Comparing Nearest Neighbor Algorithms in High-Dimensional Space

Hendra Gunadi

College of Engineering and Computer Science
Australian National University

November 2011

Abstract

Nearest neighbor search, a problem that asks for a nearest point in the database given a query, is a problem in areas of computer science such as information retrieval, pattern recognition, image, and text processing. There is still ongoing research toward the improvement of the performance. Despite that, there is no comparison between methods proposed. We have no idea whether a method could work in high dimension or not, or how about the time and space complexity, or what about the results returned by specific method. In this paper, I compare six of them here, they are Exhaustive, Vantage Point, Random Projection Matrix, Random Projection Tree (RP Tree), Random Ball Cover (RBC), and Locality-Sensitive Hashing (LSH). This paper consists of comparisons between each method mentioned so that readers could see some characteristics of these methods to solve nearest neighbor search problem. In this paper, I use two distance metrics, that is Euclidean and Cosine Similarity, to compare the performance of each method.

Contents

List of Figures

Chapter 1

Introduction

Nearest neighbor search is a problem in areas of computer science such as information retrieval, image processing, data mining, pattern recognition, and text processing. There are methods proposed during the last few years that attempts to improve the performance so as to solve the problem. Yet research continues since 1970, signalling its importance.

Usually, the objects of interest in each of these fields are represented as points that lie in $\mathbb{R}^d$ dimensional spaces. Then, using distance metric to calculate the distance between points, the focus of the inquiry now becomes searching the object that is similar to the query (nearest neighbor search). An example for this representation is when we do document processing where we can represent each word as a dimension, and the number of occurrences as the value in that respective dimension. Then, using cosine similarity as the distance metric, we can find the most similar document to the document that is in query. If a person read a certain document (query document), then it is likely that he will be interested in reading similar document. Since we already have that similar document, we can suggest him to read it as well.

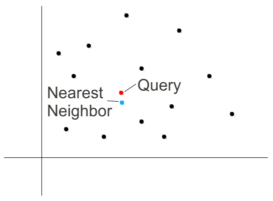


Figure 1.1: Example of nearest neighbor search in 2D space

Here are some methods that I compare in this paper:

- Prune search space:
  Some methods try to prune out the search space based on certain condition(s). So, for the given query we do not have to compare all points to find the nearest one. Example of methods compared in this paper that uses this approach are:
  - Vantage Point; and
  - Random Ball Cover (RBC).
- Data structure:
  Some other methods try to construct a data structure using given database. There will be some overheads in constructing the data structure from the database, but the resulting efficiency for the search will pay off. Examples that belong to this category mostly attempt to build a tree, using certain criteria to split off the points to the branches. Then, by applying that criteria to the query, we find out whether we need to visit a branch or not. A method that belongs to this criteria that is compared in this paper is the Random Projection Tree (RP Tree).
- Random subspace:
  There are methods that rely on the randomized subspace property, that is, we can embed a point in high dimensional space to random lower dimensional subspaces while preserving the distance. Example of methods compared in this paper that uses this approach are:
  - Random Projection Matrix; and
  - Random Projection Tree (RP Tree).
- Hashing:
  In recent years, several researchers have proposed methods for overcoming the running time bottleneck by using approximation. In this formulation, the algorithm is allowed to return a point which is close enough from the query to be considered "nearest neighbor". The appeal for this approach is that, in many cases, an approximate nearest neighbor is almost as good as the exact one [6] i.e. we do not really need the nearest point possible. It is acceptable as long as the point returned is close enough. A method that belongs to this category which is described in this report is the Locality-Sensitive Hashing (LSH).

With so many available methods out there, one could easily drown in information about methods to solve a nearest neighbor problem. Furthermore, if we are to work in a high-dimensional problem, there is a possibility that we choose a method which does not work in that environment. That is the main goal for this paper, to provide comparison between methods to give a sense to readers about which one to choose.

The rest of the paper is organized as follows : Section 2 gives the description of how the method works for each method I compare. The experiment results and comparison between methods will be presented in the Section 3. Following that section will be conclusions. The comparison in Section 3 will be using these control variables:

- the size of dataset
- the number of dimension

and compare them to these metrics:

- time
- space (memory usage)
- error

Chapter 2

Background

Nearest neighbor search is a problem comprising a collection of points in the database, in which we try to return the nearest point(s) for given query that lies in the same space. If only one returned point is required, then the problem is called nearest neighbor. But if there are more than one returned points required, then the problem is called k-nearest neighbor where k is the number of returned points required. The point can lie in an arbitrary number of dimensions and any space. But to be considered an instance of this problem, those points must have the same distance metric and the same number of dimensions. For a distance to be a metric, it has to satisfy the following conditions [1]:

- $d(x, y) \geqslant 0$, and $d(x, y) = 0$ if and only if $x = y$
- it is symmetric : $d(x, y) = d(y, x)$ (The distance between x and y is the same in either direction).
- it satisfies the triangle inequality : $d(x, z) \leqslant d(x, y) + d(y, z)$

The distance metric is then used to calculate the distance / similarity between points.

Formally, the nearest neighbor problem is defined as follows : Given a collection of points DB and a point q that lies in $\mathbb{R}^d$, find $argmin(dist(p, q)), p \in DB$ where $argmin(.)$ is a function that returns the index of the minimum value in the list given as parameter and $dist(.,.)$ is the distance between any two points that is calculated using distance metric (euclidean or cosine similarity). For euclidean, the distance is given as $dist(a, b) = \sqrt{\Sigma_i^d (a_i - b_i)^2}$ and for cosine similarity, the distance is given as $dist(a, b) = \frac{a.b}{\|a\|\|b\|} = \frac{\Sigma_i^d a_i \times b_i}{\sqrt{\Sigma_i^d (a_i)^2 \times \Sigma_i^d (b_i)^2}}$. And then we can extend this definition to k-nearest neighbor by finding by modifying $argmin(.)$ to return k indices.

The first thing that comes to mind when we are discussing about nearest neighbor search is the exhaustive search. This method of searching compares the query point(s) with each of the points in the database. And usually, the distance metric is using the whole dimension. So, the complexity of exhaustive search is $O(n.d)$ where n is the number of points in the database and d the number of dimensions of the problem. Now, we are going to discuss about some other methods to solve nearest neighbor problems.

2.1 Prune State Space Search

Basically, the techniques that falls in this category attempt to prune the search state space so that we do not have to explore the whole database when we search the nearest neighbor. The way to prune the state is different for each technique, but there is something in common between the methods I compare in that we choose representative(s), then we prune the search state space based on the distance of the query and the representative(s). There are two techniques from this category compared in this paper : Vantage Point and Random Ball Cover (RBC).

2.1.1 Vantage Point

Vantage Point is initialized by choosing a random point in the database (including origin) as the Vantage Point. After the reference point is chosen, the points in the database is sorted based on the distance to the reference point. Then, for every incoming query, the distance of the query to the Vantage Point is calculated. This distance is then used as a reference to find the point in database which has the most similar distance to Vantage Point.

The searching then goes to left and right of that similar point in the sorted database. The search then stops if the minimum distance to the query we found so far has already within the largest difference in distance to Vantage Point. Graphical representation of what is happening in this algorithm is shown in Figure 2.1.

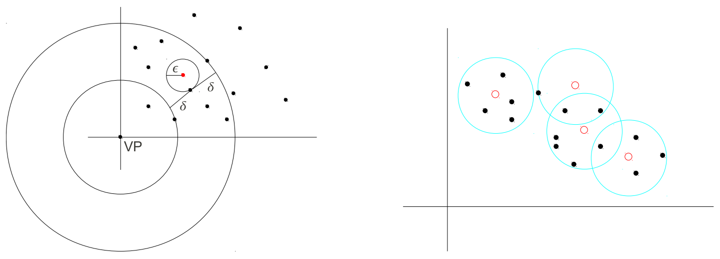
The algorithm is given as follows:

Algorithm 1 *Vantage Point initialization*

```
$vp \leftarrow$ random point in database (including origin)
$distlist \leftarrow$ all points sorted according to $dist(p, vp)$, where $dist(.,.)$ is the
distance function between any two points and $p$ is each point accordingly
```

Algorithm 2 *Vantage Point searching*

```
$dist \leftarrow$ distance from query to $vp$
$ref \leftarrow$ search $distlist$ for point with the most similar value to $dist$
$\delta \leftarrow 0$ // distance of the query to ref
$\epsilon \leftarrow dist$ // minimum distance we found so far
while $\epsilon > \delta$ do
    $now \leftarrow$ index of right / left in $distlist$ with $min(abs(value - distlist[ref].value))$, where $min(.)$ is a function that returns the minimal value in the list given as parameter and $abs(.)$ is the absolute value of the parameter.
    $\delta \leftarrow distlist[now].value - ref$
    if $\epsilon > distlist[now].value$ then
        $\epsilon \leftarrow distlist[now].value$
        $ID \leftarrow distlist[now].ID$
    end if
end while
return $ID$
```



(a) stopping condition for Vantage Point

(b) Random Ball Cover

Figure 2.1: Graphical representation of the methods

2.1.2 Random Ball Cover [2]

Random Ball Cover is initialized by choosing a number of random representatives. After we initialize the representatives, we construct the lists to indicate the points that belong to each of the representatives. Then, for each of the query that comes, we calculate the distance to each of the representatives using the exhaustive search. There are two variances of this algorithm, though the difference lies only in the way they do the search based on the distance to representatives. The first one is called One-Shot algorithm. The way it works is to take the nearest representative, then an exhaustive search is done for the points that belongs to this representative. This one is not guaranteed to return the nearest point possible as there is a probability that the nearest point belongs to another representative.

The other variance is called Exact algorithm, and this is the one that I use in the comparison. This one works by trying to prune as many representatives as possible instead of taking the nearest one as in the One-Shot algorithm. Then, an exhaustive search is done for the points that belongs to the representative(s) that can not be pruned. Let γ to be distance to the closest representative, then for a representative to be included in the list of searching, the representative must lie within 3γ from the query. The algorithm is given as follows:

Algorithm 3 *Random Ball Cover initialization*

```
Require: $n$ : number of representatives
  $R \leftarrow$ list of $n$ random points in database chosen as representatives
  $L_r \leftarrow$ list of points that belongs to representative $r \in R$
```

Algorithm 4 *Random Ball Cover Exact searching*

```
Require: : $q$ : query point
  $\gamma \leftarrow min(dist(q,r)), r \in R$, where $min(.)$ is a function that returns the
  minimal value in the list given as parameter and $dist(.,.)$ is the distance
  between any two points.
  $X \leftarrow$ empty list
  for $r \in R$ do
    if $d(q,r) < 3\gamma$ then
      $X \leftarrow X + L_r$
    end if
  end for
  $ID \leftarrow L_r[argmin(dist(q,x))], x \in X$, where $argmin(.)$ is a function that
  returns the index of the minimal value in the list given as parameter.
  return $ID$
```

In this paper, I used the code that Lawrence Cayton (the author of the paper) provided in [3]. I slightly modified the code though to add the functionality of measuring the memory usage. Because the code does not support the cosine similarity distance metric, this algorithm is not included in the comparison for the cosine similarity. But it is still included in the comparison for the euclidean distance metric.

2.2 Random Projection

Techniques that falls to this category is using Johnson-Lindenstrauss theorem that says for any n point subset of Euclidean space can be embedded in $k = O(\log n/\epsilon^2)$ dimensions without distorting the distances between any pair of points by more than a factor of $(1 \pm \epsilon)$, for any $0 < \epsilon < 1$ [4]. This will imply that for any pair of points, if they were close in the original space, they will remain close in projected space.

Here in this paper, I am going to compare two techniques that falls to this category. One is by constructing a projection matrix to project the database and the query to the lower dimension. The other one is Random Projection Tree (RP Tree), tree based structure that splits the database depending on the resulting projection on a randomly created vector.

2.2.1 Random Projection Matrix

Initially pick $k << d$ as the target dimension. We then proceed by creating a $k \times d$-matrix whose entries are random values, and each of the columns is a unit vector. After we create the matrix, we project (by doing matrix multiplication) the database which originally lies in d-dimension to k-dimension. Then, for each of the incoming query, we just need to project the query point to k-dimension, after which we can use any technique that works in low dimension to find the nearest neighbor. For implementation of this approach in this paper, I used exhaustive search to find the nearest neighbor after the projection because exhaustive search is guaranteed to return the nearest neighbor.

The algorithm implemented in this paper is described as follows :

Algorithm 5 *Random Projection Matrix initialization*

```
Require: $d$ : original dimension, $k$ : desired dimension, $DB$ : $d \times n$-matrix
  $projM \leftarrow$ a matrix of $k \times d$ random values
  make each of $projM$ column-entries to be unit vector
  $projDB \leftarrow projM \times DB$
```

Algorithm 6 *Random Projection Matrix searching*

```
Require: $q$ : query point
  $projQ \leftarrow projM \times q$
  $ID \leftarrow projDB[argmin(dist(projQ, p))], p \in projDB$, where $argmin(.)$ is
  a function that returns the index of the minimal value in the list given as
  parameter and $dist(.,.)$ is the distance between any two points..
  return $ID$
```

2.2.2 Random Projection Tree [5]

Random Projection Tree works just like other tree-based techniques, that is, recursively splits the points in the database to the child nodes depending on certain criteria. There are two variants that Sanjoy Dasgupta and Yoav Freund (the authors) provide, but in this paper I only implement the first variant. The splitting criteria for Random Projection Tree is initialized by creating a random unit vector (a direction is chosen at random from the unit sphere). After we have the random unit vector, all the points in this particular node is projected to that vector. After we have finished with the splitting of the database, the random unit vector still can not be thrown away because it is used to project the incoming query.

A point will then go to the left child node if its projection length is less than the median value, or otherwise go to the right child node. We have to store the median value as well, to decide whether we will need to inspect the left child node or the right child node for a given query. There is a slight difference with the original algorithm though, in that I do not add the perturbation as criteria to split the database. We stop splitting the database if the number of points is small enough (in this implementation, I stopped at 5), or if there is a child node which has no point.

For every query against the database, the point is first projected against the random unit vector in the inspected node (initially the root). Then, based on the length of the projection, we will inspect the left child node if its length is less than the median, or the right child node otherwise. We do this inspection recursively until we reach a leaf node. Then we can use the exhaustive search to find the nearest neighbor as the number of points in the leaf node is already small enough.

The algorithm implemented in this paper is described as follows :

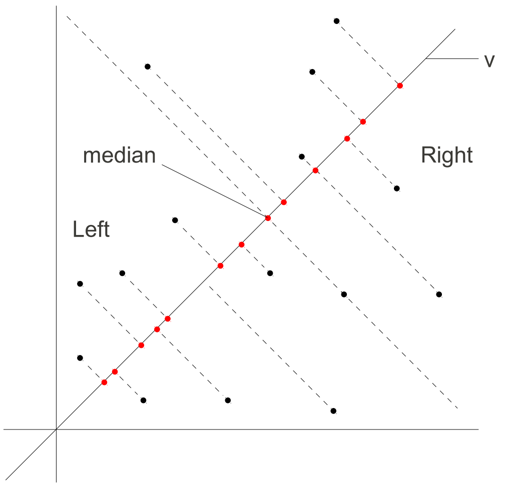
Algorithm 7 *MakeTree*

```
Require: $S$ : list of points
  $LeftPoints, RightPoints \leftarrow$ empty list
  if $|S| < MinSize$ then
    return Leaf
  end if
  $v \leftarrow$ random unit vector (splitter)
  $median \leftarrow median(dot(s, v)), s \in S$, where $median(.)$ is a function that returns the median value of the list given as parameter and $dot(.,.)$ is a function that returns the dot product of two vectors.
  for $p \in S$ do
    if $p.v \leqslant median$ then
      $LeftPoints \leftarrow LeftPoints + p$
    else
      $RightPoints \leftarrow RightPoints + p$
    end if
  end for
  $LeftTree \leftarrow MakeTree(LeftPoints)$
  $RightTree \leftarrow MakeTree(RightPoints)$
```

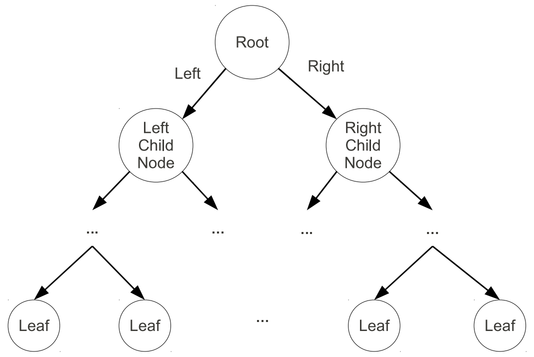
*Notes) Originally, the condition is $p.v \leqslant median + \delta$ instead of $p.v \leqslant median$, where δ is defined as follows : pick a point $y \in S$ farthest from p, then choose δ uniformly at random in $[-1, 1].6 \parallel p - y \parallel \sqrt{D}$

Algorithm 8 *RP Tree searching*

```
Require: $q$ : query point
  if $|S| < MinSize$ then
    $ID \leftarrow$ nearest neighbor by calling $\mathbf{BF}(q, S)$
    return $ID$
  end if
  if $q.v \leqslant median$ then
    $ID \leftarrow$ do search on left branch
  else
    $ID \leftarrow$ do search on right branch
  end if
  return $ID$
```



(a) Split points to left or right child node



(b) RP Tree data structure

Figure 2.2: Graphical representation of the RP Tree

2.3 Hashing [6, 7]

The hashing algorithm that I compared in this paper called Locality-Sensitive Hashing (LSH). The key idea is to hash the points using several hash functions to ensure that for each function the probability of collision is much higher for objects that are close to each other than for those that are far apart. Then, one can determine near neighbors by hashing the query point and retrieving elements stored in buckets containing that point.

This algorithm relies on existence of *locality-sensitive hash functions.* Let H be a family of hash functions mapping $\mathbb{R}^d$ to some universe U. For any two points p and q, consider a process in which we choose a function h from H uniformly at random, and analyze the probability that $h(p) = h(q)$. The family H is called *locality sensitive* (with proper parameters) if it satisfies the following condition.

Definition 2.3 (Locality-sensitive hashing). A family H is called (R, cR, P_1, P_2)-sensitive if for any two points $p, q \in \mathbb{R}^d$.

- if $\| p - q \| \leqslant R$ then $Pr_H[h(q) = h(p)] \geqslant P_1$,
- if $\| p - q \| \geqslant cR$ then $Pr_H[h(q) = h(p)] \leqslant P_2$,

In order for a locality-sensitive hash(LSH) family to be useful, it has to satisfy $P_1 > P_2$.

Usually, the gap between P_1 and P_2 is quite small. So we need an amplification process to achieve the desired probabilities of collision. We can amplify this gap between P_1 and P_2 by concatenating several functions. So, when we choose function to hash the point, we are not choosing one particular function. Instead, we choose L hash functions $g_j = (h_{1,j}, ..., h_{k,j})$ which consists of independent and uniform randomly selected hash functions $h_{i,j}$ from H where $1 \leqslant i \leqslant k, 1 \leqslant j \leqslant L$. The details of how choosing k and L is described in the papers.

In this paper, I used a package called E^{2}LSH which is developed by Alex Andoni. I get the code by asking him through the address that is located in LSH home page[8]. I slightly modified the code to print the memory usage (the variable is already there in the code). As the code uses euclidean as the distance metric, the comparison for this algorithm is only for the euclidean, not the cosine similarity.

The algorithm described by the author as follows :

Algorithm 9 *LSH initialization*

```
for $j \in L$ do
    $g_j \leftarrow (h_{1,j}, h_{2,j}, ..., h_{k,j})$ where $h_{1,j}, ..., h_{k,j}$ are chosen at random from the LSH family $H$
    $hashtable_j \leftarrow$ dataset points hashed using the function $g_j$
end for
```

Algorithm 10 *LSH searching*

```
for $j \in L$ do
    $P \leftarrow$ points from the bucket $g_j(q)$ in the $hashtable_j$
    for $p \in P$ do
        compute the distance from $q$ to it, and report the point if it is a correct answer
        (optional) Stop as soon as the number of reported points is more than $L'$. This require additional parameter $L'$
    end for
end for
```

Chapter 3

Evaluation and Analysis

The data set used for the experiment in this paper is a collection of text documents in the form of XML format. Each document will be a vector / point in the database, with the value taken from the text contained within each document. Each word that appears in the whole document will be a dimension in itself, and the occurrence(s) of that particular word in each of the document will be the value in that respective dimension. The resulting vector will be a very sparse vector. For the purpose of testing against the number of dimensions, I will first find the most frequent words from all documents. Then, I will iterate through all the points to remove the dimension that is not included as the most frequent words.

Here are the things that I measured as the comparison of performance between each methods:

- Time : It is measured as $metric = average(queryTime)$, where $average(n) = \frac{\Sigma_i n[i]}{|n|}$. The metric time included in the comparison is measured by the average time in seconds to do the searching. This does not include the overhead time in preprocessing the database and the query. There are some cases that the time measured is smaller than the resolution. For these cases, the time is set to 10^{-10} for the purpose of plotting the result in the graph.

- Memory : It is measured as $metric = preprocessing + average(queryMemory)$. The metric memory included in the comparison is measured for average memory used to store the data and memory used to do the searching.

- Error : It is measured as $metric = \frac{accumulation(error)}{dimension}$, where $accumulation(n) = \Sigma_i n[i]$ and the $error$ is measured differently between nearest neighbor and k-nearest neighbor:

- Nearest Neighbor : It is measured as $error = dist(r, q) - dist(rE, q)$, where r = point returned by a method, rE = point returned by exhaustive method, and q = query point. The exhaustive search is used as a benchmark, and then the error rate is measured as the distance between query and the returned point by each of the methods compared to the distance of query and the point returned by exhaustive search.
- k-Nearest Neighbor : It is measured as $error = dist(kr, q) - dist(krE, q)$, where kr = the farthest (k-th) point returned by a method, krE = the farthest (k-th) point returned by exhaustive method, and q = query point. The exhaustive search is used as a benchmark, and then the error rate is measured as the distance between query and the farthest point returned by each of the methods compared to the distance of query and the farthest point returned by exhaustive search. In this experiment, I used $k = 10$.

For RBC and LSH I used the code provided by the authors, and for the input, I generated an input file of dense vectors obtained by iterating the dimension in sparse vector representation. For this reason, it is not compared for the full dimension (but only included up to 1,000 dimensions) as the resulting file that contains the dense vectors is too large. As for the rest of the methods I mentioned, I implemented using Java as I am quite familiar with Java and Java has already provided the class to deal with sparse vectors. I represented the dimension as a mapping from word to a number, and each of the vector as a mapping from dimension number to value (total occurrences in a particular document).

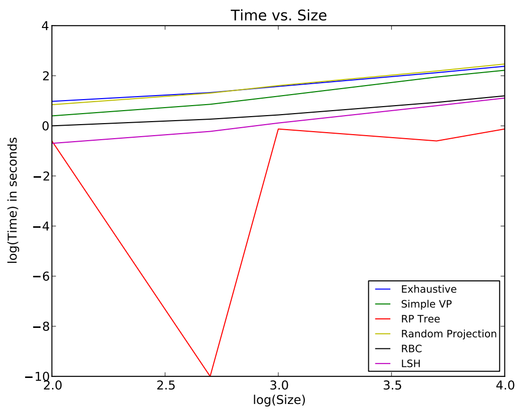
This section will be divided into two : testing against increasing size of dataset with a fixed number of dimension and testing against increasing number of dimension with a fixed size of dataset. Note that apply through the experiments : for Random Projection Matrix, the dimension of the problem will be projected to a subspace with 20% of its original number of dimension. As for RP Tree, the result is only for nearest neighbor.

3.1 Performance of the methods against increasing size of dataset with fixed number of dimension

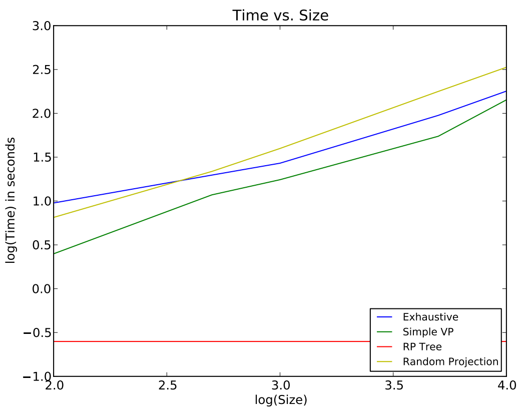
Here in this section I will provide the experiment result I have gathered so far for the increasing size of dataset. The fixed number of dimension used in this experiment is 1,000. Their performance is gathered against the dataset size of 100, 500, 1,000, 5,000, and 10,000.

3.1.1 Time

- Nearest Neighbor (See Figure 3.1)



(a) Euclidean

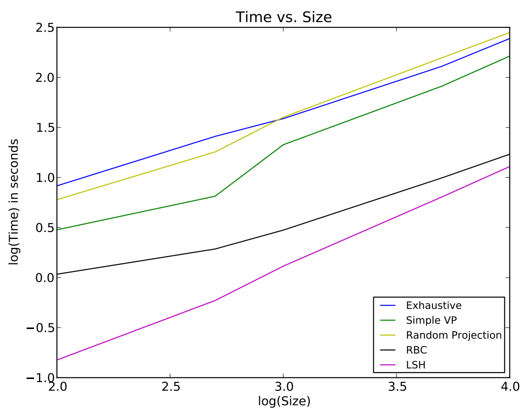


(b) Cosine Similarity

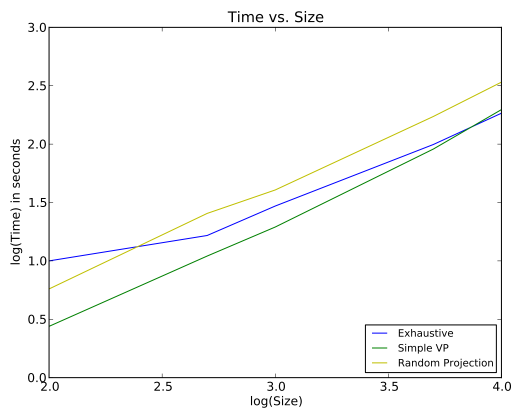
Figure 3.1: Nearest Neighbor measured Time against Size

Mostly the methods are behaving in almost the same way, that is, the time required to do the query is increasing along with the increasing size of dataset at almost the same rate. But there are two methods which behaves quite differently. RP Tree is the fastest in terms of time, and it seems not much affected by increasing size of the dataset (there is still an increase). Then, we have RBC which growing rate is slower than the other four methods.

- k-Nearest Neighbor (See Figure 3.2)
  Exhaustive, Vantage Point, Random Projection Matrix, and LSH have almost the same growth rate, that is, their growth in the time required to do the query against an increasing size of dataset are almost the



(a) Euclidean



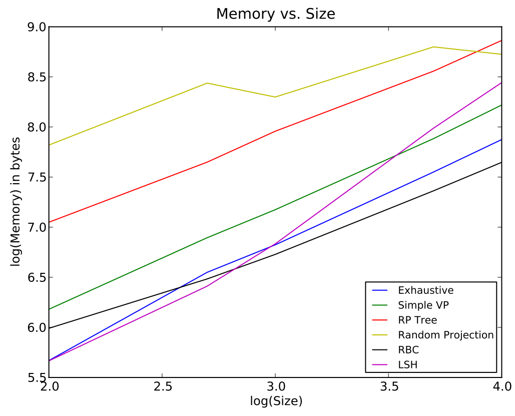
(b) Cosine Similarity

Figure 3.2: k-Nearest Neighbor measuring Time against Size

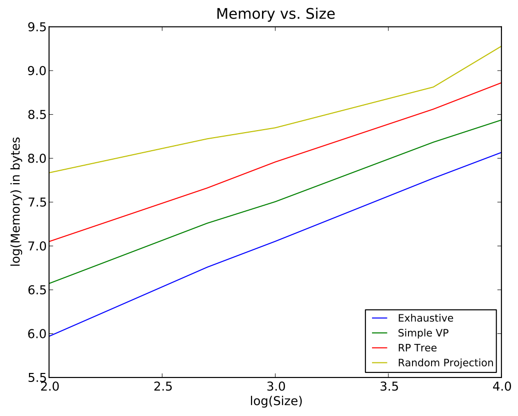
same. LSH is the fastest method when we are dealing with k-nearest neighbor problems. But, since its growing rate is faster than RBC, RBC possibly performs better if we are to work with larger dataset.

3.1.2 Space

- Nearest Neighbor (See Figure 3.3)



(a) Euclidean

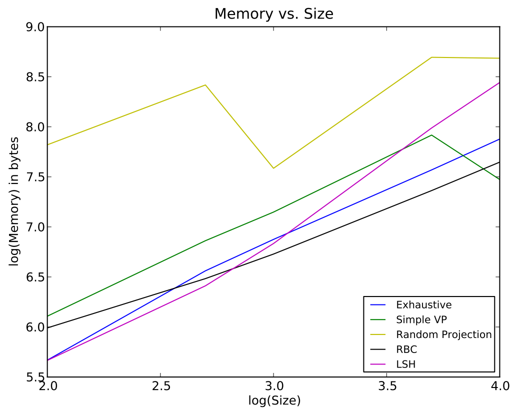


(b) Cosine Similarity

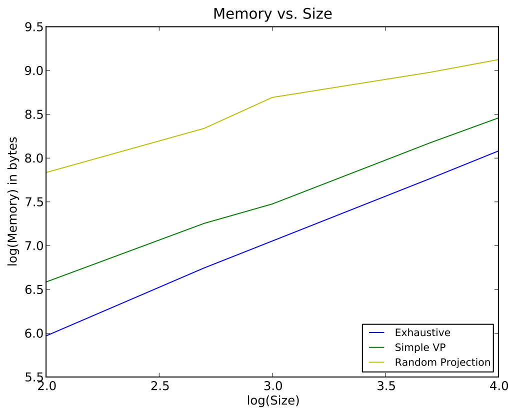
Figure 3.3: Nearest Neighbor measured Memory against Size

Although Random Projection Matrix memory usage also grows along with the increase of the size of dataset, but it is at a slower rate than the other methods. On the other hand, LSH is using the smallest memory initially, but it is growing at a faster rate than the other methods. As for the other four methods, the memory usage are growing at a similar rate.

- k-Nearest Neighbor (See Figure 3.4)



(a) Euclidean



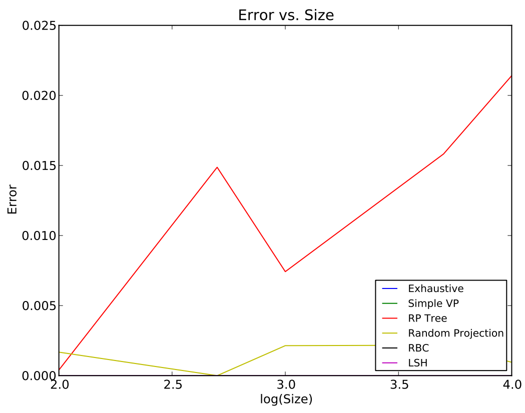
(b) Cosine Similarity

Figure 3.4: k-Nearest Neighbor measuring Memory against Size

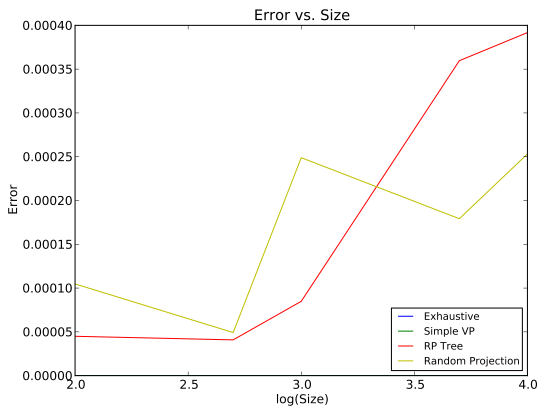
The result as we can see here is similar to that of nearest neighbor in that the growth rate of the memory usage of Random Projection Matrix is slower than the other methods, while LSH has faster growth rate than the other methods.

3.1.3 Error

- Nearest Neighbor (See Figure 3.5)
  Here, we do not see some of the methods because they yield no error. For exhaustive search, it yields no error because this method is used as the benchmark to measure error rate. Except for Random Projection Matrix and RP Tree, the other methods yield no error as well. For Random Projection Matrix, when we are working with the euclidean distance metric, the error rate is not much affected by the size of dataset. But it is growing with the size of dataset when we are working with the cosine similarity distance metric. On the other hand, the error rate for RP Tree seems to be increasing along with the size of the dataset both for the euclidean and the cosine similarity.



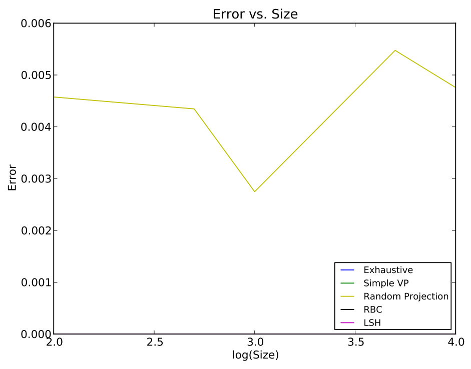
(a) Euclidean



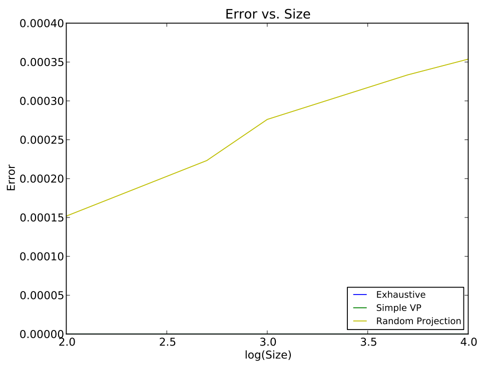
(b) Cosine Similarity

Figure 3.5: Nearest Neighbor measured Error rate against Size

- k-Nearest Neighbor (See Figure 3.6)



(a) Euclidean



(b) Cosine Similarity

Figure 3.6: k-Nearest Neighbor measuring Error rate against Size

For k-nearest neighbor, the experiment result for measuring error rate is similar to that of nearest neighbor as well. That is, we do not see all of the methods plotted on the graph. Here, we can only see the error of Random Projection Matrix because all the methods return the same result as exhaustive search. The trend is similar as well, in that the error rate measured for Random Projection Matrix is not much affected by the size of the dataset for euclidean, but is increasing for cosine similarity.

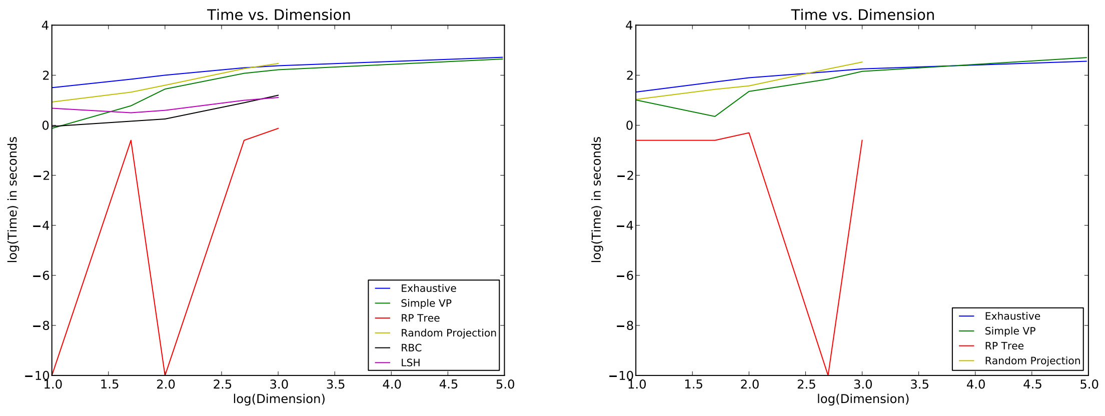
When dealing with nearest neighbor, RP Tree is the fastest. But Random Ball Cover and LSH are able to deal with k-nearest neighbor as well and performing well in terms of time. In terms of memory usage, LSH uses the least memory initially, but since it is growing at faster rate, it may not work well in much larger dataset. On the other hand Random Projection Matrix will perform better as it is growing at a slower rate. But, Random Projection Matrix and RP Tree should not be used if we want exact result, as these methods do not return the nearest point possible.

3.2 Performance of the methods against increasing number of dimension with fixed size of dataset

Here in this section I will provide the experiment result I have gathered so far for the increasing size of dataset. The fixed size of dataset used in this experiment is 10,000. Its performance is gathered against the dimension number of 10, 50, 100, 500, and 1,000.

3.2.1 Time

- Nearest Neighbor (See Figure 3.7)

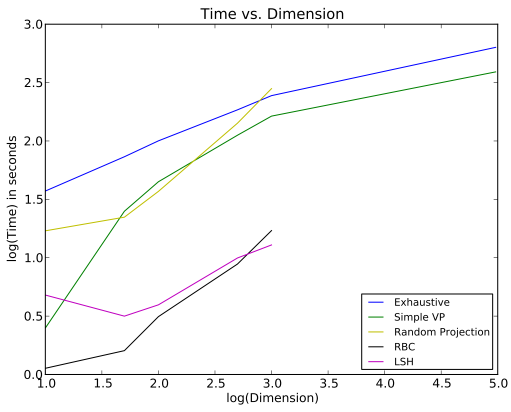


(a) Euclidean (b) Cosine Similarity

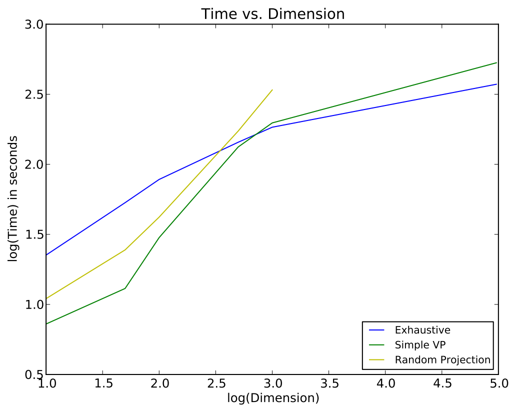
Figure 3.7: Nearest Neighbor measured Time against Dimension

Vantage Point and Random Projection Matrix work faster than exhaustive method when working on low dimensional problem. But with increasing number of dimensions, the performance gets worsened (for Vantage Point, until it is similar to the performance of exhaustive method). On the other hand, the required time to solve the problem for RP Tree and LSH approximately stays the same regardless the change in the number of dimensions. As for exhaustive and RBC, the time required to solve the problem is increasing along with the number of dimensions with similar growth rate.

- k-Nearest Neighbor (See Figure 3.8)



(a) Euclidean



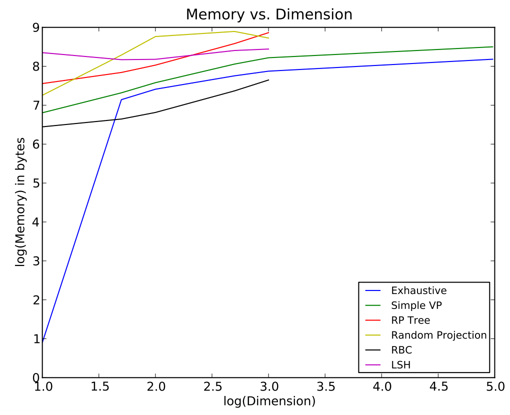
(b) Cosine Similarity

Figure 3.8: k-Nearest Neighbor measuring Time against Dimension

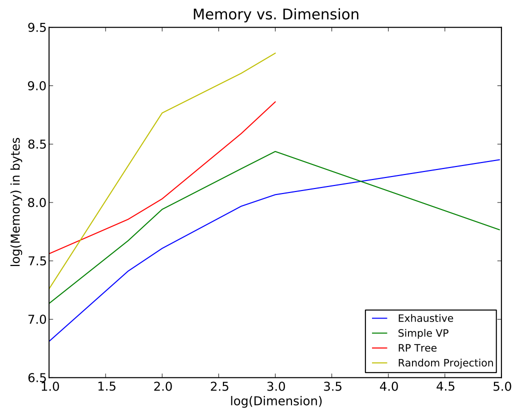
The trend I noticed for the result of k-nearest neighbor is similar to that of nearest neighbor. Vantage Point and Random Projection Matrix perform better in lower dimensional problem (better than exhaustive method), but the performance get worsened with higher number of dimensions. For LSH, the time required to solve the problem is not much affected by the number of dimensions (there is an increase in time required, but at a slower rate than other methods).

3.2.2 Space

- Nearest Neighbor (See Figure 3.9)
  Only LSH behaviour is noticable here, that is, the memory usage is not much affected by the number of dimensions. The rest of the methods behave almost the same in that they are consuming more memory as the number of dimensions is getting higher.



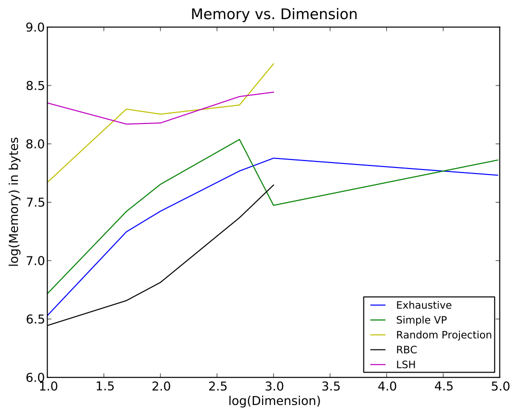
(a) Euclidean



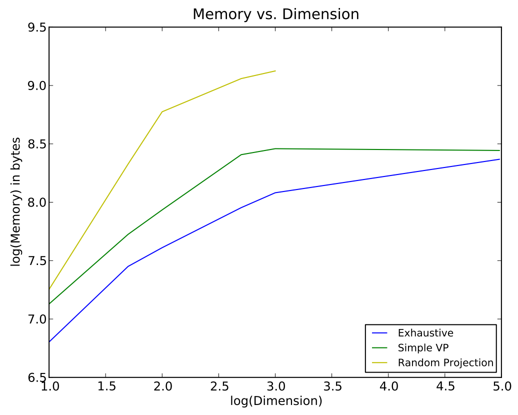
(b) Cosine Similarity

Figure 3.9: Nearest Neighbor measured Memory against Dimension

- k-Nearest Neighbor (See Figure 3.10)



(a) Euclidean



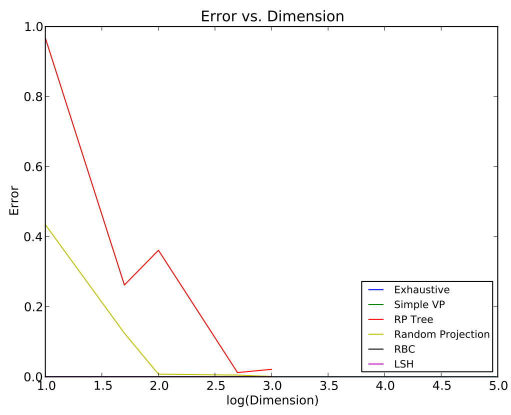
(b) Cosine Similarity

Figure 3.10: k-Nearest Neighbor measuring Memory against Dimension

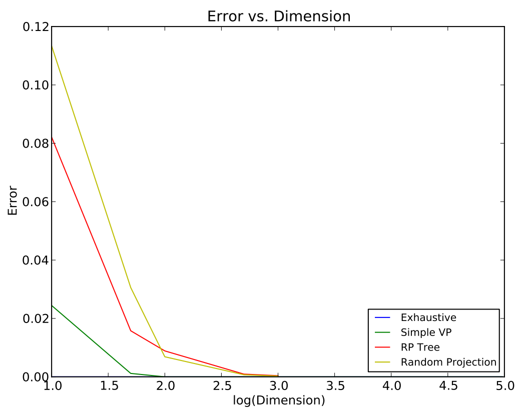
As with nearest neighbor problem, only LSH behaves differently than other methods in that the memory usage is not much changing while the rest of the methods grows in terms of memory usage for higher number of dimensions.

3.2.3 Error

- Nearest Neighbor (See Figure 3.11)



(a) Euclidean

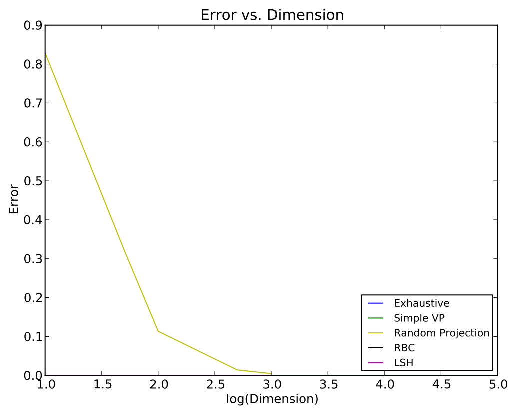


(b) Cosine Similarity

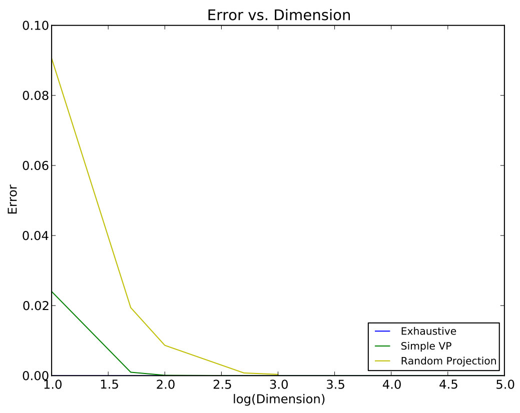
Figure 3.11: Nearest Neighbor measured Error rate against Dimension

When we measure the error rate for nearest neighbor, we do not see all of them in the figure because those methods give the same output as exhaustive method give which is used as the benchmark. But here, we see three methods that generate error: Random Projection Matrix, RP Tree, and Vantage Point. As for Vantage Point, error only occurred when we use cosine similarity as the distance metric in problems with low dimension (no error for cosine similarity in problems with high dimension). On the other hand, the error generated by Random Projection Matrix and RP Tree are going down with the increasing number of dimensions.

- k-Nearest Neighbor (See Figure 3.12)
  The behaviour of these methods are similar compared to that of the result in nearest neighbor. Vantage Point only yields error when it is working with problems in low dimension and use cosine similarity as its distance metric. As for Random Projection Matrix and RP Tree, the error rate are going down with the increasing number of dimensions, regardless the distance metric they used (it is the same for euclidean and cosine similarity).



(a) Euclidean



(b) Cosine Similarity

Figure 3.12: k-Nearest Neighbor measuring Error rate against Dimension

When dealing with nearest neighbor, RP Tree is the fastest. But Random Ball Cover and LSH are able to deal with k-nearest neighbor as well and performing well in terms of time. In terms of memory usage, LSH uses the most memory initially, but since it is growing at a slower rate, it will work better than other methods in higher number of dimensions (the behaviour is different compared to the result in 3.1). Random Projection Matrix and RP Tree should not be used if we want exact result, as these methods do not return the nearest point possible.

Chapter 4

Conclusion

There were a total of six algorithms compared in this paper : Exhaustive, Vantage Point, Random Ball Cover, Random Projection Matrix, RP Tree, and LSH.

- Exhaustive : Exhaustive method is the naïve ways of dealing with the problem of nearest neighbor. The performance (time and memory usage) is much affected by the number of dimensions and the size of dataset. This method always returns the nearest point possible to the query.
- Vantage Point : Vantage Point works well in low dimensional problems. But, as the number of dimensions increase, the performance is worsened until it is comparable to the exhaustive search. The reason is that in high dimension, the algorithm has to explore most of the spaces before it terminates. This method always returns the nearest point possible to the query when dealing with euclidean metric. But when we are dealing with low dimensional problem using the cosine similarity distance metric, it seems that this method is not returning the nearest point possible to the query.
- Random Ball Cover : Works well in both low dimension and high dimension. This method always returns the nearest point possible to the query.
- Random Projection Matrix : There is a trend noticable when working with this method in high dimensional problem that the performance is worse than exhaustive search because the projection resulted in a dense vector, although originally it is a sparse vector problem. It is because working in the dense vector is less advantageous than that in sparse

vector as we do not have to access every dimension of the vector. In addition, Random Projection Matrix is that the approximation to the nearest point is better in high dimension as the performance from the theory only depends on the number of points instead of dimensions. This method is not guaranteed to return the nearest point possible to the query.

- RP Tree : This method is the fastest method in measured average query time compared in this paper. This method works better in high dimension than in low dimension as the error rate is going down along with increasing number of dimension. However, this method is not guaranteed to return the nearest point possible to the query.

- LSH : In theory, this method is approximate in that we are not guaranteed to get the nearest point possible to the query. But in this experiment, I always get the nearest point possible to the query (the same as Exhaustive). LSH has different noticable trait as well. It shows no significant change in performance whether it is working on low dimension or high dimension. But, the memory usage is considerably increased along with the increase in number of points in the database.

Bibliography

[1] Weisstein, Eric W n.d, *"Metric." From MathWorld– A Wolfram Web Resource*, accessed 4 november 2011, <http://mathworld.wolfram.com/metric.html>.

[2] Cayton, Lawrence. Accelerating nearest neighbor search on manycore systems. 2011.

[3] Cayton, Lawrence n.d, *Lawrence Cayton : code*, accessed 4 november 2011, <http://people.kyb.tuebingen.mpg.de/lcayton/code.html>.

[4] S. Dasgupta and A. Gupta. An elementary proof of a theorem of Johnson and Lindenstrauss. *Random Structures and Algorithms*, 22(1):60-65, 2003.

[5] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. *Fortieth ACM Symposium on Theory of Computing (STOC)*, 2008.

[6] Andoni. Alexandr and Indyk, p. 2008. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions, Communications of the ACM, pp.117-122.

[7] Indyk, P. and Motwani, R. 1998. Approximate nearest neighbor: Towards removing the curse of dimensionality. In *Proceedings of the Symposium on Theory of Computing.*

[8] Andoni, Alexandr n.d, *LSH Algorithm and Implementation (E2LSH)*, accessed 4 november 2011, <http://www.mit.edu/∼andoni/lsh/>.

[9] Lifshits, Yury 2007. *The Homepage of Nearest Neighbors and Similarity Search*, accessed 4 november 2011, <http://simsearch.yury.name/tutorial.html>.

[10] Orchard, M.T. A fast nearest-neighbor search algorithm, ICASSP'91. 1991.

[11] E. Vidal, An algorithm for finding nearest neighbours in(approximately) constant average time complexity, Inform.Process. Lett. 4 (1986) 145-157.