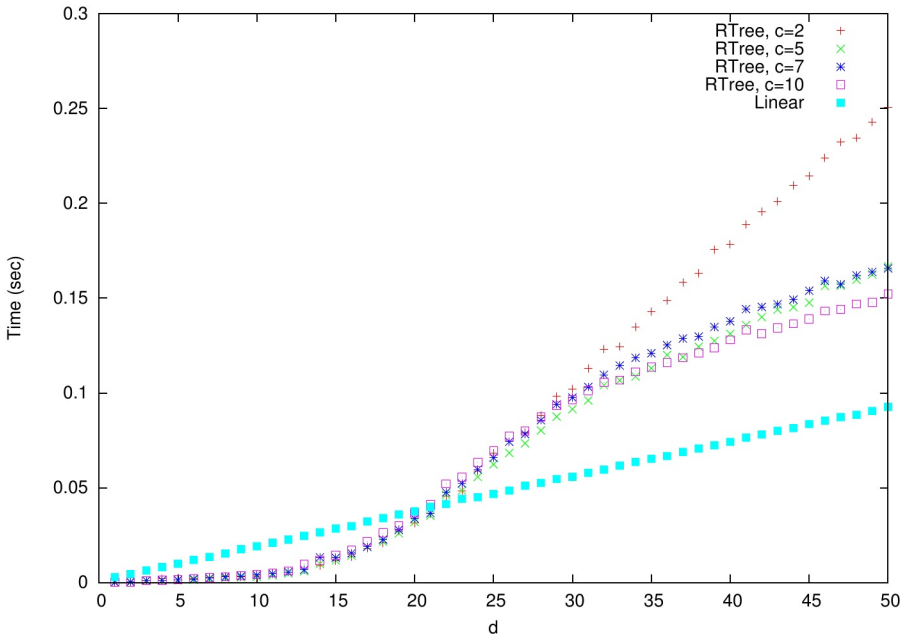


Figure 5.5: Comparison of an R-Tree with outdegree, c, and a linear search.

5.2 Conclusion

Consider Figure 5.5 to see the result. It is clear that the R-Trees degrade to a linear search when $d > 30$.

From the test it seems that R-Trees are impractical for more than around 15 dimensions. Like KdTrees, however, they start to degrade around 10 dimensions.

6 Build times

Intuitively, the TGS bulk loading algorithm has a much higher build cost than the KdTree build algorithm.

Thus, we need to examine the time complexity of the TGS bulk loading algorithm from a theoretical stand point. We shall establish if the TGS algorithm has a higher

time complexity than the KdTree build algorithm with respect to the size of the data set.

Secondly, we test the TGS algorithm empirically to see how it performs.

6.1 Time complexity

The time complexity for the TGS algorithm, with respect to n, is given by:

$$T(n) \in O(d) \cdot \#orderings \cdot O\left(2^{c-1}\right) \cdot O(dn) + O(dn) + (c-1) \cdot T\left(\frac{n}{c-1}\right) \tag{5.3}$$

$$\in O(n) + (c-1) \cdot T\left(\frac{n}{c-1}\right) \tag{5.4}$$

$$\in O(n \log n) \tag{5.5}$$

For each dimension and for each ordering, we try all cuts, test which is better and split data. We can omit both d, $\#orderings$, and $O\left(2^{c-1}\right)$, from the complexity analysis because these are constant for a given tree. We also assume that we, prior to the splitting process, have sorted the data set according to all d coordinates and saved the results in d lists. Thus, enabling us to test a split and split data without having to sort the data set and thereby reducing the time for the test and splitting from $O(n \log n)$ to $O(n)$.

Despite the time complexity being $O(n \log n)$ we should, however, not neglect the fact that $O(d^2)$ and $O\left(2^{c-1}\right)$ will grow fast and affect the performance when d and c grow.

6.2 Testing methodology

A synthetically generated data set of 100000 points in dimensions $\{5, 15, 25, \ldots, 255\}$, with coordinates in $\{0, \ldots, 100000\}$ has been used. We test different outdegrees for the R-Tree, $c \in \{2, 5, 7, 10\}$, to see if it makes any difference in performance

6.3 Conclusion

Consider the results in Figure 5.6. From the results it is easy to see that the TGS bulk loading algorithm is heavily affected by the number of dimensions and the outdegree of the nodes.

When we stopped the tests at 245 dimensions, the build times where at $\sim$ 11000 seconds or $\sim$ 3 hours and rising fast for outdegree 10. If we fit a 2-degree polynomial to the construction times of the R-Trees with outdegree 10, we can extrapolate that the build time for 500 dimensions would be $\sim$ 13 hours. From sampling, we know this holds. The slow build times render testing of the R-Trees in high dimensions very time consuming.

Figure 5.7 shows the fitting used.

The fitted polynomial is given by

$$f(x) = 0.187838x^2 + 5.5851x - 5.5997 \tag{5.6}$$

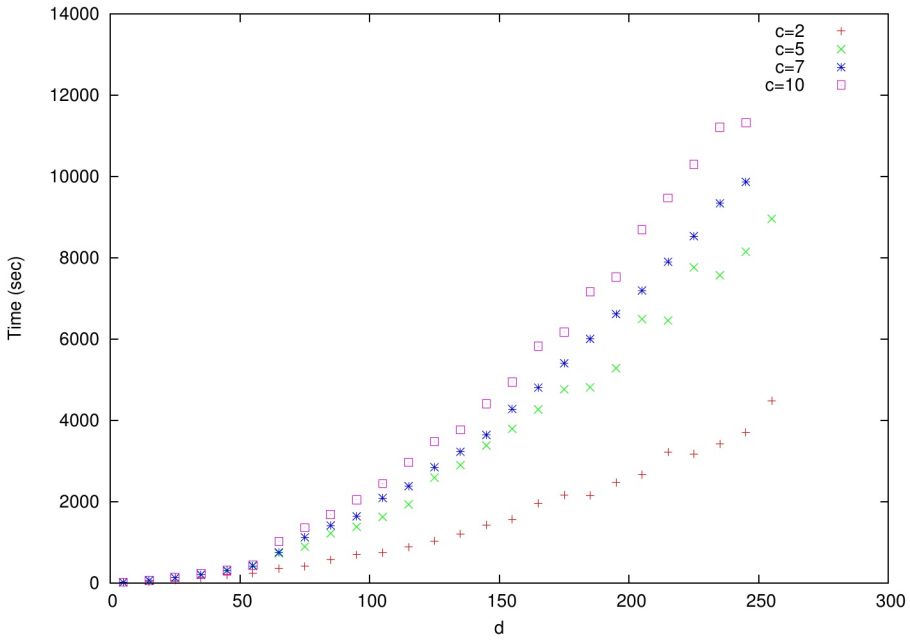


Figure 5.6: Comparison of an R-Tree construction times with outdegree, c.

We used a 2-degree polynomial, because the time complexity of the TGS algorithm is $O\left(d^2\right)$, when we assume the time complexity as a function of d.

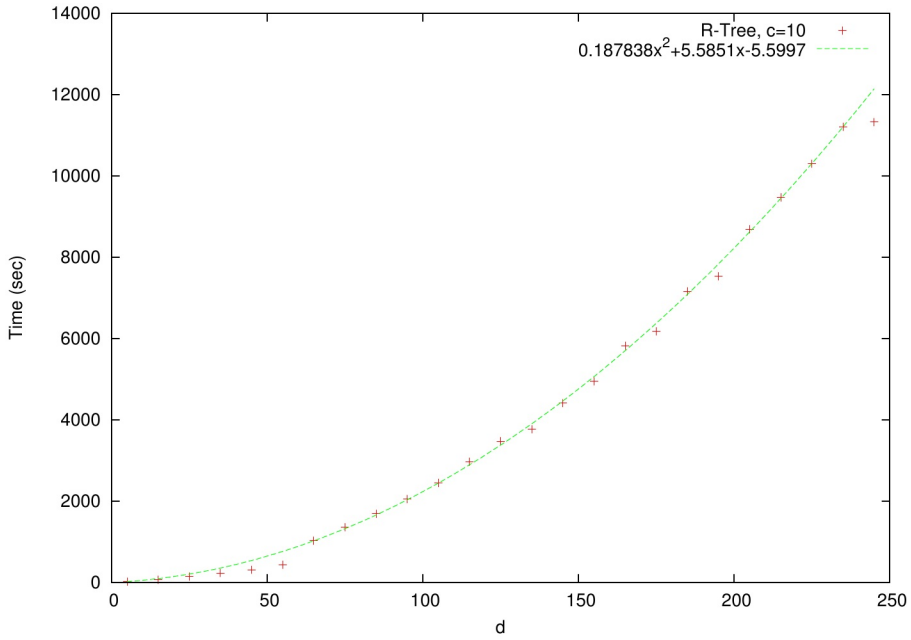


Figure 5.7: Fitting a 2-degree polynomial to the construction time of the R-Tree with outdegree 10.

6

Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) is a data structure providing $O(do)$ R near neighbor query time [SDI06], where d is the number of dimensions and o is the number of candidate near neighbors to a query point. The construction time is $O(nLkd)$, where n is the size of the data set, L and k are parameters to the data structure.

By using random projections to sketch vectors, the data structure is able to reduce the dimensionality from d to some constant k. The data structure does this a number of times, in effect creating L data structures.

The structure does not inherently solve the nearest neighbor problem, but instead solve the R near neighbor problem.

1 The General Scheme

We will now present the skeleton of the LSH scheme.

1.1 The Setup

First, we give the definition of a locality sensitive hash function as presented in [AI08]. Consider a family, $\mathcal{H}$, of hash functions mapping $\mathbb{R}^d$ to some universe U.

Definition 6.1 (Locality Sensitive Hash Function) *A family $\mathcal{H}$ is called (R, cR, P_1, P_2)-sensitive if for any $p, q \in \mathbb{R}^d$ it holds that*

1. *if $\|p - q\| \leq R$ then $Pr_{\mathcal{H}}[h(q) = h(p)] \geq P_1$*
2. *if $\|p - q\| \geq cR$ then $Pr_{\mathcal{H}}[h(q) = h(p)] \leq P_2$*

The definition gives us a guarantee: when using a locality sensitive hash function, we know that

1. if 2 elements, $p, q \in \mathbb{R}^d$, are close (with respect to R) their hash values $h(p)$, $h(q)$ will collide with a probability of at least P_1.

2. if 2 elements, $p, q \in \mathbb{R}^d$, are far (with respect to cR) their hash values $h(p)$, $h(q)$ will collide with a probability of at most P_2.

To be useful, a family $\mathcal{H}$ must satisfy $P_1 > P_2$. However, even if $P_1 > P_2$, it does not guarantee that the gap $P_1 - P_2$ is large. Therefor, an amplification process is usually applied.

1.2 The amplification process

First, a new compound hash function is defined:

$$g(q) = \langle h_1(q), h_2(q), \ldots, h_k(q) \rangle \tag{6.1}$$

Assuming that $h_i \in \mathcal{H} : \mathbb{R}^d \mapsto \mathbb{Z}$ then $g : \mathbb{R}^d \mapsto \mathbb{Z}^k$. If $h_i \in \mathcal{H}$ maps from a vector of real numbers to an integer, g maps from a vector of real numbers to a vector of integers. Thus, to get a collision in g, we need a collision in all $h_1, \ldots, h_k$.

Second, an array of g functions are made, so now the setup is

$$g_i(q) = \langle h_1(q), h_2(q), \ldots, h_k(q) \rangle \text{ for } 1 \leq i \leq L \tag{6.2}$$

That is, we have k inner hash functions from the $\mathcal{H}$ family, and L compound hash functions.

The amplification process obviously affects P_1 and P_2. The probabilities are affected in the following manner:

$$Pr[\text{collision in any } h \in g_i] = P^k \tag{6.3}$$

$$Pr[\text{no collision in any } h \in g_i] = 1 - P^k \tag{6.4}$$

$$Pr[\text{no collision in any } g_i] = (1 - P^k)^L \tag{6.5}$$

$$Pr[\text{collision in any } g_i] = 1 - (1 - P^k)^L \tag{6.6}$$

Both P_1 and P_2 are affected in this manner.

2 LSH Using Stable Distributions

In this section, a family of locality sensitive hash functions, as described in [SDI06], will be presented. The family is defined where the distances are measured according to the l_s norm for any $s \in \{1, 2\}$. The case of $s = 1$ will not be treated explicitly.

2.1 s-stable distributions

The most well known stable distribution is the normal distribution, but many other exist, including heavy-tailed distributions.

An s-stable distribution is defined as follows [SDI06]

Definition 6.2 (s-stability) *A distribution $\mathcal{D}$ over $\mathbb{R}$ is called s-stable, if there exists $s \geq 0$ such that for any n real numbers $v_1, \ldots, v_n$ and i.i.d. variables $X_1, \ldots, X_n$ with distribution $\mathcal{D}$, the random variable $\sum_i v_i X_i$ has the same distribution as the variable $(\sum_i |v_i|^s)^{1/s} X$ where X is a random variable with distribution $\mathcal{D}$.*

It is known that the Gaussian distribution, defined by the density function $g(x) = \frac{1}{\sqrt{a\pi}} e^{-x^2/2}$, is 2-stable [Zol86]. Despite the lack of closed form density and distribution functions, it is known that s-stable random variables essentially can be generated from two independent variables distributed uniformly over $[0, 1]$ [CMS76].

What the definition reveals is a method of sketching a vector. Given a vector, v, in d dimensions, we generate a vector, a, also in d dimensions with each entry chosen independently from an s-stable distribution. The dot product $a \cdot v$ is a random variable distributed as $(\sum_i \|v_i\|^s)^{1/s}$ i.e. $\|v\|_s X$ where X is a random variable with an s-stable distribution. A collection of such dot products with different a's is called a sketch of a vector, which can be used to estimate $\|v\|_s$ [Ind00].

2.2 Hash family based on s-stable distributions

The family of hash functions presented in this section utilizes the idea of vector sketching. This is done by using the dot product $a \cdot v$ to map vectors onto the real line. The real line is then chopped into equi-width segments of size w. Finally, hash values are assigned to vectors based on which segment they are projected onto.

A hash function $h_{a,b}(v) : \mathbb{R}^d \mapsto \mathbb{Z}$ maps a d-dimensional vector, v, onto the set of integers. Each hash function is indexed by a, b where a is chosen as before and b is a number chosen uniformly from the range $[0 : w]$. For a fixed a and b, $h_{a,b}$ is given by

$$h_{a,b}(v) = \left\lfloor \frac{a \cdot v + b}{w} \right\rfloor \tag{6.7}$$

From now on, we drop the a, b-subscript and simply enumerate hash functions when necessary.

Collision probability

We can calculate the probability of a collision between two points v_1, v_2, with $||v_1 - v_2||_2 = u$, in a hash function h.

Say we project the two points onto the real line, what is the probability that $a \cdot v_1 - a \cdot v_2 \leq w$? The difference between two vectors dotted with a vector from an s-stable distribution, X, will distribute as uX. Thus, we get

$$Pr(a \cdot v_1 - a \cdot v_2 \leq w) = Pr(uX \leq w) = Pr(X \leq w/u) = \int_0^{w/u} f_s(t)\, dt \tag{6.8}$$

$f_s(t)$ denotes the probability density function of the absolute value of the s-stable distribution.

The variable t can be seen as a line segment, which we want to place in an interval of length w, the probability for t to fit is $\frac{w-t}{w} = (1-t)/w$. Thus, the probability of a collision must be the probability of getting a line segment of length t times the probability that it fits, thus

$$p(u) = Pr\left[h(v_1) = h(v_2)\right] = \int_0^{w/u} f_s(t)(1 - t/w)\, dt \tag{6.9}$$

Which can be written as

$$p(u) = \int_0^{w} f_s(t/u)\frac{(1 - t/w)}{u}\, dt = \int_0^{w} \frac{1}{u} f_s\left(\frac{t}{u}\right)\left(1 - \frac{t}{w}\right) dt \tag{6.10}$$

As we now know $p(u)$, we can define $P_1 = p(R)$ and $P_2 = p(cR)$.
Appendix B describes how we can estimate $p(u)$ in practice.

3 Structure

The structure is concerned with two problems: How the hash functions themselves are stored and how the data is stored.

3.1 Hash functions

When we talk about hash functions in this section we refer to the compound hash functions, denoted g_i in previous sections. The compound hash functions consist of a vector of hash functions. These functions are defined by a d dimensional vector, a, with entries chosen independently from an s-stable distribution and a number b chosen uniformly in the range $[0 : w]$.

Because a compound hash function simply contains k hash functions from $\mathcal{H}$, we represent the compound hash function by a $k \times d$ matrix (the k a vectors) and a k dimensional vector (the k b's).

3.2 The data

For every compound hash function g_i we need a hash table to store the points hashed by g_i, we call this table t_i.

In t_i we need to be able to find a point by a k-dimensional vector. There is more than one solution to this. In [And05] an algorithm for transforming the k-dimensional index into a one dimensional index with a checksum is given. In modern programming languages like Java, however, arrays can be used as indexes in hash functions.

4 Near Neighbor Search

When the structure is set up, the search for near neighbors is simple. Given a query point q we search all t_i with the key $g_i(q)$. This returns a body of candidate near neighbors.

In practice some points will be duplicates, but these are easy to weed out. Now we sort the points according to their distance to the query point. If z near neighbors are desired, we return the z closest points. If we solve the nearest neighbor problem, a linear search in the candidate near neighbor set can be used.

5 Parameter Choice

If we want to be able to use LSH, we need to specify the parameters k and L.

We want to return the correct result with success probability $1 - \delta$, thus we need to ensure that

$$1 - \left(1 - P_1^k\right)^L \geq 1 - \delta \tag{6.11}$$

thus

$$L \geq \frac{\log \delta}{\log\left(1 - P_1^k\right)} \tag{6.12}$$

Because there are no other restrictions on L, we choose

$$L = \left\lceil \frac{\log \delta}{\log\left(1 - P_1^k\right)} \right\rceil \tag{6.13}$$

When we need to choose k, things become difficult. We want to choose k so that query time is minimized. When we search for near neighbors, we firstly calculate the k-dimensional hash in all L hash functions. We denote the time this takes T_g. Then we calculate distances from the query point to all the near neighbor candidates. We denote the time this takes T_c. So our task is to find a k which minimize $T_g + T_c$.

It is clear that $T_g \in O(dkL)$.

T_c only depends on how many candidates we find, so $T_c \in O(d \cdot \#candidates)$. The number of candidate near neighbors, is exactly the number of collisions a query point induce in the hash table.

Given a sample query point, q, and the data set, P, and the function p calculating the probability of a collision between two points, we can estimate the number of collisions by:

$$E[\#collisions] \in O\left(L \cdot \sum_{e \in P} p(\|e - q\|_2)^k\right) \tag{6.14}$$

thus

$$T_c \in O\left(dL \cdot \sum_{e \in P} p(\|e - q\|_2)^k\right) \tag{6.15}$$

There might be different best k's for different query points, so in practice a body of representative query points should be used.

Now we are only left with the choice of w. There is no conclusive method of choosing this value, but [And05] suggests $w = 4.0$ as a good choice where all data points have their l_2 norm normalized to one. This value seems to work well with the data set [SDI06] used for testing.

6 Implementation of the Basic Structure

The data set is allocated with the type `ArrayList<double[]>`, meaning we have a list object containing arrays of doubles.

No type casting is used. If an object is defined as type A, it will stay that way. Thus, type checking is done at compile time and induces no extra time penalty at runtime.

A compound hash function is represented by a `NearNeighbor` object. The object contains a $k \times d$ dimensional array and a k dimensional array representing a compound hash function. The object contains a hash table defined as `HashMap<Integer, Bucket>` as well. When points are hashed to the hash table the hash value is calculated as an array. The array is then hashed to an Integer with the built in `Arrays.hashCode()` method.

This would represent exactly one compound hash function. To complete the amplification process we simply instantiate L `NearNeighbor` objects.

When points are stored in the hash tables, they are wrapped in a `Bucket` object. The `Bucket` object contains the index of a point in the original array, as well as a pointer to the next `Bucket` containing a point with the same hash value. Because we want to be able to sort buckets according to how close the points they represent are to the query point, the `Bucket` objects contain a distance attribute as well.

When we search the structure, we need to deal with duplicate points. We do this by using a `BitSet` to remember which points we have seen so far. In this way, we ensure that we only calculate the distance to a point in the body of candidate near neighbors once.

When all distances are calculated and duplicate points are weeded out, we use the `Collections.sort` to sort the points according to their distance to the query point. If we only need the nearest neighbor candidate, a linear search is employed.

7 Collision Probabilities in Hash Function

From Section 2.2 we have a function, p, which calculates the probability of a collision in a hash function, h, between two points v_1, v_2, with $\|v_1 - v_2\|_2 = u$. In this section we want to test if it holds in practice.

Also, we know that we should choose w "large enough"[SDI06], but how large is that for our test case?

7.1 Generating points

The first thing we need to address is how to generate points with distance u. In addition to the distance constraint, we want all points to have coordinates in $[0:1]$, thus residing in the unit box. Algorithm 6.1 generates a point pair with these properties.

Algorithm 6.1: Generate a point pair, with distance u, residing in the unit box.

$a \leftarrow$ unit length vector;
Multiply every coordinate in a with u;
foreach $d \leftarrow 1$ **to** $\#dimensions$ **do**
  $l \leftarrow \max\{0, -a_d\}$;
  $r \leftarrow \min\{1, 1 - a_d\}$;
  $b_d \leftarrow$ uniform random number in $[l:r]$;
  $a_d \leftarrow a_d + b_d$;
end
return (a, b);

When generating the unit length vector, simply generate a vector with all coordinates chosen at uniform random in $[-1:1]$ and divide every coordinate in the vector with the length of the vector.

The algorithm works because

- If a is a unit length vector, multiplying every coordinate in a with u will ensure $\|a\|_2 = u$.
- If a is a vector of length u and b is a random vector, setting $a = a + b$ will ensure $\|a - b\|_2 = u$.
- Because every coordinate in b is chosen in the interval $[\max\{0, -a_d\} : \min\{1, 1 - a_d\}]$, both a and b have coordinates between zero and one.

7.2 The test

The test is best explained by pseudo code, consider Algorithm 6.2.

We iterate over dimensions and in every dimension a new point-pair set is generated with a fixed seed.

For every seed, we initialize a hash function and count the number of collisions the point-pair set induces.

For the test we tried 100 random seeds, dimensions $\{5, 10, 15, \ldots, 500\}$ and a point-pair set of size 100000 (thus having 200000 points in total).

7.3 Result

Figures 6.1 and 6.2 show the results of our test runs. The results will be discussed below.

Algorithm 6.2: Count collisions in hash functions

```
seeds ← seeds we want to test;
foreach dimension do
    points ← point-pair set generated by fixed seed;
    c ← 0;
    foreach s ∈ seeds do
        h ← hash function generated by seed s;
        c = c+ #collisions in h induced by points;
    end
    print c/#points/#seeds;
end
```

7.4 Conclusion

The locality sensitive hashing scheme is not affected by dimensionality. This is a property which shows great promise of providing a method to handle similarity searches in high dimensional spaces.

It is also clear from the figures that if w is not chosen properly, we will not get the expected collision probabilities in practice. It seems, however, it is simply a matter of choosing w large enough.

8 Uniform data set

In most articles, LSH is only tested on natural data sets. This test will use LSH for finding nearest neighbors on a uniform random data set, with uniform random query points.

We will throughout the tests choose k and L so that we get an expected success probability of $(1 - \delta) = 0.9$ within a given radius, R.

8.1 The data set

The data set is 100000 points in 500 dimensions, with every coordinate in every point chosen at uniform random from $[0 : 1]$. We use 10 query points generated in exactly the same manner.

In [SDI06], points are normalized so that each point has its l_2 norm equal to one. For consistency, we will do the same.

8.2 Parameter setup

First, we need to establish the parameters for LSH.

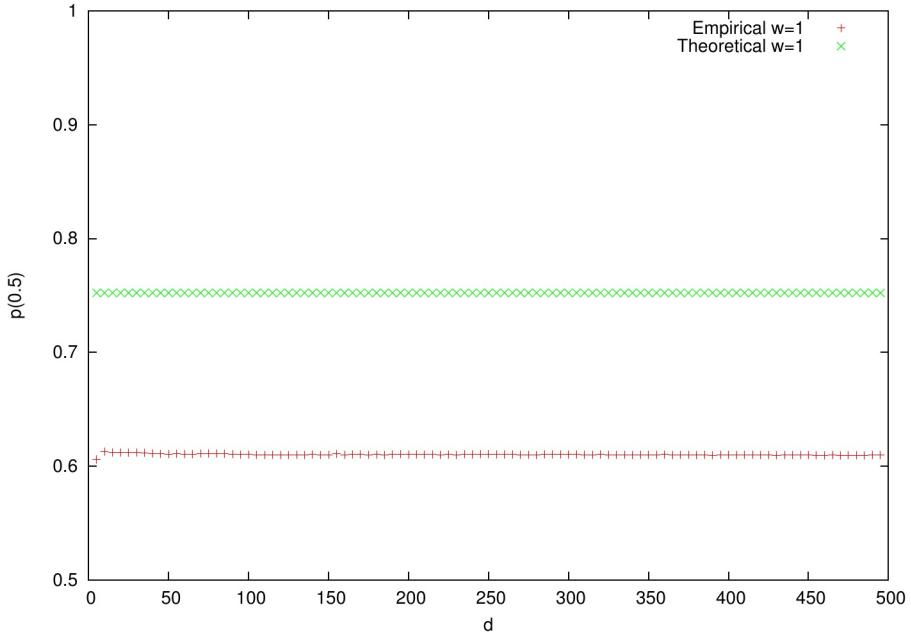


Figure 6.1: The theoretical probability of $p(0.5)$ versus the empirical, for $w = 1.0$

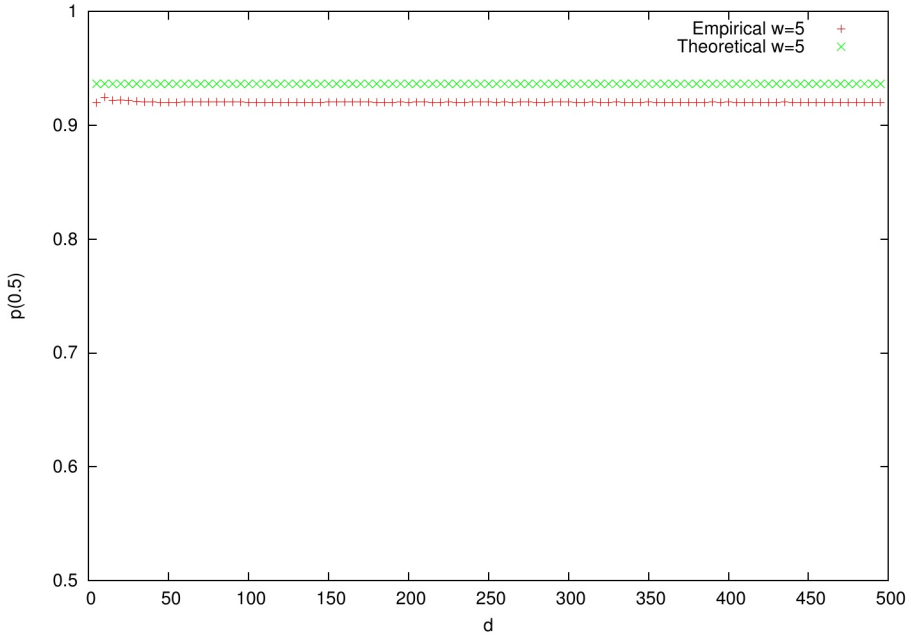


Figure 6.2: The theoretical probability of $p(0.5)$ versus the empirical, for $w = 5.0$

w and R

Because our search space is exactly the same as in [SDI06] we use $w = 4$ and $R = 1$ as well.

k and L

Now we want to choose k so that we get the smallest number of candidate near neighbors, as well as minimize the time it takes to compute the compound hash functions, in order to minimize query time. Section 5 discusses this.

Figure 6.3 shows a plot of $T_g + T_c$ where $T_g = dkL$ and $T_c = dL \cdot E\,[\#collisions]$. From the plot it seems that a smaller k is better. According to [And05], we would expect to find an equilibrium for some k.

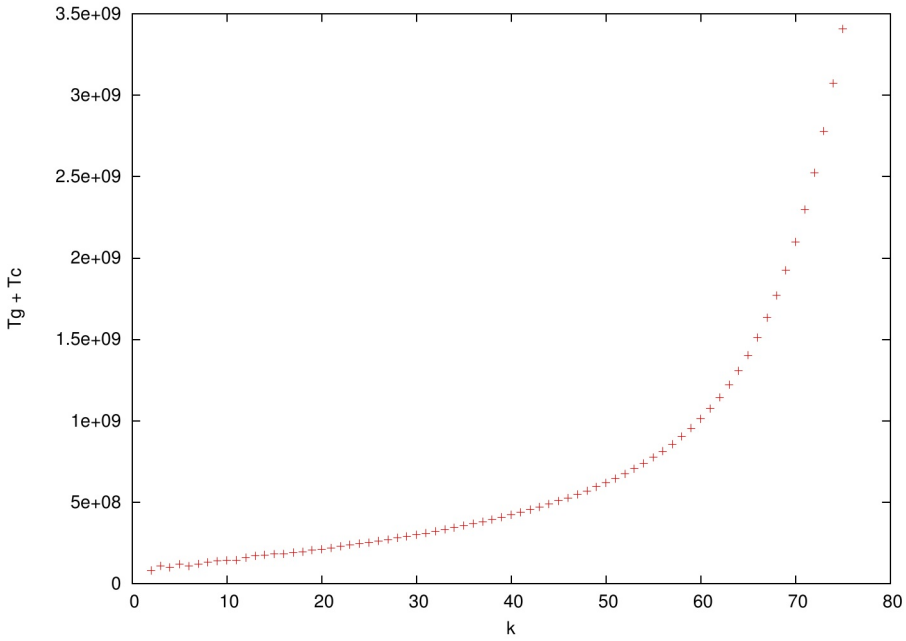


Figure 6.3: The relationship between $T_g + T_c$ and the parameter k, with $R = 1$.

The figure leaves out the constants involved, so the data may be skewed. However, the plot gives a clear indication that the value of k should be chosen very small.

8.3 The test

When considering the result tables, the following notation has been used: "candidates" is the number of unique candidate near neighbors found, "percent" is the percentage of the data set examined, "recall" is defined as in the preliminaries.

We tried different values for k in order to see if it would improve results.

When considering Table 6.1 it is clear that the number of near neighbor candidates is not consistent with what would be expected, considering Figure 6.3. One could suspect that T_g distorts the data, but even when considering T_c alone the result is the same.

The method for estimating T_c counts the total number of collisions. In Table 6.1, we only count unique candidate near neighbors as duplicate points are easily weeded out on the fly. So the model we use for estimating collisions is not totally practical. We will,

k	L	Candidates	Percent	Recall
2	2	77668	77%	100%
4	3	89734	89%	100%
5	7	94825	94%	100%
16	19	87381	87%	100%
24	58	79204	79%	100%
32	172	74402	74%	100%
36	295	73832	73%	100%

Table 6.1: LSH performance on uniform random data set, with $w = 4.0$ and $R = 1$.

however, try to see if the method can give us some insight as to how we should choose k.

Because L grows exponentially with k when we have a large R and success probability, we did not have enough memory to perform the test with larger k.

8.4 Readjusting parameters

When we get too many candidate near neighbors from the structure, we want to decrease the search space. We do this by reducing R. So we set $R = 0.5$ and redid the test, to see which k we should choose now. Figure 6.4 shows what we would have expected the

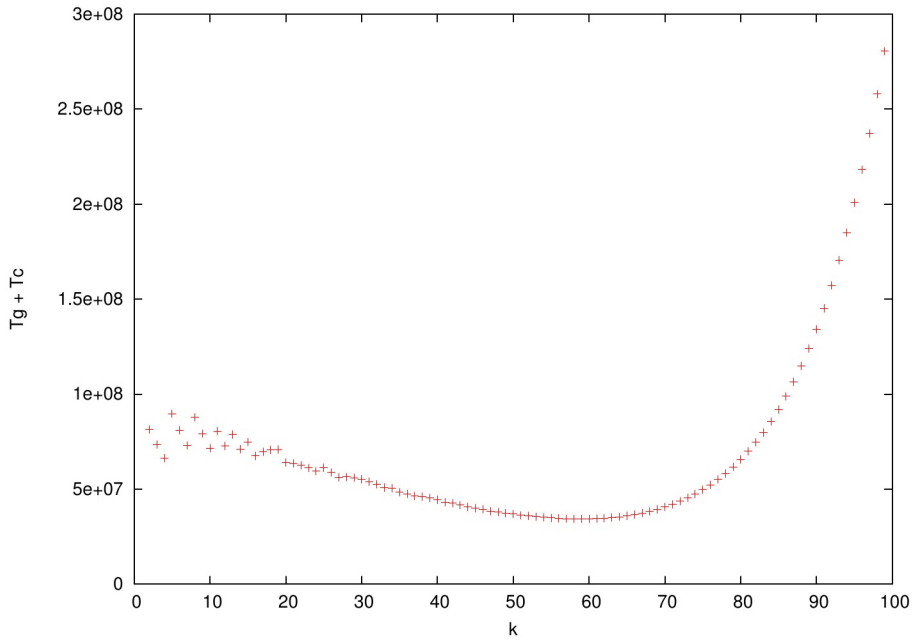


Figure 6.4: The relationship between $T_g + T_c$ and the parameter k, with $R = 0.5$.

first time. We try to run the test again, with readjusted parameters, the results can be seen in Table 6.2.

k	L	Candidates	Percent	Recall
40	56	12422	12%	90%
45	78	5548	5%	60%
50	116	6867	6%	80%
55	172	5259	5%	50%

Table 6.2: LSH performance on uniform random data set, with $w = 4.0$ and $R = 0.5$.

The number of candidates examined is much more acceptable now. The recall, however is much lower than we would prefer.

8.5 Last iteration

Because of the low recall, we try with $R = 0.75$. This middle ground will hopefully give us the best of both tests: a good recall and an acceptable number of collisions.

We plot $T_c + T_g$ again, see Figure 6.5.

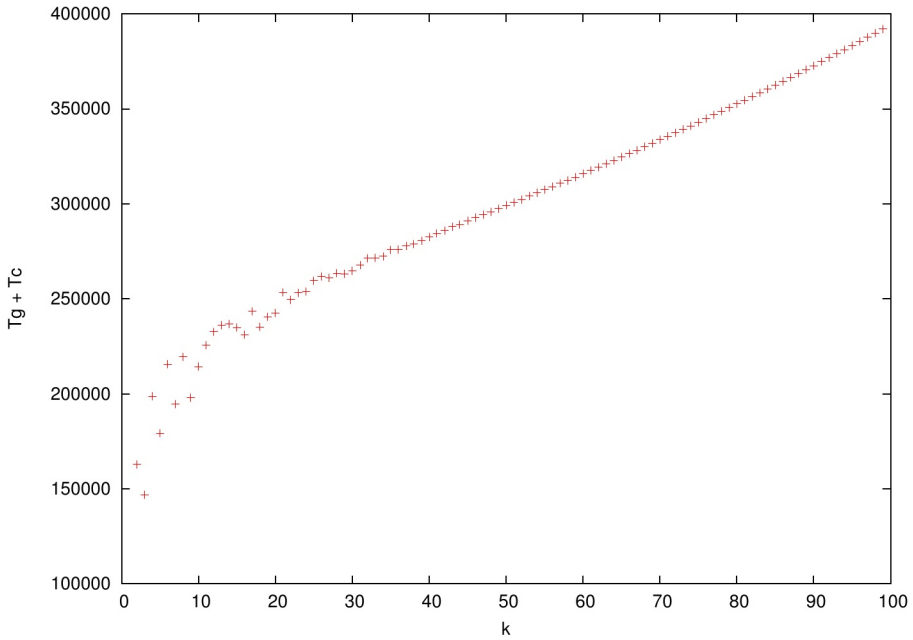


Figure 6.5: The relationship between $T_c + T_g$ and the parameter k, with $R = 0.75$.

Like the first time around it seems we should just pick a small k. We are going to try some larger values of k, even though Figure 6.5 suggests that it may be futile.

The test results are in Table 6.3.

Again, we see that it pays off to choose a larger k and thus spend more memory. The recall is acceptable and the number of candidates examined is much smaller than in the first test.

k	L	Candidates	Percent	Recall
3	2	58922	58%	100%
5	3	94421	94%	100%
10	6	82948	82%	100%
20	19	60090	60%	100%
30	58	55902	55%	100%
35	101	32490	32%	100%
40	173	32867	32%	100%

Table 6.3: LSH performance on uniform random data set, with $w = 4.0$ and $R = 0.75$.

8.6 Conclusion

It seems that the best way to tune k, is to actually build the data structure and test how well the chosen value works. The estimation paradigm suggested in Section 5 does not seem to work well in practice. A way of estimating the number of duplicate points returned is needed for this approach to be practical. If this could be taken into account, the estimated query times would be much more sound.

The tests also show that it does pay off to use more memory, until a certain point. However, the number of candidate near neighbors examined is still large if the goal is to find the exact nearest neighbor.

9 Uniform In More Dimensions

We want to test if LSH is resistant to dimensionality in practice. Thus, we choose two parameter configurations from the previous section and test them on a set of dimensions.

We use the exact same generator for the data sets here as we used in the previous section.

The dimensionality will start at 100 and jump with steps of 23 up to 690.

9.1 Low R

First we try with the configuration $w = 4.0$, $k = 35$, $L = 101$, which gives us a success probability of 0.9 within $R = 0.75$. This configuration provided good recall and low feedback on the data set used in Section 8.

Figures 6.6 and 6.7 show the results.

On the positive side, it seems that recall and feedback are somewhat stable as dimensionality increases. On the negative side, it seems that the configuration chosen, lead to poor recall.

9.2 Higher R

Because the recall was a bit lower than expected, a new configuration was tried: $w = 4.0$, $k = 24$, $L = 58$, which gives a success probability of 0.9 within $R = 1$. This should insure

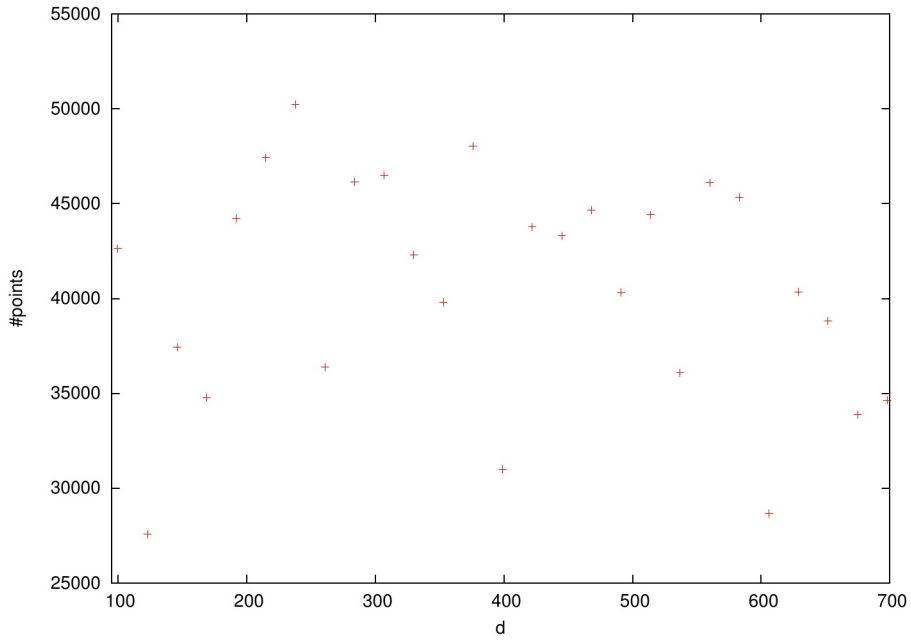


Figure 6.6: Performance in terms of feedback as dimensionality increase. Configuration: $k = 35$, $L = 101$.

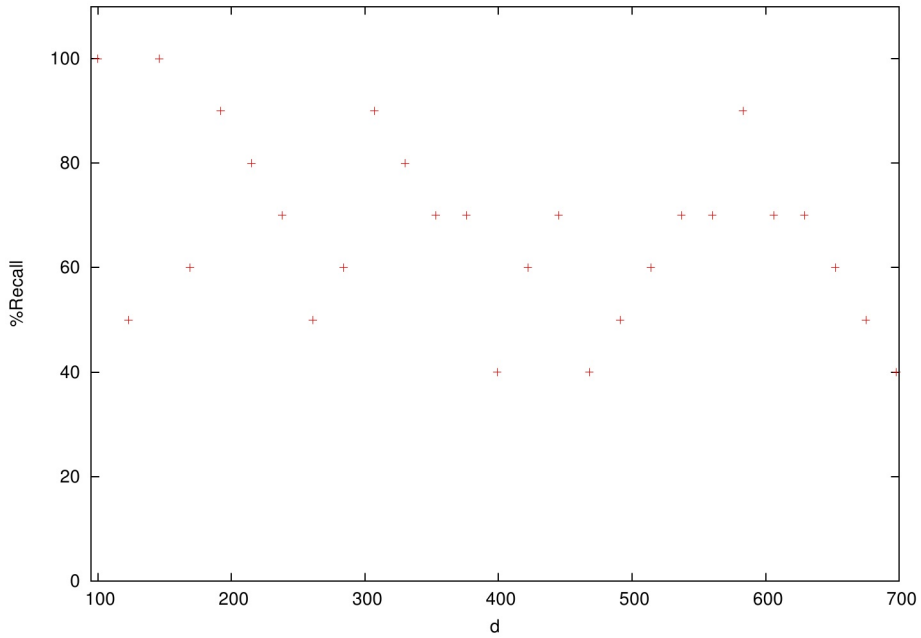


Figure 6.7: Performance in terms of recall as dimensionality increase. Configuration: $k = 35$, $L = 101$.

higher recall, but the feedback might get high as well.

Figures 6.8 and 6.9 show the results.

As expected we now get a much better recall. However, notice how we must examine more than 75% of the data set to attain this recall. Even for very large data sets, this is not much of an improvement.

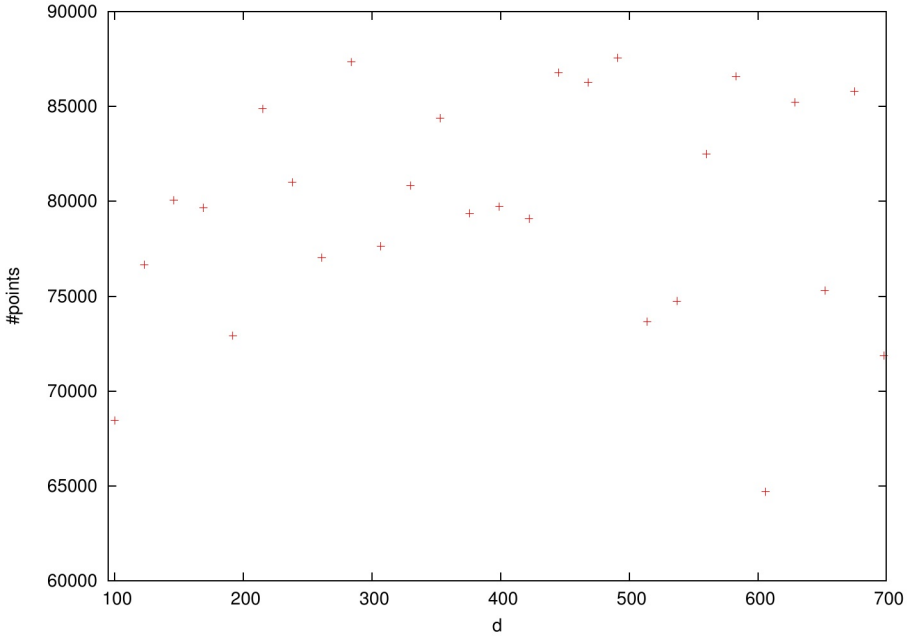


Figure 6.8: Performance in terms of feedback, as dimensionality increase. Configuration: $k = 24$, $L = 58$.

9.3 Conclusion

From both tests we can conclude that LSH does not seem to degrade as dimensionality grows. There is no evidence that performance will suffer from higher dimensionality, on uniform data.

However, the method struggles to perform well on uniform data. It seems that when query points are placed far from the data points, with respect to the distribution of the data points, the method needs a very high feedback to attain good results.

10 Clustered Data

When considering the MNIST[LC] data set used to test LSH in other articles, it is clear that it is a clustered data set. The data set is generated by vectorizing handwritten digits, so you would expect ~ 10 clusters of data each representing a digit. Thus, when searching into a cluster, everything is close and a low value of R can be used.

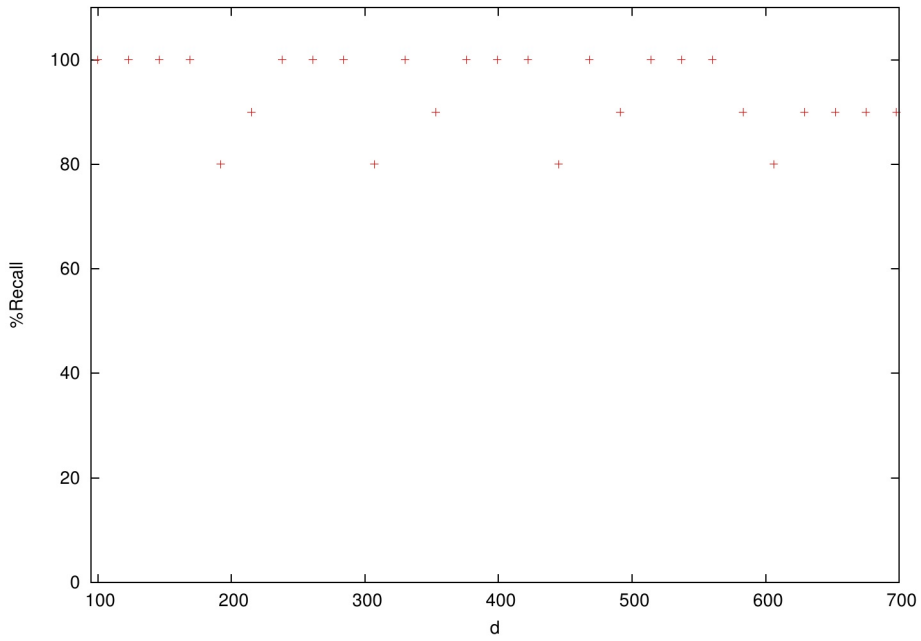


Figure 6.9: Performance in terms of recall, as dimensionality increases. Configuration: $k = 24$, $L = 58$.

10.1 The data set

We want to generate a synthetic data set with some of the same properties as MNIST. To achieve this, we generate n_c points with coordinates in $[0 : s]$ and around each point we choose n_p points from a normal distribution, $\mathcal{N}(m, 1)$, where m is one of the n_c points. In effect the n_c points act as medians. Algorithm 6.3 reveals the details.

Algorithm 6.3: Generate a data set with n_c clusters with n_p points in each cluster

```
medians ← n_c points, having coordinates in [0 : s] ;
foreach m ∈ medians do
    for 1 → n_p do
        for i ← 1 to d do
            p_i ← random number from N(m_i, 1);
        end
        Add p to data set.
    end
end
```

The data set we will be using for the tests are generated with the parameters: $d = 500$, $s = 100$, $n_p = 1000$, and $n_c = 100$. Query points are simply 10 points chosen at uniform random from the data set. Thus we have 99990 data points and 10 query points.

Points are normalized so that each point has its l_2 norm equal to one.

10.2 Parameter setup

A lesson learned from the previous section is that k and L are best found simply by sampling values. Thus, this technique will be used here. We will choose k and L, so that the success probability is 0.9 for a given value of R.

Because we do not have a good way of estimating w we will simply keep using $w = 4.0$, unless this leads to problems. We will try different values for R, slowly decreasing it, to see how low we can go.

To get an idea of how close data are packed, we use the linear search to find the nearest neighbors and they all reside within ~ 0.03 of the query points.

10.3 R = 0.5

Considering how close the query points are to their nearest neighbor, $R = 0.5$ seems like a conservative guess.

k	L	Candidates	Percent	Recall
2	2	70285	70%	100%
10	4	52281	52%	100%
20	10	37271	37%	100%
30	24	27212	27%	100%
40	53	11838	11%	100%
45	78	5271	5%	100%
50	116	5027	5%	100%

Table 6.4: LSH performance on a clustered data set, with $w = 4.0$ and $R = 0.5$.

From Table 6.4 we see that this configuration gives a good recall and for large k a much smaller set of candidate near neighbors than we have seen so far has to be examined.

10.4 R = 0.1

Inspired by the success, we tried a smaller value of R.

Table 6.5 show that we can keep the good recall with a much smaller R. For high k we get the smallest candidate near neighbor sets back, so far.

10.5 Conclusion

From the results, it seems like k simply should be chosen "large enough". Handling the extra duplicates, from the larger value of L, is not a problem. So, in a practical scenario, one should simply choose the largest k, while keeping everything in the memory.

k	L	Candidates	Percent	Recall
2	1	50861	50%	100%
10	2	32025	32%	100%
20	3	8382	8%	100%
40	4	1096	1%	100%
50	5	1338	1%	100%
60	7	1081	1%	100%
70	8	999	1%	100%
80	10	999	1%	100%

Table 6.5: LSH performance on a clustered data set, with $w = 4.0$ and $R = 0.1$.

For this data set, we had to examine at least 1% of the points to find the nearest neighbor. This could indicate that we are searching through exactly one cluster. For comparison, we ran the KdTree on the exact same data set. $\sim 5\%$ of the points were examined in the nearest neighbor search.

Remember that all query points where placed in a cluster, so the nearest neighbor is never far away in these tests. When query points are placed in an empty space, LSH will return useless results and the KdTree will degrade to an inefficient linear search.

11 Close Clusters Data

In the test above, clusters were deliberately placed far from each other. This meant that the data set was easy to handle for both LSH and the KdTree. We now test how close we can place clusters before we run into problems.

When LSH returns a body of candidate near neighbors, every candidate will belong to a cluster. We count how many candidates we get from each cluster and note the size of the largest represented cluster. This test will tell us if LSH actually returns a cluster or if it was a coincident that we got cluster-sized results before.

11.1 The data set

The data set is generated exactly as before, except that we are going to try all $s \in \{1, 4, 7, \ldots, 100\}$. As before, points are normalized so that each point has its l_2 norm equal to one.

11.2 Parameter setup

Because we got the best results with the configuration: $w = 4.0$, $k = 80$, $L = 10$ in the previous section, we are going to stick with this.

11.3 Results

Figures 6.10 and 6.11 show the results of our test runs. The results will be discussed below.

11.4 Conclusion

The first thing that calls for an explanation is why we get zero feedback for a very low value of s, as seen in Figure 6.10. We need to remember that all points are normalized to have their l_2 norm equal to one, so every point resides on the unit hyper ball. When distances between clusters decrease, the distribution looks more and more like uniform distribution. When normalized, distances between points on the unit hyper ball become large, compared to when distances between clusters were large.

As expected, the largest cluster represented in the feedback is almost as big as the total feedback returned. This is not unexpected. When LSH hashes points, close points are hashed to the same value and every point in a cluster is close. Thus, almost every point in the cluster will be returned when a query point is in a cluster.

Recall drops significantly with the feedback. Because distances increase on the unit hyper ball, even points in the same cluster are hashed to different buckets.

If we had refrained from normalizing the points and tuned the algorithm for this data set without normalization, results would definitely have been different. In this case we would expect an increase in the size of the feedback corpus as the value of s decreases. A drop in recall would be expected as well. Results like the ones seen in the uniform test would be expected for small values of s.

12 Another Implementation

To verify the results of our LSH implementation, we will now compare some of the results with the implementation E^{2}LSH [And] made by A. Andoni.

Note that we record how many points are examined, not how many points, p, having $\|p - q\|_" \leq R$, to a query point, q. With a standard setup, E^{2}LSH will count points having $\|p - q\|_2 \leq R$.

12.1 Uniform data set

In this section the E^{2}LSH implementation will be tested on the data set described in Section 8.

E^{2}LSH has a build-in auto tuning which uses the concepts described in Section 5. We will try to use the parameters chosen by the auto tune here. E^{2}LSH use an optimization for faster computation of the g_i functions described in [And05], thus having 3 parameters: k, m, L.

We use `lsh_compute_params` to suggest parameters, given a radius R and a success probability.

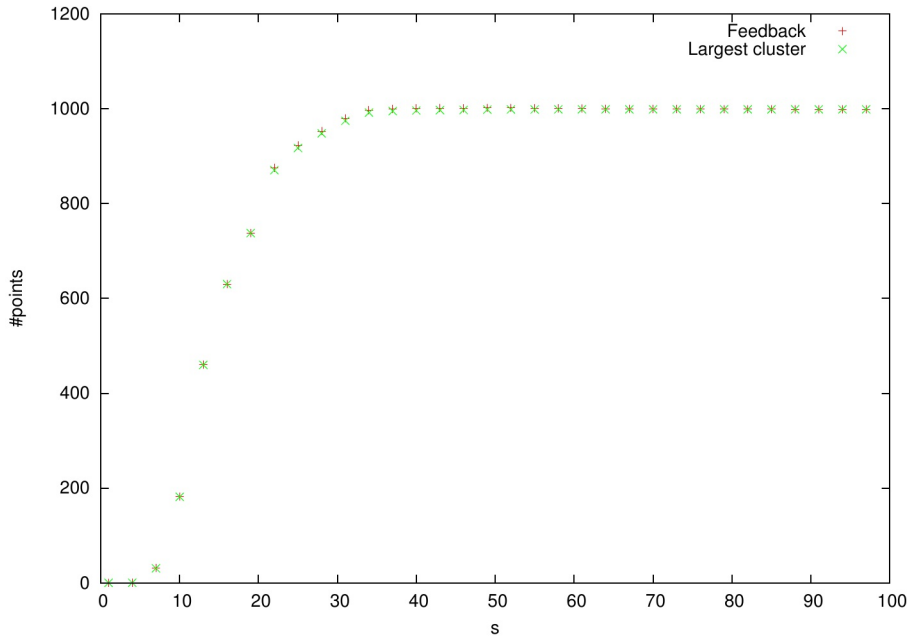


Figure 6.10: The total feedback versus the largest cluster represented in the feedback, for a given spacing between clusters, s.

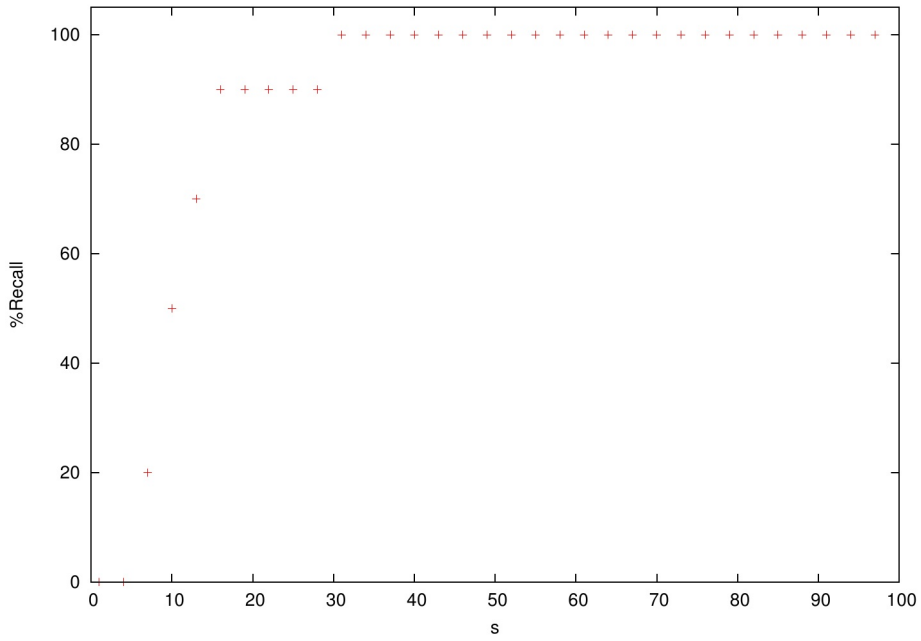


Figure 6.11: Recall for a given spacing between clusters, s.

From Table 6.6, it is clear that the auto tune does not work very well. We now turn off the optimization for faster computation of g_i functions, and try some of the

R	k	m	L	Candidates	Percent	Recall
1	4	5	10	100000	100%	100%
0.75	16	22	231	99892	99%	100%
0.5	26	69	2346	99977	99%	100%

Table 6.6: E^{2}LSH performance, with auto tune, on a uniform data set, with $w = 4.0$.

configurations with which we had success, in the previous chapter.

k	L	Candidates	Percent	Recall
36	295	70553	70%	90%
40	506	70498	70%	100%
45	994	66582	66%	90%

Table 6.7: E^{2}LSH performance on a uniform data set, with hand tuned parameters and $w = 4.0$ and $R = 1$.

Table 6.7 and 6.8 have the results. Because E^{2}LSH is written in C, it is more space efficient than our own Java version, thus we could try out configurations with larger k.

k	L	Candidates	Percent	Recall
3	2	99996	99%	100%
40	173	33235	33%	40%
50	512	26637	26%	80%

Table 6.8: E^{2}LSH performance on a uniform data set, with hand tuned parameters and $w = 4.0$ and $R = 0.75$.

Even with the setup for $R = 1$, E^{2}LSH does not get a 100% recall. This result is very surprising. It can, however, be attributed to an unlucky choice of seed for the E^{2}LSH random number generator.

Conclusion

With the uniform data set it seems that the more advanced E^{2}LSH-implementation performs worse than what we have seen previously. This is surprising as you would expect the two LSH-implementations to perform equally well on two similar configurations. E^{2}LSH uses a lot of optimizations so even though the configuration is the same, we cannot expect the two implementations to yield the same results.

We can confirm that, on uniform data, the auto tuning method presented in Section 5 is impractical for this implementation as well.

12.2 Clustered data set

In this section the E^{2}LSH-implementation will be tested on the data set described in Section 10.

Again, use the `lsh_compute_params` to suggest parameters, given a radius R and a success probability.

R	k	m	L	Candidates	Percent	Recall
1	4	5	10	99990	100%	100%
0.75	16	22	231	99902	99%	100%
0.5	26	69	2346	99990	100%	100%
0.1	8	9	36	99891	99%	100%

Table 6.9: E^{2}LSH performance on a clustered data set, with auto tuned parameters and $w = 4.0$.

Considering Figure 6.9, we see that the auto tune is unable to find a good configuration for the data set. Moreover, the auto tune suggests the exact same configurations, as with the last data set. We expected that the clustered data set would induce a new configuration.

We now turn off the optimization for faster computation of g_i functions and try some of the configurations with which we had success in the previous chapter.

k	L	Candidates	Percent	Recall
2	1	98990	98%	100%
10	2	92165	92%	100%
20	3	17580	17%	100%
40	4	1808	1%	100%
50	5	1174	1%	100%
60	7	1302	1%	100%
70	8	1206	1%	100%
80	10	999	1%	100%

Table 6.10: E^{2}LSH-performance on a clustered data set with hand tuned parameters and $w = 4.0$.

From Figure 6.10, we see that given hand tuned configurations the E^{2}LSH-implementation performs much better.

12.3 Conclusion

From the results above it is safe to say that the auto tuning in the E^{2}LSH implementation does a very poor job of finding good configurations. On the uniform data set, it is not unexpected that the auto tune does a poor job. However, on the clustered data set one would expect that it is possible to auto tune parameters. A naive strategy of simply choosing the largest possible k would have done better than the auto tune.

When given hand tuned configurations, the E^{2}LSH performs slightly worse on the uniform data set. This behavior can be attributed to the fact that not all seeds work well on the uniform data set. There seems to be a huge difference in performance, simply by choosing a lucky seed to the random number generator.

On the clustered data set, performance is much more comparable. Here near neighbors are close and choosing a bad seed will not have as much impact. With high k, there is no difference in performance between the two implementations.

13 Multi-Probe LSH

In practice, the basic LSH scheme needs a large number of hash tables to provide high accuracy. To overcome this drawback, strategies for searching multiple times in the same hash table are proposed in [Pan06] and [LJW+07].

In [Pan06], the strategy is entropy based: Given a query point, q, a set of new query points is generated around q. The new query points are then used to probe the hash tables for more possible near neighbors to q.

In [LJW+07], the strategy is based on perturbing the k-dimensional hashes. From [LJW+07], this seems to be the more promising approach.

13.1 The Strategy

The strategy described in [LJW+07] takes a k-dimensional compound hash and perturbs z values in the hash so that the new hash has a high probability of leading to a better solution. This section describes the algorithm for generating the m most promising hashes.

First, we formalize how to perturb a vector. Recall how $g = \langle h_1, \cdots, h_k \rangle$. We define a perturbation on g as

$$\pi = (i, \delta) \text{ where } i \in \{1 \cdots k\},\ \delta \in \{-1, 1\} \tag{6.16}$$

which represents adding δ to the i'th hash value in g. So for $g(p) = \langle h_1(p), h_2(p), h_3(p) \rangle$, $\pi = (2, 1)$ would induce $g(p) = \langle h_1(p), h_2(p) + 1, h_3(p) \rangle$. For a hash function g, $2k$ distinct perturbations exist.

The family of hash functions described in Section 2, projects a point onto the real line and then chops the line into buckets. Let $f_i(q)$ be q's projection onto the real line with respect to a hash function, h_i.

From this logic we set a score for a perturbation: $score(\pi) = x_i(\delta_i)$, where $x_i(-1)$ is the distance from $f_i(q)$ to the left side of the bucket and $x_i(1)$ is the distance to the right. A lower score suggests a more promising perturbation. See Figure 6.12 for a visual description.

Let Δ be the sorted set of all π so that it holds $score(\Delta_i) \leq score(\Delta_{i+1})$.

We now define a perturbation set as a set of indexes from Δ. This represents perturbing a set of hashes in g.

We want to be able to generate perturbation sets which will lead us to a better solution with high probability. A perturbation set, A, can be scored in the same manner as a perturbation: $score(A) = \sum_{j \in A} score(\Delta_j)$. Recall that $\Delta_j = \pi_i$ for some j, i. Algorithm 6.4 generates perturbation sets in decreasing order according to score.

We define 2 operations on the perturbation sets

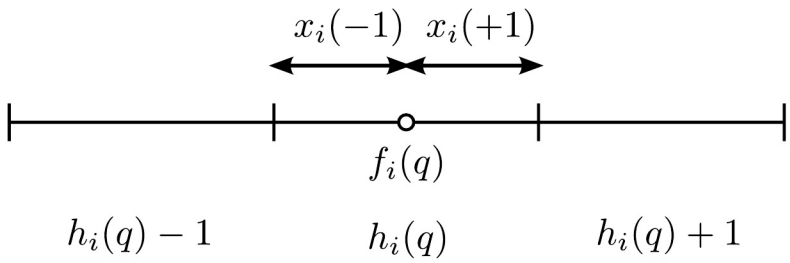


Figure 6.12: Notation for the perturbation strategy.

Algorithm 6.4: Generate m perturbation sets

```
A_0 ← {0};
insert ⟨A_0, score(A_0)⟩ into minHeap;
for i ← 1 to m do
    repeat
        A_i ← extract min from minHeap;
        A_s ← shift(A_i);
        insert ⟨A_s, score(A_s)⟩ into minHeap;
        A_e ← expand(A_i);
        insert ⟨A_e, score(A_e)⟩ into minHeap;
    until valid(A_i);
end
return A;
```

- $shift(A)$: Replace the largest value in A, l, with $l + 1$. E.g. $shift(\{1, 2, 3\}) = \{1, 2, 4\}$.
- $expand(A)$: Let l be the largest value in A. Insert l+1 into A. E.g. $expand(\{1, 2, 4\}) = \{1, 2, 4, 5\}$

In this manner, we generate the perturbation sets used to probe the hash tables t_i. On query time, it is simply a matter of generating the m perturbation sets and use them on the k-dimensional hashes, generated by the g_i hash functions. In practice, the cost of validating the perturbation set is exactly the same as simply probing the hash table with the perturbed hash. So in order to avoid the substantial overhead of validating the perturbation sets, we use perturbed hashes without validation. Thus, some of the m perturbation sets generated may be invalid.

14 Uniform Multi-Probe

The data set having caused the most problems so far is the uniform data set. We need large amounts of memory just to get mediocre results. In this section, we test if the multi-probe technique will work well for the uniform data set.

14.1 Methodology

We keep some of the configuration from Section 8: $w = 4.0$, $k = 35$, and $L = 101$. Now we will try different configurations: $L \in \{25, 50, 75\}$ and $m \in \{0, 10, 50, 100, 150, \ldots, 500\}$. We try all combinations of L and m. The exact same data set and seed as in Section 8 have been used. Therefore, results are comparable.

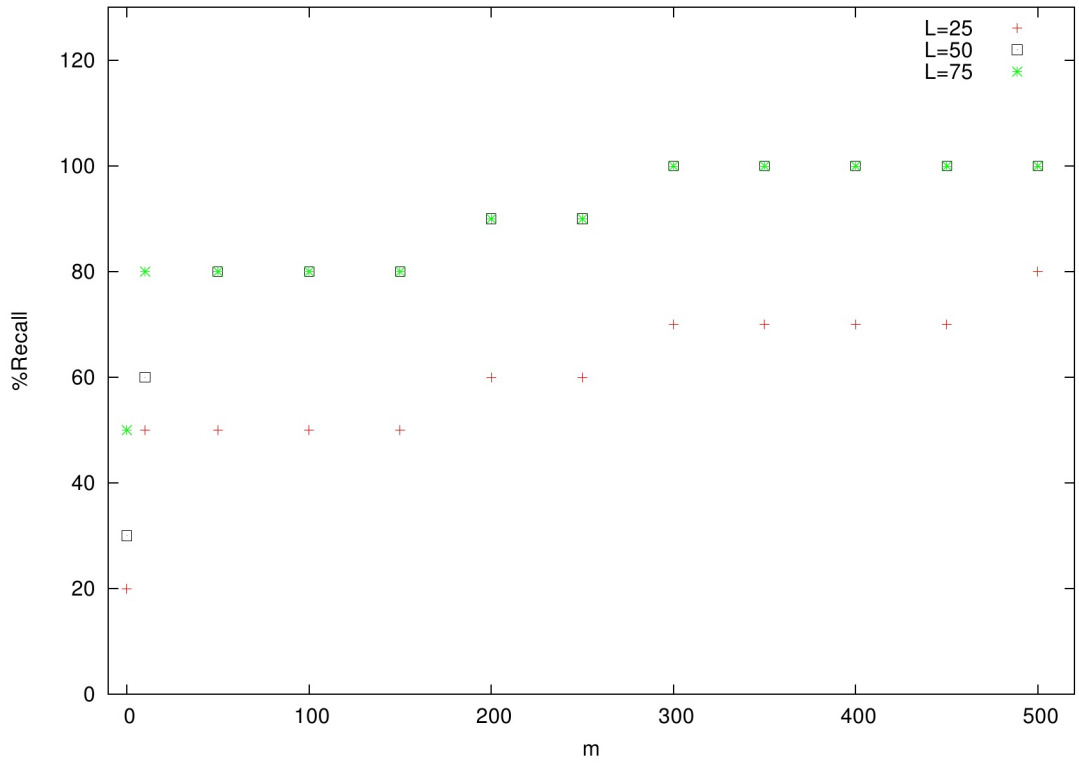


Figure 6.13: The recall with different settings of m and L.

14.2 Conclusion

Figure 6.13 and 6.14 show that multi-probe can be used to amplify how much accuracy we get from a fixed L. However, what we gain in accuracy we pay for in the amount of feedback we get.

In the first test with uniform data, the best result was 100% recall and 32% feedback. This result could not be sustained when we tested in more dimensions where more than 75% feedback for 100% recall was needed. So depending on context, the results gained with MultiProbe-LSH are neither weaker nor stronger than what we have seen before.

The multi-probe scheme definitively helps bring down the memory usage while increasing accuracy. We have yet to measure the extra time penalty for generating the probing sequence. If the cost is high, the optimization may not be desirable. Because the probing sequence depends on information from both the hash functions and other

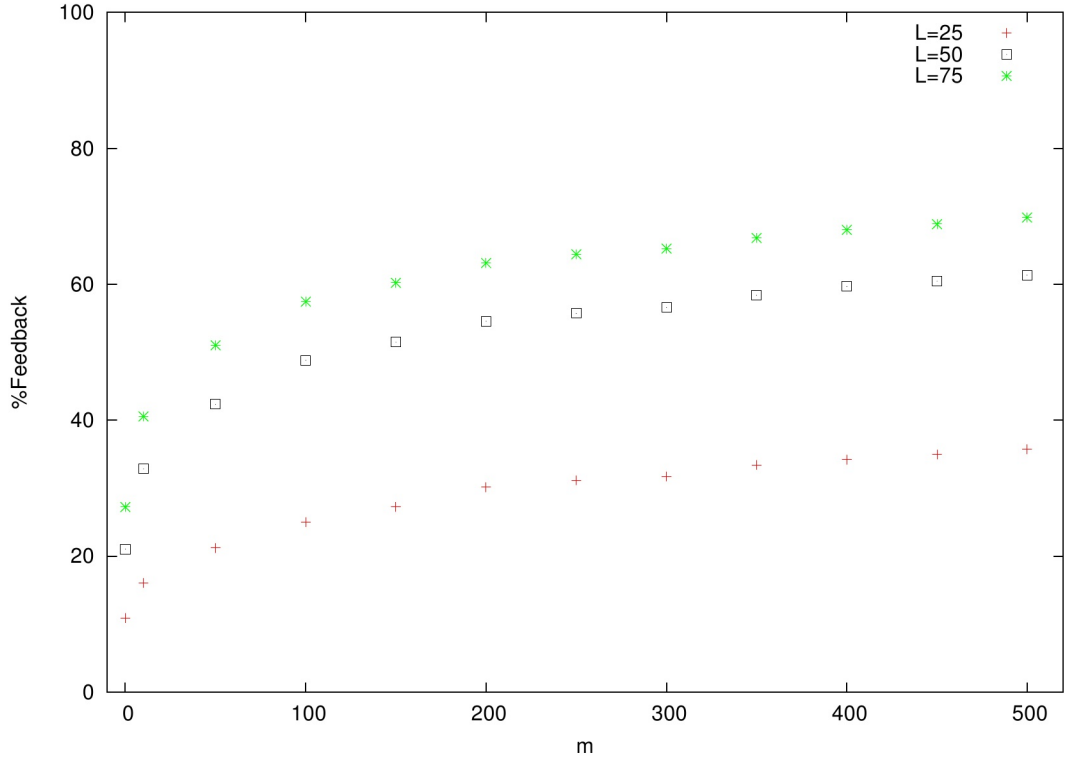


Figure 6.14: The feedback with different settings of m and L.

parts of the LSH scheme, the probing sequence will add a small cost in many places. Thus, it is difficult to measure what the real cost of the probing sequence is. We will test the MultiProbe-LSH against the basic LSH in a later chapter. This test will depict more precisely what the cost of multi-probe is.

7

GPU Based search

In recent years, the improvement in graphics processing units (GPU) has provided the computer vision community with powerful processing units. With the addition of GPGPU (General-Purpose computation on GPU) via e.g. the CUDA interface, highly parallel problems can now be solved on the GPU.

The nearest neighbor problem is very paralizable when using a simple linear search, also known as brute force search. Thus, it should be able to utilize the power of the GPU. In recent papers, this approach shows great promise for the all pairs nearest neighbor problem [GDB08], [QMN09].

1 The Architecture

Before considering the algorithm, the architecture on which the algorithm will be running is presented.

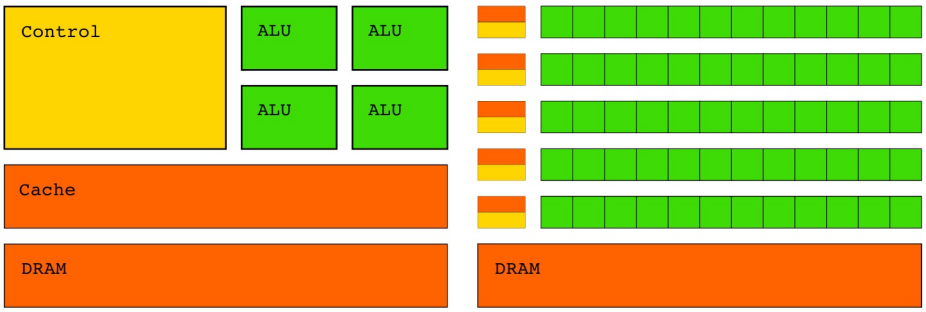
1.1 Hardware

The basic idea is to have a highly parallel, multi-threaded, many-core architecture with high processing power and high bandwidth. Consider Figure 7.1 for an illustrative comparison of the GPU with a regular CPU.

On the GPU, the ALUs are split into blocks which share the same fast memory (not to be confused with cache). The GPU operates with 3 levels of memory:

Global: Visible to all cores in all blocks. Can be used for inter-block communication. The CPU will upload data for processing here. Global memory can be mapped into textures. The texture mapped memory has some caching, thus it makes sense to map global memory into textures, if it is necessary to access it multiple times.

Shared: Visible to all processes in the appurtenant block. Used for communication between threads in the same block. Faster than global memory. Shared memory



(a) CPU (b) GPU

Figure 7.1: A illustrative comparison of the CPU versus the GPU

is expected to be about as fast as L1 cache.

Local: Only visible to one thread. Used for private data and sub computations.

The GPU devotes many more transistors to arithmetic operations compared to the CPU. The trade-off is that the GPU uses much fewer transistors on caching and flow control. This means that the GPU is much more efficient at computational intensive tasks. Especially tasks that can be expressed as data parallel tasks where the same code is executed on many data elements in parallel.

1.2 Programming model

When using the CUDA programming interface, the user is programming in a heterogeneous environment. The code contains a mix of host and device code where host code will be executed on the CPU and device code on the GPU. The host code will load in data, copy data to device memory and start the device code. The host code is normal C-code with some extended functions.

The device code looks a lot like C-code as well and is partitioned into functions called kernels. All kernels are defined by how many blocks and threads per block they are executed on, as well as a set of parameters. All kernels have void as return type. Kernels cannot be recursive and are always started from host code.

Threads are enumerated by a 1, 2 or 3-dimensional vector, forming a thread block. This can be seen as a grid of threads. Each thread block resides on the same physical core on the GPU, and thus have to share the same limited memory. This puts a limit on the number of threads per block, which is currently 1024.

Blocks are enumerated by a 1, 2 or 3-dimensional vector as well, forming a block grid. On execution, the blocks in the grid are mapped onto physical cores so threads can be executed. A core may serve more than one block or none at all. Thread blocks must run independently, thus it must be possible to execute them in any order, serial or parallel.

As mentioned before, threads can communicate within a block by using shared memory. They can also synchronize execution in a block by using a barrier, a function all threads in the block must reach before executing any further.

For more details, the NVIDIA CUDA programming guide[nVi] should be consulted.

2 Algorithm

In practice we need to define how threads and blocks are instantiated, and thus how threads are allocated to cores. The CUDA manual suggests to make a large amount of threads compared to the number of cores on the GPU. Thus, we went for 512 blocks each containing 512 threads, considering our test machine has 16 CUDA cores. See Appendix A.1 for details on the test machines.

We say that thread t belongs to block $1 \leq t_b \leq 512$, and within this block it has thread id $1 \leq t_i \leq 512$.

If $t_i < d$, t will copy coordinate t_i of the query point from global memory into shared memory in the block. This allows fast access to the query point at all times.

Thread t will now proceed to calculate the distance between the query point and point $t_b * 512 + t_i$ in the data set if such a point exists. The result is placed in shared memory.

In each block, thread t with $t_i = 1$ will find the best near neighbor and place the result in global memory.

The CPU will now fetch the sub results and calculate the nearest neighbor.

3 Implementation

Everything has been done in CUDA-C. This makes the results measured in seconds incomparable to other results in this thesis. In general, C has much faster memory access than Java.

Test data is generated by the host code.

Different memory models have been implemented:

- The naive approach, where points are placed in a one dimensional array with the first d entries representing the first point and so on. This can easily be transfered to global memory on the GPU.
- Texture mapped global memory, with data allocated as before.
- And lastly, CUDA arrays have been tested. These are arrays as known in C except that they are aligned in memory to be favorable to the GPU memory.

4 Test

In the test, we measure the speed of the CUDA implementation to a linear search implemented in C, to give a realistic picture of the speedup one might attain by utilizing

the GPU.

4.1 data set

The data set consists of uniform random data and query points. We use 32000 data points in 500 dimensions and 1 query point. We repeat the test 10 times and note the average.

We had to use very small instances due to the limited global memory on the graphics card in the test machine. The test machine is a Mac Mini, described in Appendix A.1.

4.2 Results

On average, it takes the CUDA implementation 0.109823 seconds to find the nearest neighbor when using texture mapped memory. The linear search implemented in C used 0.000446 seconds.

Task	**time**
GPU Memory allocation	0.000290 sec
Copy data to GPU memory	0.054459 sec
Texture map memory	0.000027 sec
Calculations on GPU	0.054909 sec
Fetch final result	0.000084 sec

Table 7.1: The time used for each task in the CUDA implementation. The test used 32000 data points in 500 dimensions. The time noted is the average of 10 runs. Global memory was texture mapped.

From Table 7.1, it is easy to see that it is the memory intensive operations which take time. The test in Table 7.1 was done with texture mapped memory. We also tried using global memory without texture mapping and cuda arrays as memory model, but with similar results.

4.3 Conclusion

Because global memory on the GPU is slow, the task performed on the GPU needs to be very computationally intensive. However, it seems that the task at hand is very memory intensive. This is not to say that the GPU cannot be used for special cases of the nearest neighbor problem. The all pairs nearest neighbor problem seems to be very well suited for the GPU [GDB08].

8

Method Comparison

This chapter is devoted to comparing a subset of the algorithms presented. We leave out GPU based linear search because it is not practical. The R-Tree has been omitted as well, as build times makes extensive testing very time consuming. Furthermore, our tests of R-Trees show that they do not provide superior performance compared to KdTrees.

We want to compare the algorithms on two parameters: speed and quality. Speed will be measured in seconds per query. Quality will be measured in recall of nearest neighbors.

All data sets will consist of 100 query points and a total of 100000 data points. All data are in 500 dimensions.

1 Cluster Linear, KdTree, LSH on Clustered Data

In this section, we compare Cluster Linear, KdTree and LSH on clustered data with different parameters. Each subsection will consist of three parts: First we sketch how the test data is generated, then we plot the results of the test runs and lastly we analyze the results.

We generate the clustered data as described in Chapter 6, Section 10. Thus, we essentially generate data from three parameters: The space which medians are chosen from, the number of clusters and the number of points in each cluster. The space we choose medians from is defined by $[0 : s]$. The number of clusters and points in each cluster is given by n_c and n_p respectively. Throughout all the tests $s = 100$. Data points are chosen from $\mathcal{N}(m, 1)$, where m is a median generated as described in Chapter 6, Section 10.

For each data set, we generate a number of query point sets. Each query point set guarantees that there exists a data point, p, for each query point, q, so that $\|p - q\| \leq v$, for some v. Query points are generated by choosing 100 random points from the data set and for each point generate a query point with distance v to the data point. For every data set, we generate query point sets for all $v \in \{1, 3, 5, \ldots, 149\}$.

Throughout the tests, LSH will be run with the parameters: $w = 400.0$, $k = 10$ and $L = 6$ which ensure we get all data points within $R = 70$ from the query point with success probability 0.9. We have chosen a reasonably low value of L for fairness. Both Cluster Linear and the KdTree have linear space complexity, thus for comparable results LSH should not use much more. The configuration for LSH has been found by empirical experiments.

Cluster Linear needs to know which cluster each point belongs to. Because all data are synthetic, we simply make an index while generating the data.

1.1 Cluster size: 1000

Our first test has parameters $n_c = 100$, $n_p = 1000$. We now proceed to run the algorithms. The results can be found in Figure 8.1.

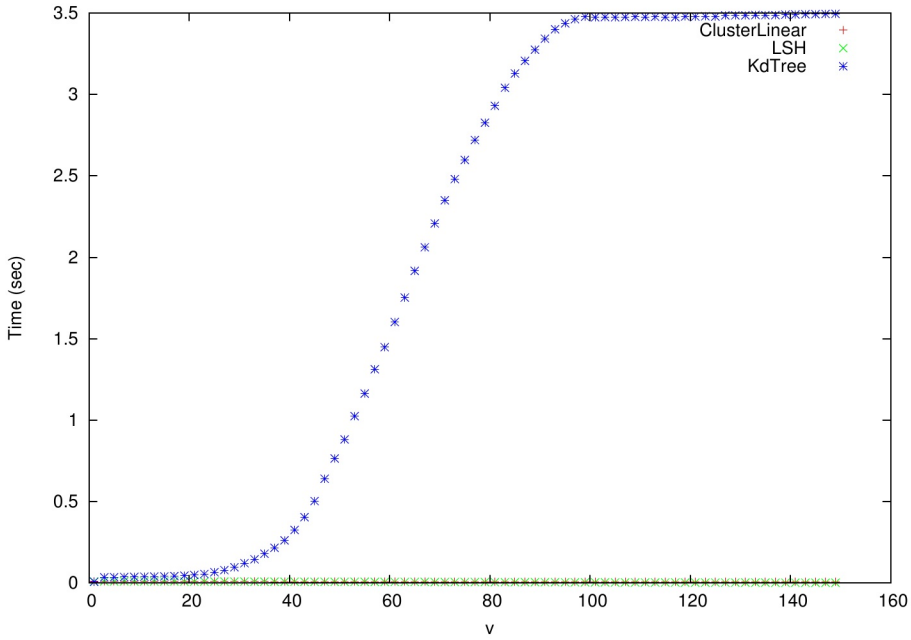


Figure 8.1: The average query time on 100 query points with distance at most v to a data point. The data set consists of 100 clusters, each with 1000 points.

The KdTree quickly degrade to a linear search. Figure 8.2 show the same data but without the KdTree.

Surprisingly, the Cluster Linear achieves better query times as v increases. This speedup should not be attributed to the parameters of the data set given that the method examines a constant number of points in each run. Difference in room temperature, favorable process scheduling and the like must be the reasons for the drop in query time.

LSH also achieves better query times as v increases. As v increases, LSH will get less feedback and thus faster query time. This is supported by Figure 8.3 which shows the recall of Cluster Linear and LSH. Cluster Linear gets 100% no matter how high v gets,

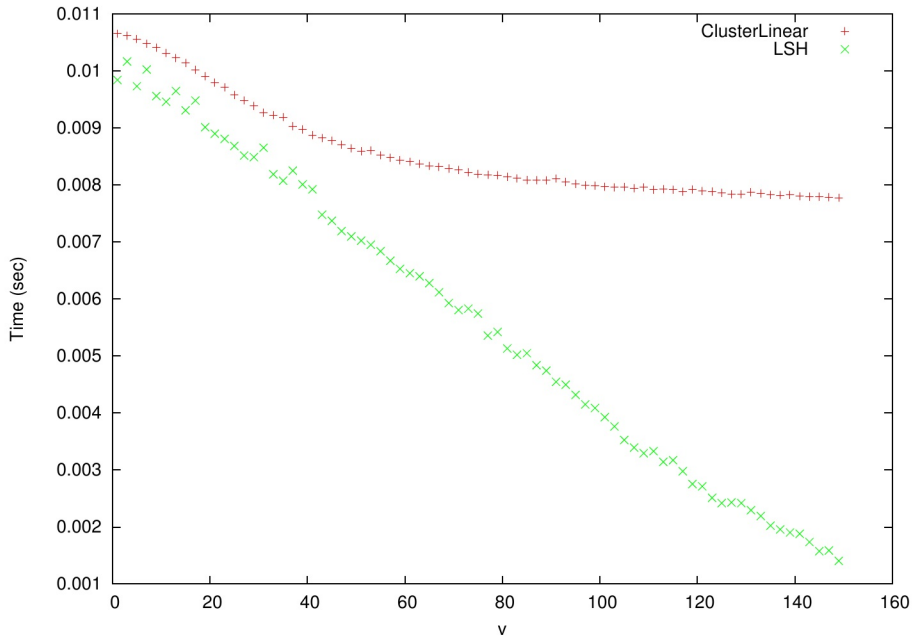


Figure 8.2: The average query time on 100 query points with distance at most v to a data point. The data set consists of 100 clusters, each with 1000 points.

which is surprising. For LSH we would expect at least 90% recall up to $v = 70$. We do not quite get that much accuracy. A higher value of w might change that.

1.2 Cluster size: 100

This test has parameters $n_c = 1000$, $n_p = 100$. We now proceed to run the algorithms. Because the KdTree achieved exactly the same query times as in the first test, the KdTree has been omitted. The results can be found in Figure 8.4.

Because of the smaller clusters, LSH gets less feedback, which speeds up query time. The Cluster Linear examines the same number of points as in the previous test and thus query time is the same. Figure 8.5 shows the recall for both algorithms.

The trend is very similar to the previous test. As the distance to the nearest neighbor increases, the recall drops for LSH. Again, Cluster Linear achieves 100% throughout the test.

1.3 Cluster size: 10

This test has parameters $n_c = 10000$, $n_p = 10$. Because the KdTree achieved exactly the same query times as in the first test, the KdTree has been omitted again. The results can be found in Figure 8.6.

Because the clusters only contain 10 points, LSH gets great query times. Because LSH only examines $\sim n_p$ points, it is evident that LSH benefits from smaller clusters. Figure 8.8 supports this.

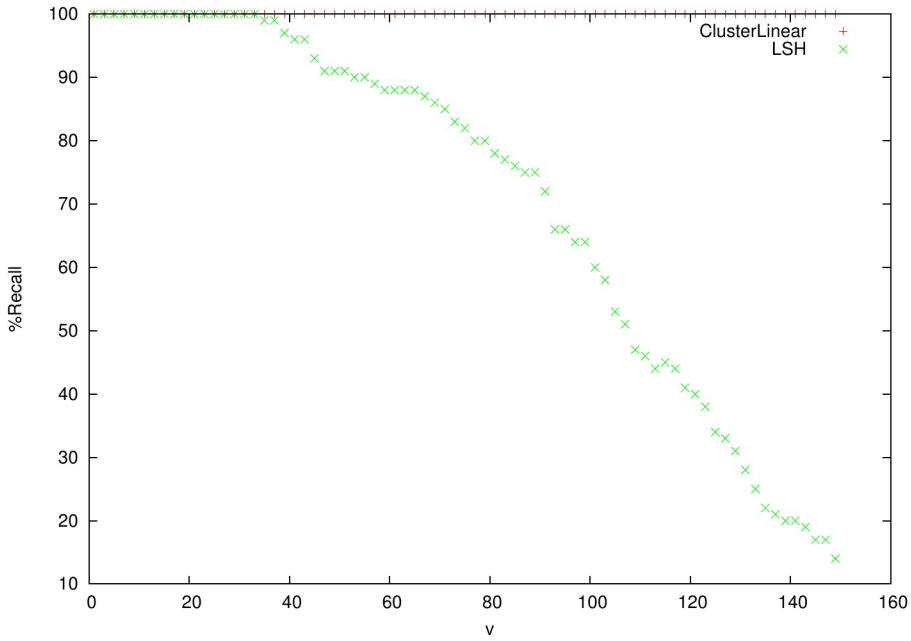


Figure 8.3: The recall on 100 query points with distance at most v to a data point. The data set consists of 100 clusters, each with 1000 points.

Figure 8.7 shows the recall. It is worth noticing how LSH recall seems to drop at about the same place every time. Again, Cluster Linear gets great recall. It seems that clusters need to be closer before the algorithm chooses the wrong cluster to search.

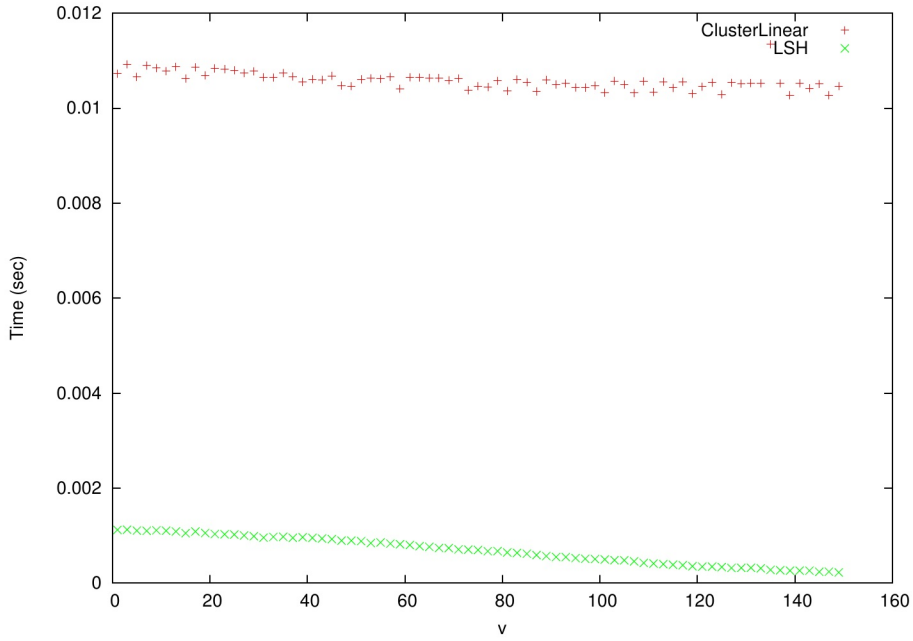


Figure 8.4: The average query time on 100 query points with distance at most v to a data point. The data set consists of 1000 clusters, each with 100 points.

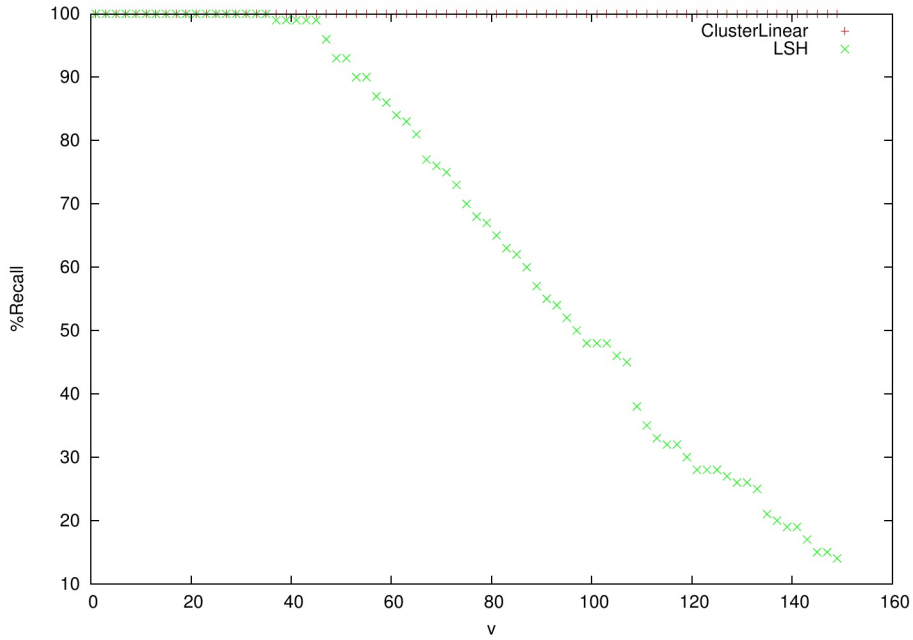


Figure 8.5: The recall on 100 query points with distance at most v to a data point. The data set consists of 1000 clusters, each with 100 points.

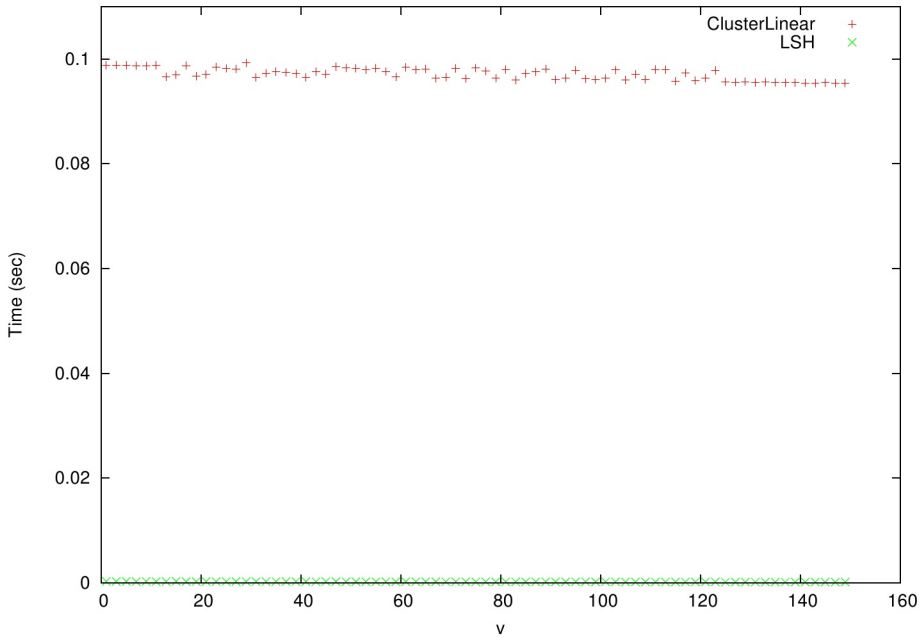


Figure 8.6: The average query time on 100 query points with distance at most v to a data point. The data set consists of 10000 clusters, each with 10 points.

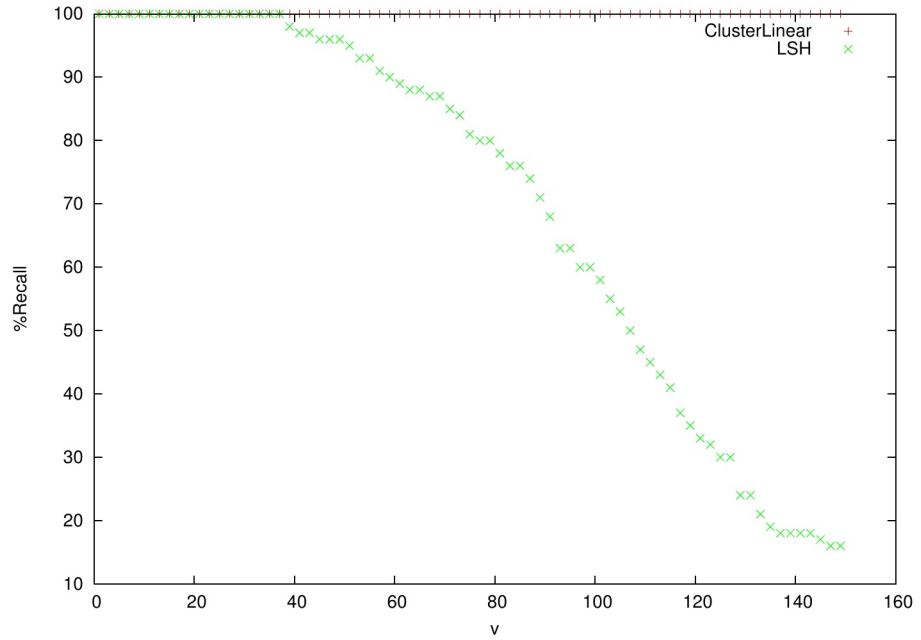


Figure 8.7: The recall on 100 query points with distance at most v to a data point. The data set consists of 10000 clusters, each with 10 points.

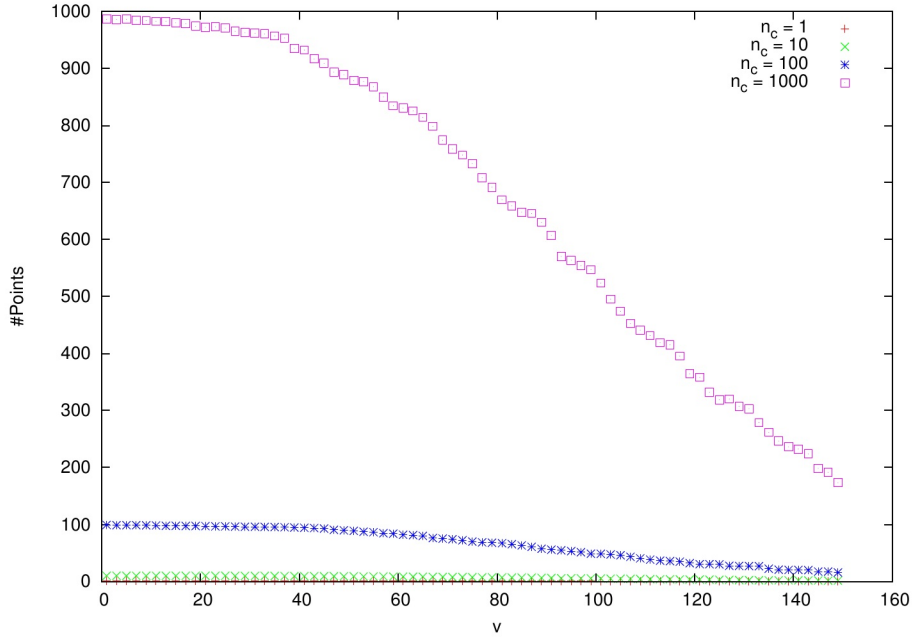


Figure 8.8: The average feedback LSH get on 100 query points with distance at most v to a data point. The data set consists of n_c clusters and a total of 100000 points.

2 KdTree and LSH on Uniform Data

In this section, we compare the KdTree and LSH on uniform data with different parameters.

The data set consists of 100000 points in 500 dimensions. Each point has all coordinates chosen from $[0 : 100]$.

We generate a number of query point sets. Each set guarantee that there exists a data point, p, for each query point, q so that $\|p - q\|_2 \leq v$, for some v. Query points are generated by choosing 100 random points from the data set and for each point generate a query point with distance v to the data point. For every data set, we generate query point sets for all $v \in \{1, 3, 5, \ldots, 149\}$.

Throughout the tests, LSH will run with the same configuration as in the previous section.

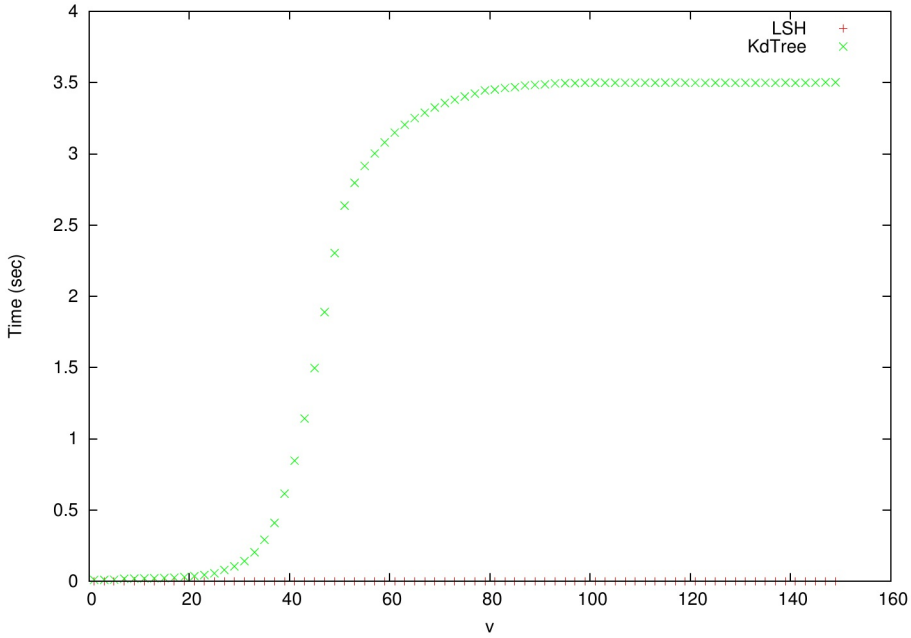


Figure 8.9: The average query time on 100 query points with distance at most v to a data point. The data set consists of 100000 points with coordinates chosen uniform random from $[0 : 100]$.

Figure 8.9 shows the results. The KdTrees do well up until they cannot obtain strong bounds on the distance to the nearest neighbor. LSH is very fast throughout the test.

Figure 8.10 shows the recall for LSH. It seems that it starts to drop a little earlier than in the tests with clustered data. From the configuration we would expect to find the nearest neighbor within $R = 70$ with success probability 0.9. We clearly do not get that high accuracy.

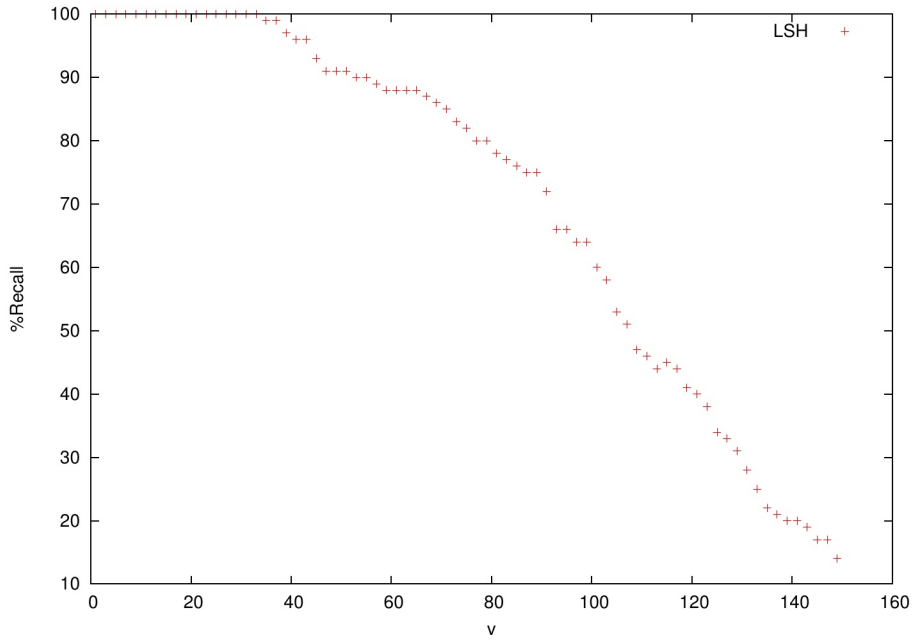


Figure 8.10: The recall on 100 query points with distance at most v to a data point. The data set consists of 100000 points chosen uniform random from $[0 : 100]$.

3 Conclusion

The clustered tests have shown that LSH performs well on clustered data with query points close to a data point. On average, it seems that if the query point is close to a data point we only have to examine a cluster. On uniform data, LSH performs well if query points are close to a data point as well. However, recall drops fast as the distance to the nearest neighbor increases.

The Cluster Linear have comparable query times to LSH on large clusters. As clusters decrease in size, however, LSH will dominate in query time. Cluster Linear does not seem to be as affected by the distance to the nearest neighbor as LSH, in terms of recall.

On both clustered and uniform data, we get a similar result as when we tested KdTrees on data sets with an increasing number of dimensions. They perform well up until a strong bound cannot be obtained in the backtracking and thus degrade to an inefficient linear search.

4 MultiProbe-LSH and LSH on Uniform Data

In this section, we compare the MultiProbe-LSH and LSH on uniform data.

The data set consists of 100000 points in 500 dimensions. Each point has all coordinates chosen at uniform random from $[0 : 1]$ and all points are normalized so that their l_2 norm is equal to one. We use 100 query points generated in exactly the same manner.

We want to compare the two methods on two parameters: speed and quality. We measure speed in average query time over 100 query points in seconds and quality in nearest neighbors found (recall).

We are going to try two configurations for the basic LSH setup. For each configuration on the basic LSH, we are going to test MultiProbe-LSH with 25% and 50% L and test how long a probing sequence we need to attain the same result quality.

All tests have been performed on the Mac Mini described in Appendix A.1.

4.1 R = 1

The basic LSH uses the configuration: $w = 4$, $k = 32$ and $L = 172$, thus providing a 0.9 success probability of finding the nearest neighbor within an $R = 1$ radius. Thus, MultiProbe-LSH will be tested with the same configuration, except for $L = 43$ and 86.

Figures 8.11, 8.12, and 8.13 show the results. Base is basic LSH, everything else is MultiProbe-LSH.

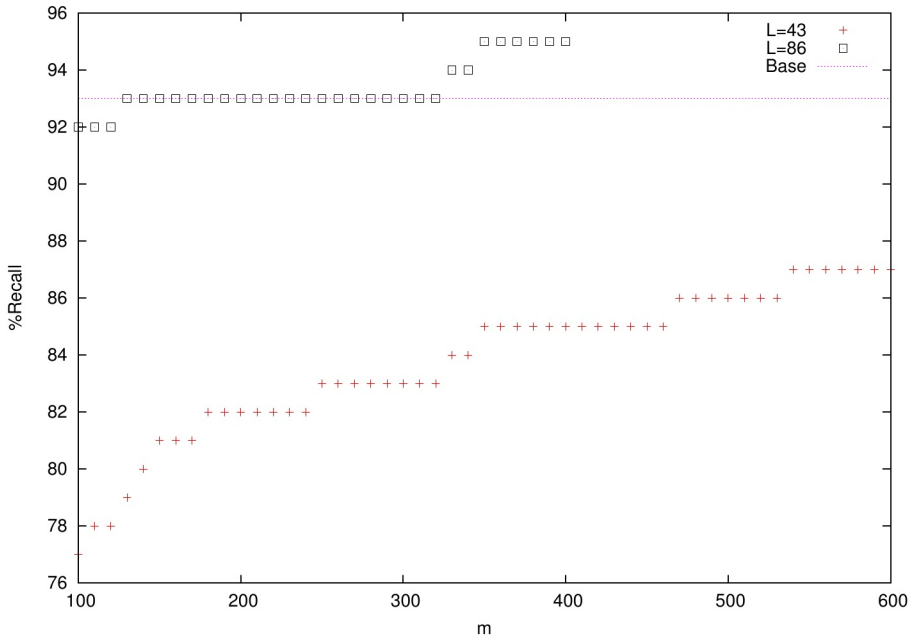


Figure 8.11: The recall on 100 query points in 100000 data points both distributed at uniform random. LSH tuned for $R = 1$. MultiProbe LSH runs with lower L and searches the hashtables m times.

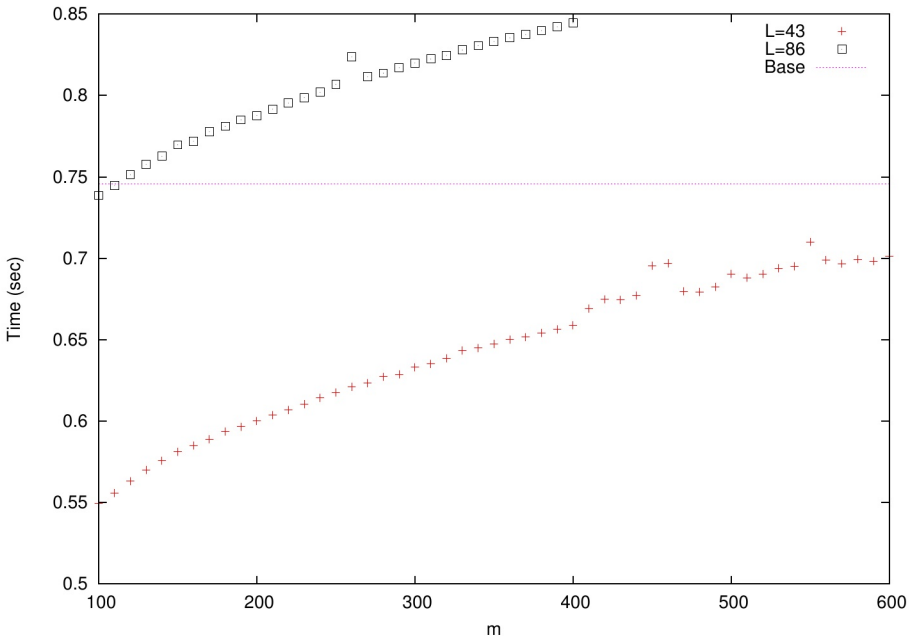


Figure 8.12: The average query time on 100 query points in 100000 data points both distributed at uniform random. LSH tuned for $R = 1$. MultiProbe LSH runs with lower L and searches the hashtables m times.

4.2 R = 0.75

The basic LSH uses the configuration: $w = 4$, $k = 35$ and $L = 101$, thus providing a 0.9 success probability of finding the nearest neighbor within an $R = 0.75$ radius. Thus, MultiProbe-LSH will be tested with the same configuration, except for $L = 25$ and 50.

Figures 8.14, 8.15, and 8.16 show the results.

4.3 Conclusion

It is easy to see that MultiProbe-LSH can attain similar recall as base LSH with 50% space complexity. Query times are higher, but not much. If we only allow 25% space usage recall is worse than base LSH, as well as query times.

It was tempting to believe that we could get linear space complexity and good recall by utilizing a long probing sequence. Our test suggests that this is not the case.

It was also tempting to believe that the size of the feedback was determining the recall. Our test does not indicate that this is the case. MultiProbe-LSH has a larger feedback for a given recall than base LSH. It seems, however, that MultiProbe-LSH handles the larger feedback more efficiently compared to base LSH.

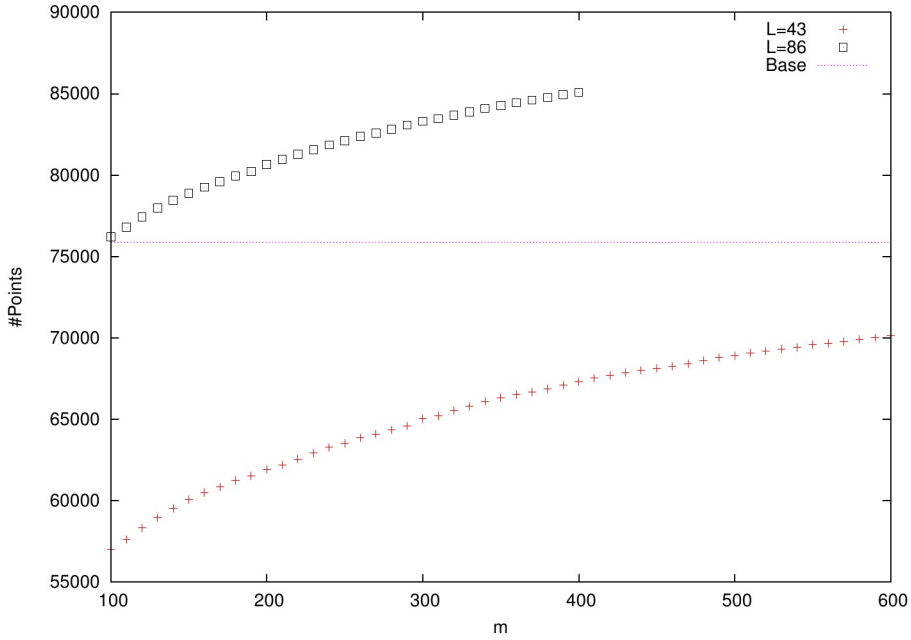


Figure 8.13: The average feedback on 100 query points in 100000 data points both distributed at uniform random. LSH tuned for $R = 1$. MultiProbe LSH runs with lower L and searches the hashtables m times.

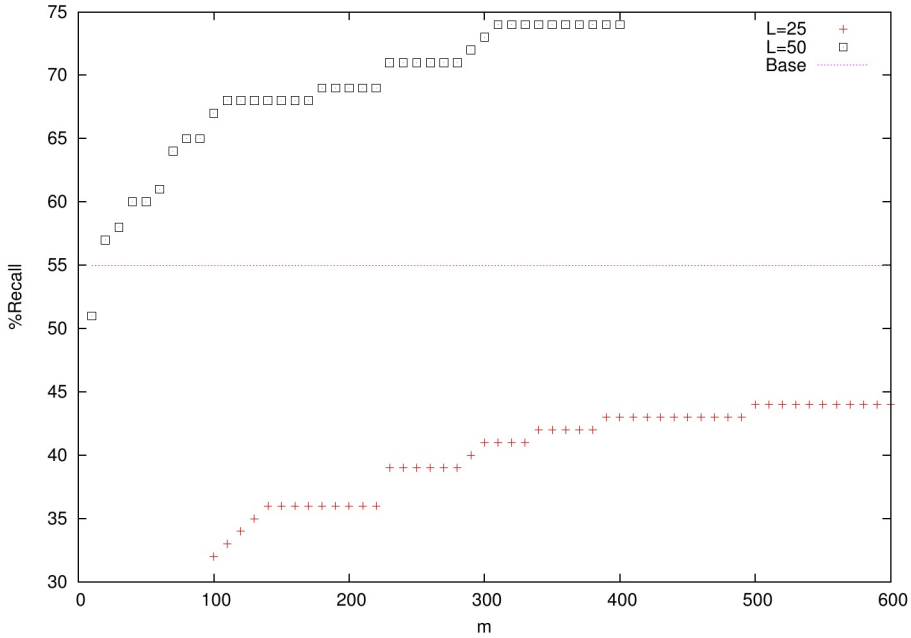


Figure 8.14: The recall on 100 query points in 100000 data points, both distributed at uniform random. LSH tuned for $R = 0.75$. MultiProbe LSH runs with lower L and searches the hashtables m times.

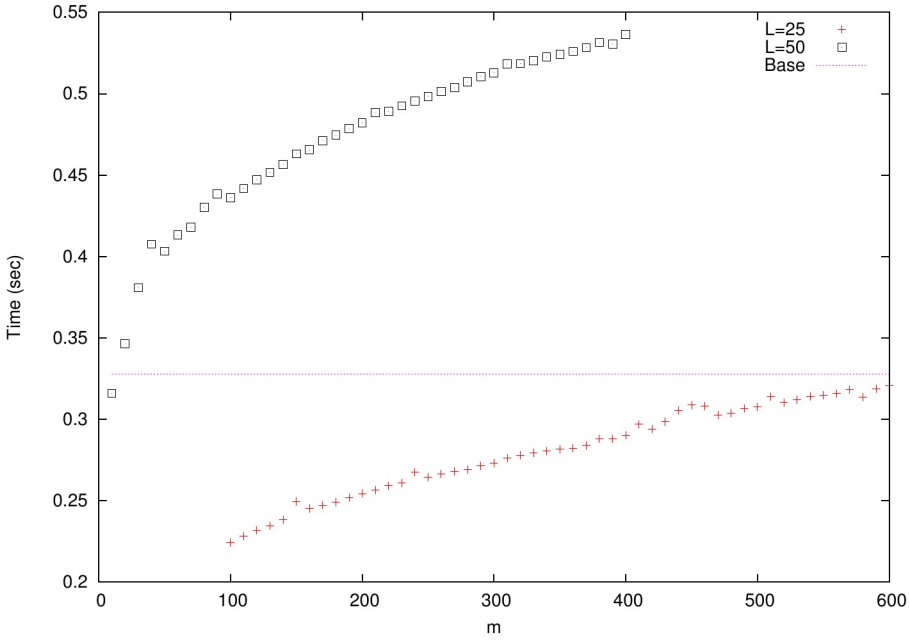


Figure 8.15: The average query time on 100 query points in 100000 data points, both distributed at uniform random. LSH tuned for $R = 0.75$. MultiProbe LSH runs with lower L and searches the hashtables m times.

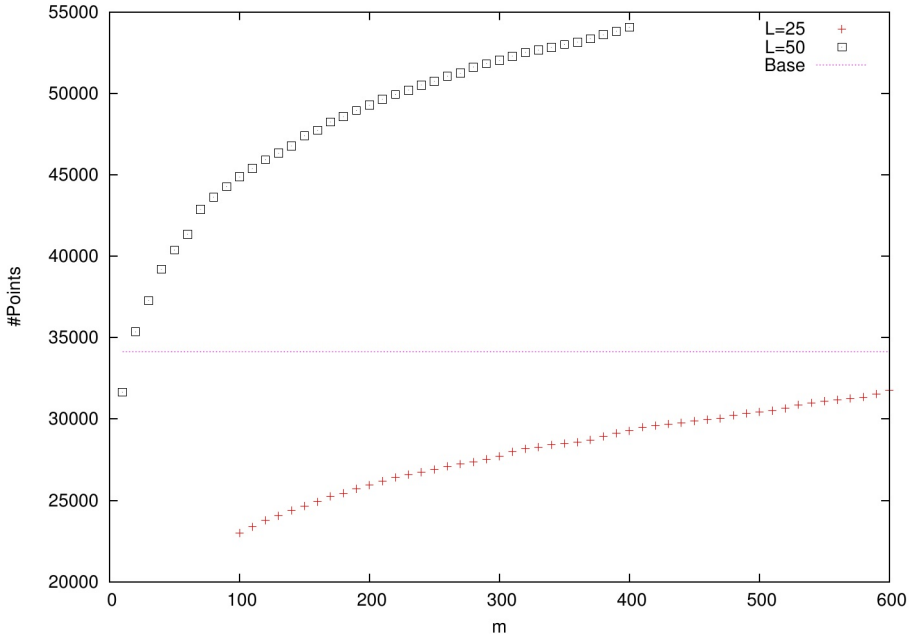


Figure 8.16: The average feedback on 100 query points in 100000 data points, both distributed at uniform random. LSH tuned for $R = 0.75$. MultiProbe LSH runs with lower L and searches the hashtables m times.

9 Conclusion

Throughout this thesis, we have investigated the problem of near neighbor search with the focus on the nearest neighbor problem in Euclidean space over the Euclidean distance function.

The work has evolved around the three base methods: Linear Search, KdTrees and Locality Sensitive Hashing. We have not been able to show absolute dominance for one method. Instead, each method or a variant hereof has shown its worth in one or more areas.

Our contributions are the following.

We have shown how Linear Search can become a competitive algorithm by applying approximation in the form of data domain knowledge. We have also shown that the nearest neighbor problem is not computationally intensive enough to be suited for GPU-based solvers. The all pairs nearest neighbor problem may be, however.

We have confirmed results from [WSB98] and [Moo91] by examining how KdTrees and R-Trees degrade to an inefficient linear search when dimensionality is sufficiently high. As an addition to the results from [WSB98], we have shown that placing query points close to data points will improve query times greatly for KdTrees in high dimensional spaces. Thus, demonstrating how it is not dimensionality itself which makes KdTrees degrade but the larger distances between query and data points induced by high dimensionality. We have also shown how KdTrees can be used for approximate nearest neighbor search by disabling or limiting backtracking.

We have shown how LSH on uniform random data is not much of an improvement over a crude linear search. We saw how LSH had to process at least 75% of the data set to find the nearest neighbor on uniform data with uniform query points. If query points are guaranteed to be close to a data point, however, LSH is very fast. We have shown that LSH works very well for clustered data with query points placed in a cluster. As a by-product of our tests on clustered data, we have shown how LSH can be used as an algorithm for identifying clusters in data sets.

1 Method Recommendation

In section we present some of our findings as a method recommendation for nearest neighbor search.

1.1 Low dimensionality

If the number of dimensions is below 10, a space partitioning tree is preferable. As shown in the tests on the KdTree, the SP-trees perform very well in low dimensions. The number 10 is not a magic number, however, it is merely a rule of thumb deduced from our tests. See Figures 4.4 and 4.5 for the details.

SP-trees are generally easy to implement. Furthermore, they are well studied and deliver exact nearest neighbor search.

1.2 Query points close to data point

If we are only interested in finding the nearest neighbor when the query point is close to a data point, with respect to the distribution of the data points, LSH will provide great query times. When LSH is tuned with a small R, we only examine a small fraction of the data set as seen in Figure 8.8.

However, it should be made perfectly clear that the implementation of LSH can be a bit tricky. Configuration is tricky as well.

Implementations of LSH such as E^2LSH[And] and LSHKIT[wdo] exist, but even with a solid theoretical knowledge of how LSH works, they are not easy to use.

If the data is clustered into large clusters, the Cluster Linear provides good query times as well as ease of implementation. In general data domain knowledge may help speed up a linear search.

1.3 Exact nearest neighbor

If an exact nearest neighbor is needed even when query points are placed far from data points in a high dimensional space, the linear search seems to be the only choice. To speedup the search methods such as parallelization can be employed.

Implementation is straightforward and speedups can be attained by applying different tactics.

A side note from Chapter 7 about GPU implementation, is that the language in which the linear search is implemented matters when it comes to query times. The C version of the linear search made in Chapter 7 is much faster than the Java implementation used elsewhere.

Bibliography

[AI08] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.

[And] Alexandr Andoni. LSH algorithm and implementation (E^2LSH). `http://www.mit.edu/~andoni/LSH/`.

[And05] Alexandr Andoni. E^2LSH user manual. `http://www.mit.edu/~andoni/LSH/manual.pdf`, 2005. [Online; accessed 18-June-2010].

[BCG05] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. LSH forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web*, WWW '05, pages 651–660, New York, NY, USA, 2005. ACM.

[Blu] Manuel Blum. Linear-time selection (randomized and deterministic). `http://www.cs.cmu.edu/afs/cs/academic/class/15451-s07/www/lecture_notes/lect0125.pdf`. [Online; accessed 15-April-2010].

[CMS76] J. M. Chambers, C. L. Mallows, and B. W. Stuck. A method for simulating stable random variables. 71(354):340–344, 1976.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[FBF77] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3:209–226, September 1977.

[GDB08] V. Garcia, E. Debreuve, and M. Barlaud. Fast k nearest neighbor search using GPU. In *CVPR Workshop on Computer Vision on GPU*, Anchorage, Alaska, USA, June 2008.

[GLL98] Yván J. García, Mario A. López, and Scott T. Leutenegger. A greedy algorithm for bulk loading R-trees. In *GIS '98: Proceedings of the 6th ACM international symposium on Advances in geographic information systems*, pages 163–164, New York, NY, USA, 1998. ACM.

[Gut84] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14:47–57, June 1984.

[Ind00] P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computation. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 189, Washington, DC, USA, 2000. IEEE Computer Society.

[KL97] Joseph K. P. Kuan and Paul H. Lewis. Fast k nearest neighbour search for R-tree family. In *In Proceedings on First International Conf. on Information, Communications, and Signal Processing. Singapore.*, pages 924—928, September 1997.

[LC] Yann LeCun and Corinna Cortes. The MNIST database of handwritten digits. `http://yann.lecun.com/exdb/mnist/`. [Online; accessed 14-September-2010].

[LEL97] Scott T. Leutenegger, J. M. Edgington, and Mario A. Lopez. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings of the Thirteenth International Conference on Data Engineering*, ICDE '97, pages 497–506, Washington, DC, USA, 1997. IEEE Computer Society.

[LJW+07] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 950–961. VLDB Endowment, 2007.

[Moo91] Andrew Moore. A tutorial on kd-trees. Extract from PhD Thesis, 1991. Available from `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.10.818`.

[nVi] nVidia. NVIDIA CUDA C programming guide, version 3.2. `http://developer.nvidia.com/object/cuda_3_2_downloads.html`. [Online; accessed 04-November-2010].

[Pan06] Rina Panigrahy. Entropy based nearest neighbor search in high dimensions. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 1186–1195, New York, NY, USA, 2006. ACM.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing*. Cambridge University Press, New York, NY, USA, 1992. `http://apps.nrbook.com/c/index.html`.

[QMN09] Deyuan Qiu, Stefan May, and Andreas Nüchter. GPU-accelerated nearest neighbor search for 3d registration. In *Proceedings of the 7th International Conference on Computer Vision Systems: Computer Vision Systems*, ICVS '09, pages 194–203, Berlin, Heidelberg, 2009. Springer-Verlag.

[RKV95] Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM-SIGMOD Intl. Conf. on Management of Data*, San Jose, 1995.

[SDI06] Gregory Shakhnarovich, Trevor Darrell, and Piotr Indyk. *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice (Neural Information Processing)*, chapter 4. The MIT Press, 2006. LSH using stable distributions.

[wdo] Wei Dong wdong. LSHKIT: A C++ locality sensitive hashing library. `http://lshkit.sourceforge.net/`. [Online; accessed 15-March-2011].

[WSB98] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[Zol86] V. M. Zolotarev. One-dimensional stable distributions. In *Translations of Mathematical Monographs*, volume 65. American Mathematical Society, 1986.

A

Test Machines

Here the specifications of the computers used for testing is presented.

1 Mac Mini

The Mac Mini is a personal media center used at home. It should reflect a standard desktop pc.

Here the relevant info from `/usr/sbin/system_profiler` is presented.

```
Hardware:

    Hardware Overview:

      Model Name: Mac mini
      Model Identifier: Macmini3,1
      Processor Name: Intel Core 2 Duo
      Processor Speed: 2.26 GHz
      Number Of Processors: 1
      Total Number Of Cores: 2
      L2 Cache: 3 MB
      Memory: 2 GB
      Bus Speed: 1.07 GHz
      Boot ROM Version: MM31.00AD.B00
      SMC Version (system): 1.35f1
      Serial Number (system): YM008C6J9G5
      Hardware UUID: DDA420E4-3C5B-50FB-B4C8-13E4638F25A1
```

```
Graphics/Displays:

    NVIDIA GeForce 9400:

      Chipset Model: NVIDIA GeForce 9400
      Type: GPU
      Bus: PCI
```

```
8       VRAM (Total): 256 MB
9       Vendor: NVIDIA (0x10de)
```

Details from `http://www.nvidia.com/object/product_geforce_9400m_g_us.html` on the graphics card:

```
1  Processor Supported:    Core 2 Extreme
2                          Core 2 Duo
3  CUDA Cores: 16
4  Memory Interface:   128-bit
5  Memory Bandwidth (GB/sec):  21
6  OpenGL: 2.1
```

```
1  Memory:
2
3      Memory Slots:
4
5        ECC: Disabled
6
7          BANK 0/DIMM0:
8
9            Size: 1 GB
10           Type: DDR3
11           Speed: 1067 MHz
12           Status: OK
13           Manufacturer: 0x80CE
14           Part Number: 0x4D34373142323837334548312D4346382020
15           Serial Number: 0x82ED012A
16
17         BANK 1/DIMM0:
18
19           Size: 1 GB
20           Type: DDR3
21           Speed: 1067 MHz
22           Status: OK
23           Manufacturer: 0x80CE
24           Part Number: 0x4D34373142323837334548312D4346382020
25           Serial Number: 0x82ED00BA
```

`java -version` gives us

```
1  java version "1.6.0_23"
2  Java(TM) SE Runtime Environment (build 1.6.0_23-b05)
3  Java HotSpot(TM) Server VM (build 19.0-b09, mixed mode)
```

`javac -version` gives us

```
1  javac 1.6.0_23
```

2 Fafner

Fafner is located at IMADA, SDU and can be used by anyone at the institute. The computer is used by many students and is therefor unsuited for tests which require timing.

The machine have four cores, which are all configure as follows:

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Xeon(R) CPU           X3210  @ 2.13GHz
stepping        : 11
cpu MHz         : 2128.000
cache size      : 4096 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 10
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
                  cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss
                  ht tm pbe syscall nx lm constant_tsc arch_perfmon pebs
                  bts rep_good pni dtes64 monitor ds_cpl vmx est tm2
                  ssse3 cx16 xtpr pdcm lahf_lm tpr_shadow vnmi
                  flexpriority
bogomips        : 4266.81
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

`free -m` gives us:

```
             total       used       free     shared    buffers     cached
Mem:          7999       1547       6452          0         83        131
-/+ buffers/cache:       1332       6667
Swap:         7632          0       7632
```

`java -version` gives us

```
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_19-b02)
Java HotSpot(TM) 64-Bit Server VM (build 1.5.0_19-b02, mixed mode)
```

`javac -version` gives us

```
javac 1.5.0_19
[..]
```

B

Success Probability in LSH

Given a radius, R, we want to be able to choose k and L in order to report every point within a radius of R from the query point with success probability $1 - \delta$. We chose

$$L = \left\lceil \frac{\log \delta}{\log\left(1 - P_1^k\right)} \right\rceil \tag{B.1}$$

and k to minimize query time. We know $1 - \delta$. Furthermore, we know that $P_1 = p(R)$ where

$$p(u) = \int_0^w \frac{1}{u} f_s\left(\frac{t}{u}\right)\left(1 - \frac{t}{w}\right) dt \tag{B.2}$$

where $f_s(\cdot)$ is the absolute value of the probability density function of the s-stable distribution we are using in our hash functions. Assuming we are using the normal distribution, we can approximate the integral.

If we set $x = w/R$, then

$$P_1 = 1 - erfc\left(\frac{x}{\sqrt{2}}\right) - \frac{2}{\sqrt{\pi}}/\sqrt{2}/x * 1 - e^{-1*\sqrt{x}/2} \tag{B.3}$$

The formula has been extracted from the E^{2}LSH[And] source code and will not be accounted for here.

The $erfc(\cdot)$ function is an error correcting function, which can be found in [PTVF92].

Source Code

The source code for the implementations and a pdf of this thesis can be found at `http://runlevel0.dk/thesis/`.