A Simple Algorithm for Nearest Neighbor Search in High Dimensions

Sameer A. Nene and Shree K. Nayar

Department of Computer Science
Columbia University
New York, N.Y. 10027

October, 1995

Technical Report No. CUCS-030-95

Abstract

The problem of finding the closest point in high-dimensional spaces is common in pattern recognition. Unfortunately, the complexity of most existing search algorithms, such as k-d tree and R-tree, grows exponentially with dimension, making them impractical for dimensionality above 15. In nearly all applications, the closest point is of interest only if it lies within a user specified distance ϵ. We present a simple and practical algorithm to efficiently search for the nearest neighbor within Euclidean distance ϵ. The use of projection search combined with a novel data structure dramatically improves performance in high dimensions. A complexity analysis is presented which helps to automatically determine ϵ in structured problems. A comprehensive set of benchmarks clearly shows the superiority of the proposed algorithm for a variety of structured and unstructured search problems. Object recognition and motion estimation in MPEG coding are demonstrated as example applications. A pseudo code implementation of the proposed algorithm is included. The simplicity of the algorithm makes it possible to construct an inexpensive hardware search engine which can be 100 times faster than its software equivalent.

Index terms: Pattern classification, nearest neighbor, searching by slicing, benchmarks, object recognition, image coding, visual correspondence, hardware architecture.

1 Introduction

Searching for nearest neighbors continues to prove itself as an important problem in many fields of science and engineering. The nearest neighbor problem in multiple dimensions is stated as follows: given a set of n points and a novel query point $\boldsymbol{Q}$ in a d-dimensional space, "Find a point in the set such that its distance from $\boldsymbol{Q}$ is lesser than, or equal to, the distance of $\boldsymbol{Q}$ from any other point in the set" [Knuth-1973]. A variety of search algorithms have been advanced since Knuth first stated this (post-office) problem. Why then, do we need a new algorithm? The answer is that existing techniques perform very poorly in high dimensional spaces. The complexity of most techniques grows exponentially with the dimensionality, d. By high dimensional, we mean when, say $d > 25$. Such high dimensionality occurs commonly in applications that use eigenspace based appearance matching, such as real-time object recognition [Murase and Nayar-1995], visual positioning, tracking and inspection [Nayar *et al.*-1994], and feature detection [Nayar *et al.*-1995]. Moreover, these techniques require that nearest neighbor search be performed using the Euclidean distance (or L_2) norm. This can be a hard problem, especially when dimensionality is high. High dimensionality is also observed in visual correspondence problems such as motion estimation in MPEG coding ($d > 256$) [Netravali-1995], disparity estimation in binocular stereo (d=25-81), and optical flow computation in structure from motion (also d=25-81).

In this paper, we propose a simple algorithm to efficiently search for the nearest neighbor within distance ϵ in high dimensions. We shall see that the complexity of the proposed algorithm does not grow exponentially with d and further, for small ϵ, the complexity is almost constant for any d. Our algorithm is successful because it does not tackle the nearest neighbor problem as originally stated; it only finds points within distance ϵ from the novel point. This property is sufficient in most pattern recognition problems (and for the problems stated above), because a "match" is declared with high confidence only when a novel point is sufficiently close to a training point. Occasionally, it is not possible to assume that ϵ is known, so we suggest a method to automatically choose ϵ. We now briefly outline the proposed algorithm.

Our algorithm is based on the projection search paradigm first used by Friedman [Friedman *et al.*-1975]. His simple technique works as follows. In the preprocessing step, d dimensional training points are ordered in d different ways by individually sorting each of their

coordinates. Each of the d sorted coordinate arrays can be thought of as a 1-D axis with the entire d dimensional space collapsed (or projected) onto it. Given a novel point $\boldsymbol{Q}$, the nearest neighbor is found as follows. A small offset ϵ is subtracted from and added to each of $\boldsymbol{Q}$'s coordinates to obtain two values. Two binary searches are performed on each of the sorted arrays to locate the positions of both the values. An axis with the minimum number of points in between the positions is chosen. Finally, points in between the positions on the chosen axis are exhaustively searched to obtain the closest point. The complexity of this technique is roughly $O(nd\epsilon)$ and is clearly inefficient.

Yunck devised a data structure to limit the exhaustive search to within a hypercube of side 2ϵ in order to reduce the number floating point distance calculations [Yunck-1976]. With this data structure, he was able to find points within this hypercube using only integer operations and floating point comparisons. However, the *total* number of operations required (integer and floating point) to find points within the hypercube are similar to that of Friedman's algorithm. Due to this and the fact that most modern CPUs do not significantly penalize floating point, the improvement is only slight. However, the simplicity of the projection technique is still attractive and we exploit this in our algorithm. We propose a data structure that significantly reduces the number of operations required to locate points within the hypercube. Moreover, this data structure facilitates a very simple hardware implementation which can result in a further increase in performance by two orders of magnitude.

2 Previous Work

Search algorithms can be divided into the following broad categories: (a) Exhaustive search, (b) hashing and indexing, (c) static space partitioning, (d) dynamic space partitioning, and (e) randomized algorithms. The algorithm described in this paper falls in category (d). Exhaustive search, as the term implies, involves computing the distance of the novel point from each and every point in the set and finding the point with the minimum distance. This approach is clearly inefficient and its complexity is $O(nd)$.

Hashing and indexing are the fastest search techniques and run in constant time. However, the space required to store an index table increases exponentially with d. Hence, hybrid schemes of hashing from a high dimensional space to a low (1 or 2) dimensional space and then indexing in this low dimensional space have been proposed. Such a dimensionality re-

duction is called geometric hashing [Wolfson-1990] [Califano and Mohan-1991]. The problem is that, with increasing dimensionality, it becomes difficult to construct a hash function that distributes data uniformly across the entire hash table (index). An added drawback arises from the fact that hashing inherently partitions space into bins. If two points in adjacent bins are closer to each other than a third point within the same bin. A search algorithm that uses a hash table, or an index, will not correctly find the point in the adjacent bin. Hence, hashing and indexing are only really effective when the novel point is exactly equal to one of the database points.

Space partitioning techniques have led to a few elegant solutions to multi-dimensional search problems. A method of particular theoretical significance divides the search space into Voronoi polygons. A Voronoi polygon is a geometrical construct obtained by intersecting perpendicular bisectors of adjacent points. In a 2-D search space, Voronoi polygons allow the nearest neighbor to be found in $O(\log_2 n)$ operations, where, n is the number of points in the database. Unfortunately, the cost of constructing and storing Voronoi diagrams grows exponentially with the number of dimensions. Details can be found in [Aurenhammer-1991], [Edelsbrunner-1987], [Klee-1980], and [Preparata and Shamos-1985]. Another algorithm of interest is the 1-D binary search generalized to d dimensions [Dobkin and Lipton-1976]. This runs in $O(\log_2 n)$ time but requires storage $O(n^4)$, which makes it impractical for $n > 100$.

Perhaps the most widely used algorithm for searching in multiple dimensions is a static space partitioning technique based on a k dimensional binary search tree, called the k-d tree [Bentley-1975] [Bentley-1979]. The k-d tree is a data structure which partitions space using hyperplanes placed perpendicular to the coordinate axes. The partitions are arranged hierarchically to form a tree. In its simplest form, a k-d tree is constructed as follows. A point in the database is chosen to be the root node. Points lying on one side of a hyperplane passing through the root node are added to the left child and the points on the other side are added to the right child. This process is applied recursively on the left and right children until a small number of points remain. The resulting tree of hierarchically arranged hyperplanes induces a partition of space into hyper-rectangular regions, termed buckets, each containing a small number of points. The k-d tree can be used to search for the nearest neighbor as follows. The k coordinates of a novel point are used to descend the tree to find the bucket which contains it. An exhaustive search is performed to determine the closest point within that bucket. The size of a "query" hypersphere is set to the distance of this closest point.

Information stored at the parent nodes is used to determine if this hypersphere intersects with any other buckets. If it does, then that bucket is exhaustively searched and the size of the hypersphere is revised if necessary. For fixed d, and under certain assumptions about the underlying data, the k-d tree requires $O(n\log_2 n)$ operations to construct and $O(\log_2 n)$ operations to search [Bentley and Weide-1980] [Bentley and Friedman-1979] [Friedman *et al.*-1977].

k-d trees are extremely versatile and efficient to use in low dimensions. However, the performance degrades exponentially with increasing dimensionality. This is because, in high dimensions, the query hypersphere tends to intersect many adjacent buckets, leading to a dramatic increase in the number of points examined. k-d trees are dynamic data structures which means that data can be added or deleted at a small cost. The impact of adding or deleting data on the search performance is however quite unpredictable and is related to the amount of imbalance the new data causes in the tree. High imbalance generally means slower searches. A number of improvements to the basic algorithm have been suggested. Friedman recommends that the partitioning hyperplane be chosen such that it passes through the median point and is placed perpendicular to the coordinate axis along whose direction the spread of the points is maximum [Friedman *et al.*-1977]. Sproull suggests using a truncated distance computation to increase efficiency in high dimensions [Sproull-1991]. Variants of the k-d tree have been used to address specific search problems [Arya-1995] [Robinson-1981].

An R-tree is also a space partitioning structure, but unlike k-d trees, the partitioning element is not a hyperplane but a hyper-rectangular region [Guttman-1984]. This hierarchical rectangular structure is useful in applications such as searching by image content [Petrakis and Faloutsos-1994] where one needs to locate the closest manifold (or cluster) to a novel manifold (or cluster). An R-tree also addresses some of the problems involved in implementing k-d trees in large disk based databases. The R-tree is also a dynamic data structure, but unlike the k-d tree, the search performance is not affected by addition or deletion of data. A number of variants of R-Trees improve on the basic technique, such as packed R-trees [Roussopoulos and Leifker-1985], R+-trees [Sellis *et al.*-1987] and R*-trees [Beckmann *et al.*-1990]. Although R-trees are useful in implementing sophisticated queries and managing large databases, the performance of nearest neighbor point searches in high dimensions is very similar to that of k-d trees; complexity grows exponentially with d.

Other static space partitioning techniques have been proposed such as branch and bound

[Fukunaga and Narendra-1975], quad-trees [Gargantini-1982], vp-trees [Yianilos-1993], and hB-trees [Lomet and Salzberg-1990], none of which significantly improve performance for high dimensions. Clarkson describes a randomized algorithm which finds the closest point in d dimensional space in $O(\log_2 n)$ operations using a RPO (randomized post office) tree [Clarkson-1988]. However, the time taken to construct the RPO tree is $O(n^{\lceil d/2 \rceil (1+\epsilon)})$ and the space required to store it is also $O(n^{\lceil d/2 \rceil (1+\epsilon)})$. This makes it impractical when the number of points n is large or or if $d > 3$.

3 The Algorithm

3.1 Searching by Slicing

We illustrate the proposed high dimensional search algorithm using a simple example in 3-D space, shown in Figure 1. We call the set of points in which we wish to search for the closest point as the *point set*. Then, our goal is to find the point in the point set that is closest, in the Euclidean sense, to a novel query point $\boldsymbol{Q}(x, y, z)$ and within a distance ϵ. Our approach is to first find all the points that lie inside a cube (see Figure 1) of side 2ϵ centered at $\boldsymbol{Q}$. Since ϵ is typically small, the number of points inside the cube is also small. The closest point can then be found by performing an exhaustive search through these points. If there are no points inside the cube, we know that there are no points within ϵ.

The points within the cube can be found as follows. First, we find the points that are sandwiched between a pair of parallel planes X_1 and X_2 (see Figure 1) and add them to a list, which we call the *candidate list*. The planes are perpendicular to the first axis of the coordinate frame and are located on either side of point $\boldsymbol{Q}$ at a distance of ϵ. Next, we trim the candidate list by discarding points that are *not* also sandwiched between the parallel pair of planes Y_1 and Y_2, that are perpendicular to X_1 and X_2, again located on either side of $\boldsymbol{Q}$ at a distance ϵ. This procedure is repeated for planes Z_1 and Z_2, at the end of which, the candidate list contains only points within the cube of size 2ϵ centered on $\boldsymbol{Q}$.

Once we have this trimmed candidate list, the closest point is found by performing an exhaustive search on the trimmed list. Since the number of points in the final trimmed list is typically small, the cost of the exhaustive search is negligible. The major computational cost in our technique is therefore in constructing *and* trimming the candidate list.

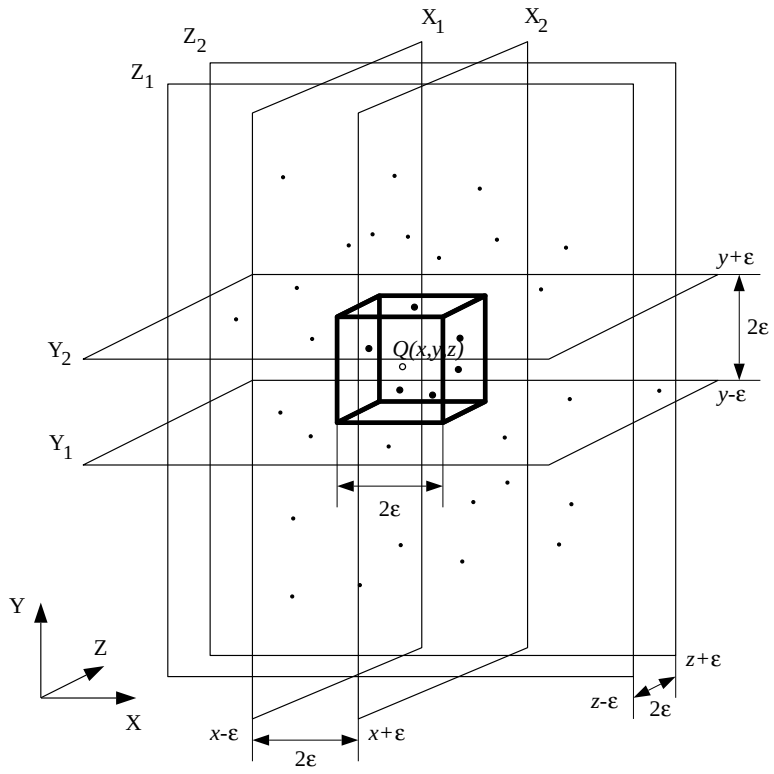


Figure 1: The proposed algorithm efficiently finds points inside a cube of size 2ϵ around the novel query point $\boldsymbol{Q}$. The closest point is then found by performing an exhaustive search within the cube using the Euclidean distance metric.

3.2 Data Structure

Candidate list construction and trimming can done in a variety of ways. Here, we propose a method that uses a simple pre-constructed data structure along with 1-D binary searches [Aho *et al.*-1974] to efficiently find points sandwiched between a pair of parallel hyperplanes. The data structure is constructed from the raw point set and is depicted in Figure 2. It is assumed that the point set is static and hence, for a given point set, the data structure needs to be constructed only once. The point set is stored as a collection of d 1-D arrays, where the j^{th} array contains the j^{th} coordinate of the points. Thus, in the point set, coordinates of a point lie along the same row. This is illustrated by the dotted lines in Figure 2. Now suppose that novel point $\boldsymbol{Q}$ has coordinates $Q_1, Q_2, \cdots, Q_d$. Recall that in order to construct the candidate list, we need to find points in the point set that lie between a pair of parallel hyperplanes separated by a distance 2ϵ, perpendicular to the first coordinate axis, and centered at Q_1; that is, we need to locate points whose first coordinate lies between the

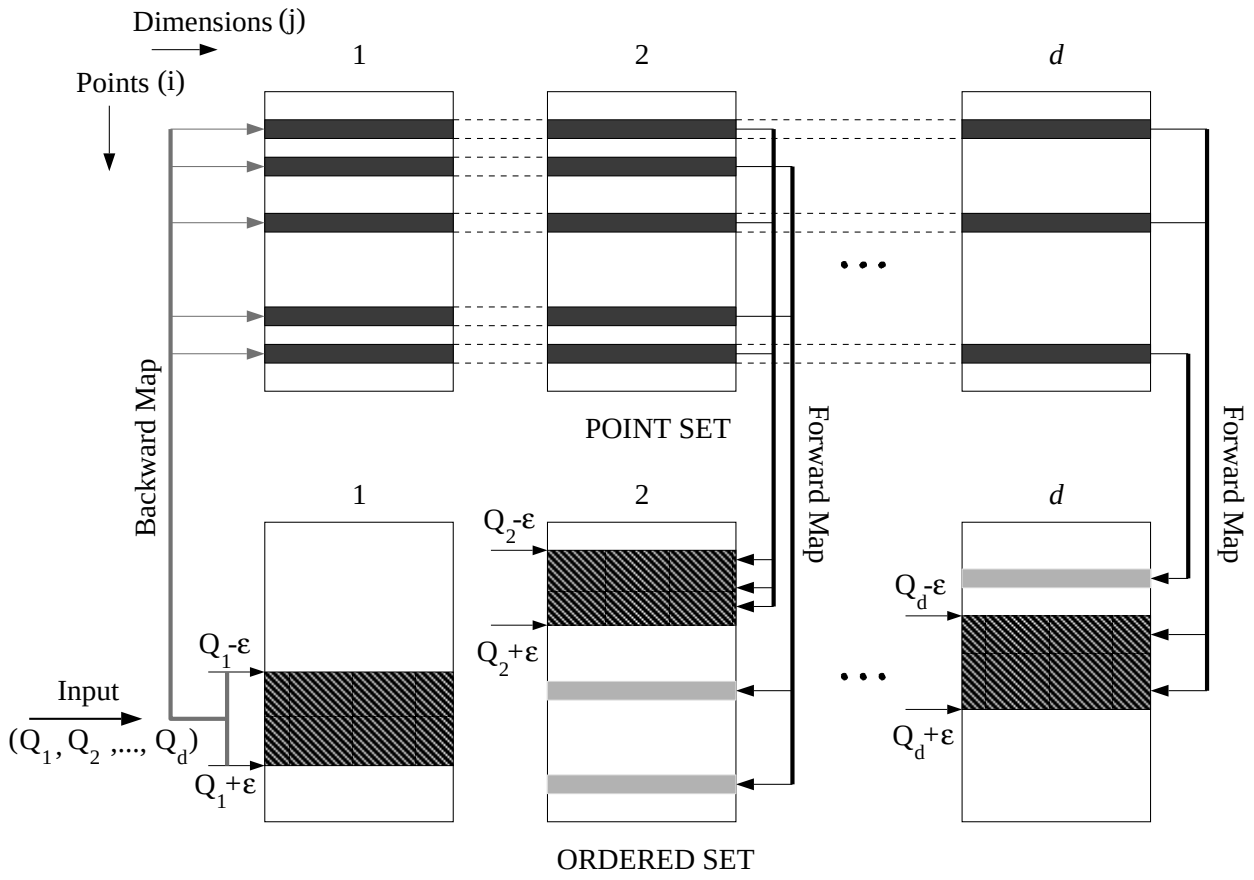


Figure 2: Data structures used for constructing and trimming the candidate list. The point set corresponds to the raw list of data points, while in the ordered set each coordinate is sorted. The forward and backward maps enable efficient correspondence between the point and ordered sets.

limits $Q_1 - \epsilon$ and $Q_1 + \epsilon$. This can be done with the help of two binary searches, one for each limit, if the coordinate array were sorted beforehand.

To this end, we sort each of the d coordinate arrays in the point set independently to obtain the *ordered set*. Unfortunately, sorting raw coordinates does not leave us with any information regarding which points in the arrays of the ordered set correspond to any given point in the point set, and vice versa. For this purpose, we maintain two maps. The *backward map* maps a coordinate in the ordered set to the corresponding coordinate in the point set and, conversely, the *forward map* maps a point in the point set to a point in the ordered set. Notice that the maps are simple integer arrays; if $P[d][n]$ is the point set, $O[d][n]$ is the ordered set, $F[d][n]$ and $B[d][n]$ are the forward and backward maps, respectively, then $O[i][F[i][j]] = P[i][j]$ and $P[i][B[i][j]] = O[i][j]$.

Observe that binary search helps us to efficiently find (in time $O(\log_2 n)$) the coordinates in the ordered set that lie between the parallel hyperplanes positioned at $Q_1 - \epsilon$ and $Q_1 + \epsilon$. Using the backward map, we find the corresponding points in the point set (shown as dark shaded areas) and add the appropriate points to the candidate list. With this, the construction of the candidate list is complete. Next, we trim the candidate list by iterating on through $k = 2, 3, \cdots, d$, as follows. In the iteration k, we check every point in the candidate list, by using the forward map, to see if its k^{th} coordinate lies within the limits $Q_k - \epsilon$ and $Q_k + \epsilon$. Each of these limits are also obtained by binary search. Points with k^{th} coordinates that lie outside this range (shown in light grey) are discarded from the list.

At the end of the final iteration, points remaining on the candidate list are the ones which lie inside a hypercube of side 2ϵ centered at $\boldsymbol{Q}$. In our discussion, we proposed constructing the candidate list using the first dimension, and then performing list trimming using dimensions $2, 3, \cdots, d$, in that order. We wish to emphasize that these operations can be done in any order and still yield the desired result. In the next section, we shall see that it is possible to determine an optimal ordering such that the cost of constructing and trimming the list is minimized.

It is important to note that the only operations used in trimming the list are integer comparisons and memory lookups. Moreover, by using the proposed data structure, we have limited the use of floating point operations to just the binary searches needed to find the row indices corresponding to the hyperplanes. This feature is critical to the efficiency of the proposed algorithm, when compared with competing ones. It not only facilitates a simple software implementation, but also permits the implementation of a hardware search engine.

As previously stated, the algorithm needs to be supplied with an "appropriate" ϵ prior to search. This is possible for a large class of problems (in pattern recognition, for instance) where a match can be declared only if the novel point $\boldsymbol{Q}$ is sufficiently close to a database point. It is reasonable to assume that ϵ is given a priori, however, the choice of ϵ can prove problematic if this is not the case. One solution is to set ϵ large, but this might seriously impact performance. On the other hand, a small ϵ could result in the hypercube being empty. How do we determine an optimal ϵ for a given problem? How exactly does ϵ affect the performance of the algorithm? We seek answers to these questions in the following section.

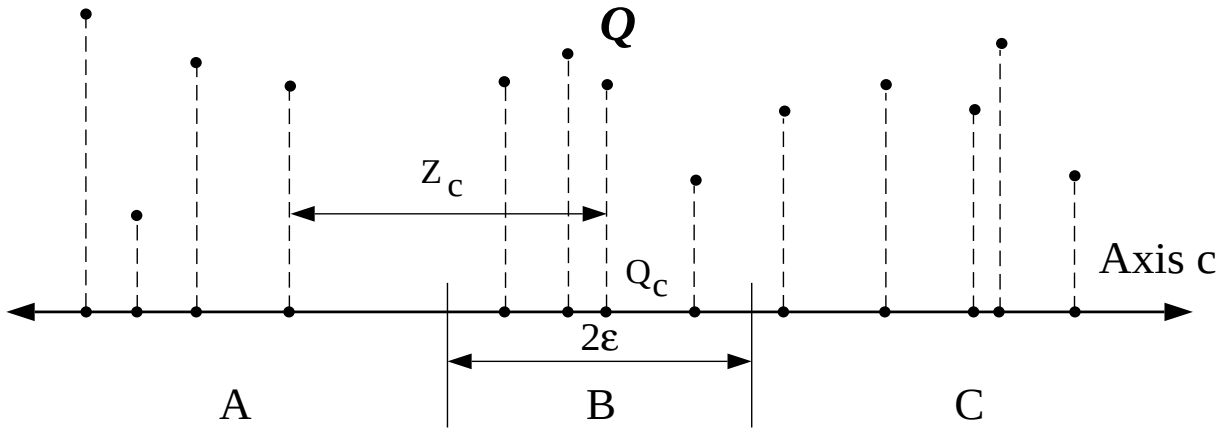


Figure 3: The projection of the point set and the novel point onto one of the dimensions of the search space. The number of points inside bin B is given by the binomial distribution.

4 Complexity

The major computational cost is in the process of candidate list construction and trimming. The number of points initially added to the candidate list depends not only on ϵ, but also on the distribution of data in the point set and the location of the novel point $\boldsymbol{Q}$. Hence, to facilitate analysis, we structure the problem by assuming widely used distributions for the point set. The following notation is used. Random variables are denoted by uppercase letters, for instance, Q. Vectors are in bold, such as, $\boldsymbol{q}$. Suffixes are used to denote individual elements of vectors, for instance, Q_k is the k^{th} element of vector $\boldsymbol{Q}$. Probability density is written as $P\{\boldsymbol{Q} = \boldsymbol{q}\}$ if $\boldsymbol{Q}$ is discrete, and as $f_{\boldsymbol{Q}}(\boldsymbol{q})$ if $\boldsymbol{Q}$ is continuous.

Figure 3 shows the novel point $\boldsymbol{Q}$ and a set of points in 2-D space drawn from a known distribution. Recall that the candidate list is initialized with points sandwiched between a hyperplane pair in the first dimension, or more generally, in the c^{th} dimension. This corresponds to the points inside bin B in Figure 3, where the entire point set and $\boldsymbol{Q}$ are projected to the c^{th} coordinate axis. The boundaries of bin B are where the hyperplanes intersect the axis c, at $Q_c - \epsilon$ and $Q_c + \epsilon$. Let M_c be the number of points in bin B. In order to determine the average number of points added to the candidate list, we must compute $E[M_c]$. Define Z_c to be the distance between Q_c and *any* point on the candidate list. The distribution of Z_c may be calculated from the the distribution of the point set. Define P_c to be the probability that any projected point in the point set is within distance ϵ from Q_c;

that is,

$$P_c = P\{-\epsilon \leq Z_c \leq \epsilon \mid Q_c\}$$

It is now possible to write an expression for the density of M_c in terms of P_c. Irrespective of the distribution of the points, M_c is binomially distributed:

$$P\{M_c = k \mid Q_c\} = P_c^k (1 - P_c)^{n-k} \begin{pmatrix} n \\ k \end{pmatrix}$$

From the above expression, the average number of points in bin B, $E[M_c \mid Q_c]$, is easily determined to be

$$\begin{aligned} E[M_c \mid Q_c] &= \sum_{k=0}^{n} kP\{M_c = k \mid Q_c\} \\ &= nP_c \end{aligned} \tag{1}$$

Note that $E[M_c \mid Q_c]$ is itself a random variable that depends on c and the location of $\boldsymbol{Q}$. If the distribution of $\boldsymbol{Q}$ is known, the expected number of points in the bin can be computed as $E[M_c] = E[E[M_c \mid Q_c]]$. Since we perform one lookup in the backward map for every point between a hyperplane pair, and this is the main computational effort, equation (1) directly estimates the cost of candidate list construction.

Next, we derive an expression for the total number of points remaining on the candidate list as we trim through the dimensions in the sequence $c_1, c_2, \ldots, c_d$. Recall that in the iteration k, we perform a forward map lookup for every point in the candidate list and see if it lies between the ${c_k}^{th}$ hyperplane pair. How many points on the candidate list lie between this hyperplane pair? Once again, equation (1) can be used, this time replacing n with the number of points on the candidate list rather than the entire point set. We assume that the point set is independently distributed. Hence, if N_k is the total number of points on the candidate list *before* the iteration k,

$$\begin{aligned} N_k &= P_{c_k} N_{k-1}, \quad N_0 = n \\ &= n \prod_{i=1}^{k} P_{c_i} \end{aligned} \tag{2}$$

Define N to be the total cost of constructing and trimming the candidate list. For each trim, we need to perform one forward map lookup and two integer comparisons. Hence, if

we assign one cost unit to each of these operations, an expression for N can be written with the aid of equation (2) as

$$\begin{aligned} N &= N_1 + 3N_1 + 3N_2 + \cdots + 3N_{d-1} \\ &= N_1 + 3\sum_{k=1}^{d-1} N_k \\ &= n\left(P_{c_1} + 3\sum_{k=1}^{d-1}\prod_{i=1}^{k} P_{c_i}\right) \end{aligned} \tag{3}$$

which, on the average is

$$E[N \mid \boldsymbol{Q}] = nE\left[P_{c_1} + \sum_{k=1}^{d-1}\prod_{i=1}^{k} P_{c_i}\right] \tag{4}$$

Equation (4) suggests that if the distributions $f_{\boldsymbol{Q}}(\boldsymbol{q})$ and $f_{\boldsymbol{Z}}(\boldsymbol{z})$ are known, we can compute the average cost $E[N] = E[E[N \mid \boldsymbol{Q}]]$ in terms of ϵ. In the next section, we shall examine two cases of particular interest: (a) $\boldsymbol{Z}$ is uniformly distributed, and (b) $\boldsymbol{Z}$ is normally distributed. Note that we have left out the cost of exhaustive search on points within the final hypercube. This is very small and can be neglected in most cases when $n \gg d$. If it needs to be considered, it is simply $N_d d$ and can be added to equation (4).

We end this section by making an observation. We had mentioned earlier that it is of advantage to examine the dimensions in a specific order. What is this order? By expanding the summation and product and by factoring terms, equation (3) can be rewritten as

$$N = n\left(P_{c_1} + 3(P_{c_1}(1 + P_{c_2}(1 + P_{c_3}(1 + \cdots))))\right)$$

It is immediate that the value of N is minimum when $P_{c_1} < P_{c_2} < \cdots < P_{c_{d-1}}$. In other words, $c_1, c_2, \cdots, c_d$ should be chosen such that the numbers of sandwiched points between hyperplane pairs are in ascending order. This can be easily ensured by simply sorting the numbers of sandwiched points. Note that there are only d such numbers, and the cost of this sorting is $O(d \log_2 d)$ by heapsort [Aho *et al.*-1974]. Clearly, this cost is negligible in any problem of reasonable dimensionality.

4.1 Uniformly Distributed Point Set

We now look at the specific case of a point set that is uniformly distributed. If $\boldsymbol{X}$ is a point in the point set, we assume an independent and uniform distribution with extent l on each

of it's coordinates as

$$f_{X_c}(x) = \begin{cases} 1/l & \text{if } -l/2 \leq x \leq l/2 \\ 0 & \text{otherwise} \end{cases}, \forall c \tag{5}$$

Using equation (5) and the fact that $Z_c = X_c - Q_c$, an expression for the density of Z_c can be written as

$$f_{Z_c|Q_c}(z) = \begin{cases} 1/l & \text{if } -l/2 - Q_c \leq z \leq l/2 - Q_c \\ 0 & \text{otherwise} \end{cases}, \forall c \tag{6}$$

P_c can now be written as

$$\begin{aligned} P_c &= P\{-\epsilon \leq Z_c \leq \epsilon \mid Q_c\} = \int_{-\epsilon}^{\epsilon} f_{Z_c|Q_c}(z)dz \\ &\leq \int_{-\epsilon}^{\epsilon} \frac{1}{l} dz \\ &\leq \frac{2\epsilon}{l} \end{aligned} \tag{7}$$

Substituting equation (7) in equation (4) and considering the upper bound (worst case), we get

$$\begin{aligned} E[N] &= n\left(\frac{2\epsilon}{l} + 3\left(\frac{2\epsilon}{l} + \left(\frac{2\epsilon}{l}\right)^2 + \cdots + \left(\frac{2\epsilon}{l}\right)^{d-1}\right)\right) \\ &= n\left(\frac{2\epsilon}{l} + 3\left(\frac{1 - \left(\frac{2\epsilon}{l}\right)^d}{1 - \frac{2\epsilon}{l}} - 1\right)\right) \end{aligned} \tag{8}$$

Putting $y = 2\epsilon/l$ in the above equation gives

$$E[N] = n\left(\frac{4y - y^2 - 3y^d}{1 - y}\right)$$

If ϵ/l is small, $y \ll 1$ and the above expression can be simplified to

$$E[N] \approx \frac{8n\epsilon}{l}$$

Hence, for small ϵ, we see that the cost is independent of d. In Figure 4, equation (8) is plotted against ϵ for different d (Figure 4(a)) and different n (Figure 4(b)) for $l = 1$. Observe that as long as $\epsilon < .25$, the cost varies little with d and is linearly proportional to n. Keeping ϵ small is crucial to the performance of the algorithm. As we shall see later, ϵ can be kept small for most problems. Hence, even though the cost grows linearly with n, the constant is small enough that in many real problems it is better to pay this price, rather than an exponential dependence on d.

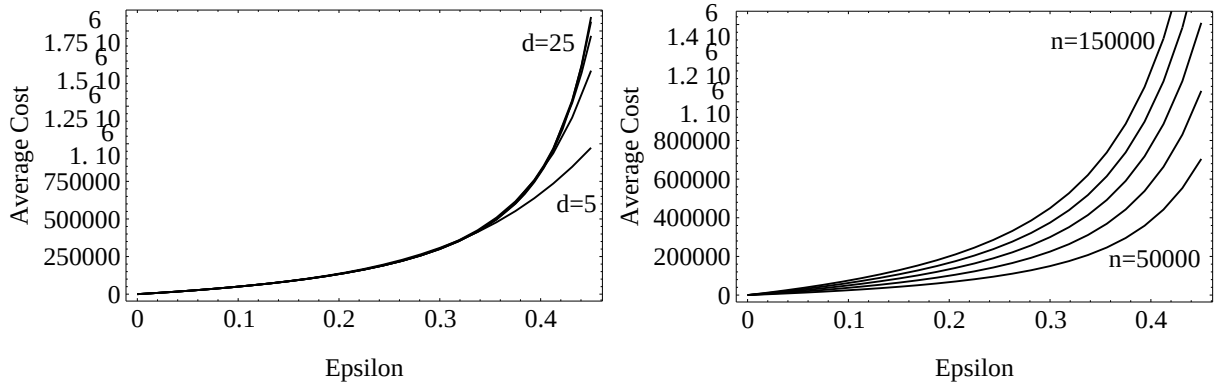


Figure 4: The average cost of the algorithm is independent of d and grows only linearly for small ϵ. (a) Uniformly distributed point set containing 100,000 points in 5-D, 10-D, 15-D, 20-D and 25-D spaces. (b) Uniformly distributed 15-D point set containing 50000, 75000, 100000, 125000 and 150000 points.

4.2 Normally Distributed Point Set

Next, we look at the case when the point set is normally distributed. If $\boldsymbol{X}$ is a point in the point set, we assume an independent and normal distribution with variance σ on each of it's coordinates:

$$f_{X_c}(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp \frac{-x^2}{2\sigma^2}$$

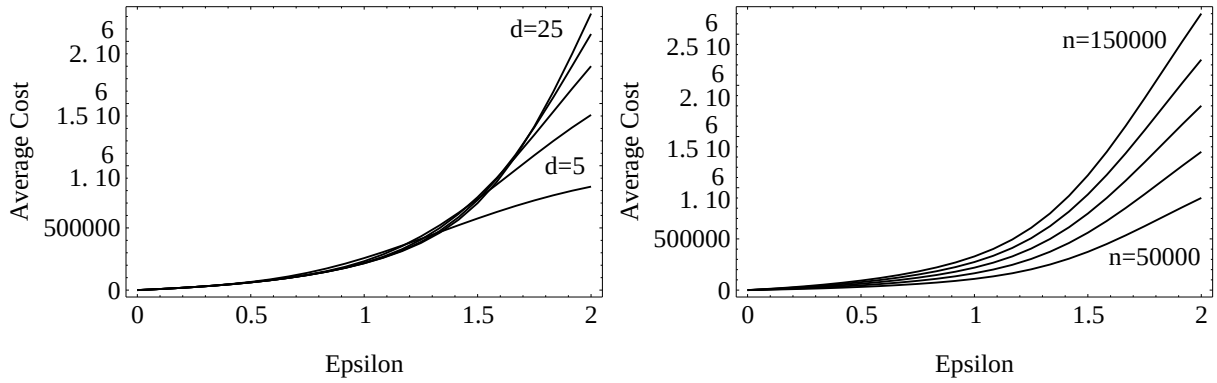
As before, using $Z_c = X_c - Q_c$, an expression for the density of Z_c can be obtained as

$$f_{Z_c|Q_c}(z) = \frac{1}{\sqrt{2\pi}\sigma} \exp \frac{-(z-Q_c)^2}{2\sigma^2}$$

P_c can then be written as

$$\begin{aligned} P_c &= P\{-\epsilon \leq Z_c \leq \epsilon \mid Q_c\} = \int_{-\epsilon}^{\epsilon} f_{Z_c|Q_c}(z)dz \\ &= \frac{1}{2}\left(\text{erf}\frac{\epsilon - Q_c}{\sigma\sqrt{2}} + \text{erf}\frac{\epsilon + Q_c}{\sigma\sqrt{2}}\right) \end{aligned}$$

This expression can be substituted into equation (4) and evaluated numerically to estimate cost for a given $\boldsymbol{Q}$. Figure 5 shows the cost as a function of ϵ for $\boldsymbol{Q} = \boldsymbol{0}$ and $\sigma = 1$. As with uniform distribution, we observe that when $\epsilon < 1$, the cost is nearly independent of d and grows linearly with n. In a variety of pattern classification problems, data take the form of individual Gaussian clusters or mixtures of Gaussian clusters. In such cases, the above results can serve as the basis for complexity analysis.



(a) (b)

Figure 5: The average cost of the algorithm is independent of d and grows only linearly for small ϵ. (a) Normally distributed point set containing 100,000 points in 5-D, 10-D, 15-D, 20-D and 25-D spaces ($\boldsymbol{Q} = \mathbf{0}$). (b) Normally distributed 15-D point set containing 50000, 75000, 100000, 125000 and 150000 points ($\boldsymbol{Q} = \mathbf{0}$).

5 Determining ϵ

It is apparent from the analysis in the preceding section that the cost of the proposed algorithm depends critically on ϵ. Setting ϵ too high results in a huge increase in cost with d, while setting ϵ too small may result in an empty candidate list. Although the freedom to choose ϵ may be attractive in some applications, it may prove non-intuitive and hard in others. In such cases, can we automatically determine ϵ so that the closest point can be found with high certainty? If the distribution of the point set is known, we can.

We first review well known facts about L_p norms. Figure 6 illustrates these norms for a few selected values of p. All points on these surfaces are equidistant (in the sense of the respective norm) from the central point. More formally, the L_p distance between two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ is defined as

$$L_p(\boldsymbol{a}, \boldsymbol{b}) = \left[\sum_k |a_k - b_k|^p \right]^{1/p}$$

These distance metrics are also known as Minkowski-p metrics. So how are these relevant to determining ϵ? The L_2 norm occurs most frequently in pattern recognition problems. Unfortunately, candidate list trimming in our algorithm does not find points within L_2, but within L_∞ (i.e. the hypercube). Since L_∞ bounds L_2, one can naively perform an exhaustive search inside L_∞. However, as seen in figure 7(a), this does not always correctly find the

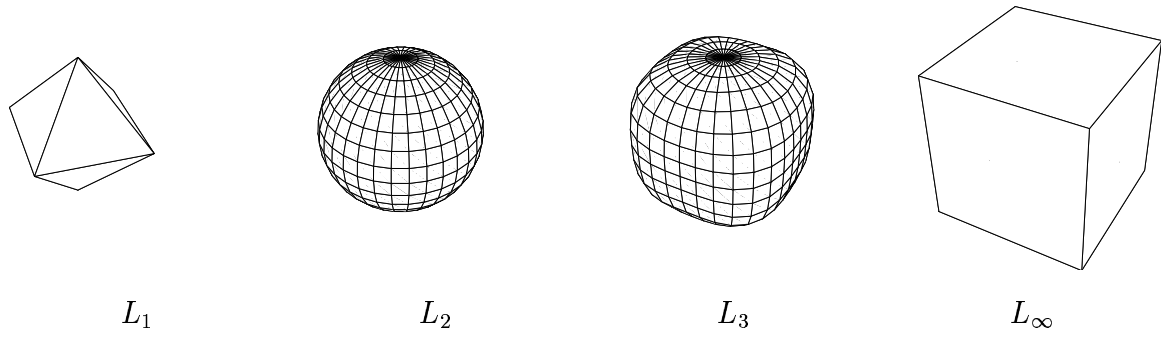


Figure 6: An illustration of various norms, also known as Minkowski p-metrics. All points on these surfaces are equidistant from the central point. The L_∞ metric bounds L_p for all p.

closest point. Notice that $\boldsymbol{P_2}$ is closer to $\boldsymbol{Q}$ than $\boldsymbol{P_1}$, although an exhaustive search within the cube will incorrectly identify $\boldsymbol{P_1}$ to be the closest. There is a simple solution to this problem. When performing an exhaustive search, impose an additional constraint that only points within an L_2 radius ϵ should be considered (see figure 7(b)). This, however, increases the possibility that the hypersphere is empty. In the above example, for instance, $\boldsymbol{P_1}$ will be discarded and we would not be able to find any point. Clearly then, we need to consider this fact in our automatic method of determining ϵ which we describe next.

We propose two methods to automatically determine ϵ. The first computes the radius of the smallest hypersphere that will contain *at least* one point with some (specified) probability. ϵ is set to this radius and the algorithm proceeds to find all points within a *circumscribing* hypercube of side ϵ. This method is however not efficient in very high dimensions; the reason being as follows. As we increase dimensionality, the difference between the hypersphere and hypercube volumes becomes so great that the hypercube "corners" contain far more points than the inscribed hypersphere. Consequently, the extra effort necessary to perform L_2 distance computations on these corner points is eventually wasted. So rather than find the circumscribing hypercube, in our second method, we simply find the length of a side of the smallest hypercube that will contain *at least* one point with some (specified) probability. ϵ can then be set to the length of this side. This leads to the problem we described earlier that, when searching some points outside a hypercube can be closer in the L_2 sense than points inside. We shall now describe both the methods in detail and see how we can remedy this problem.

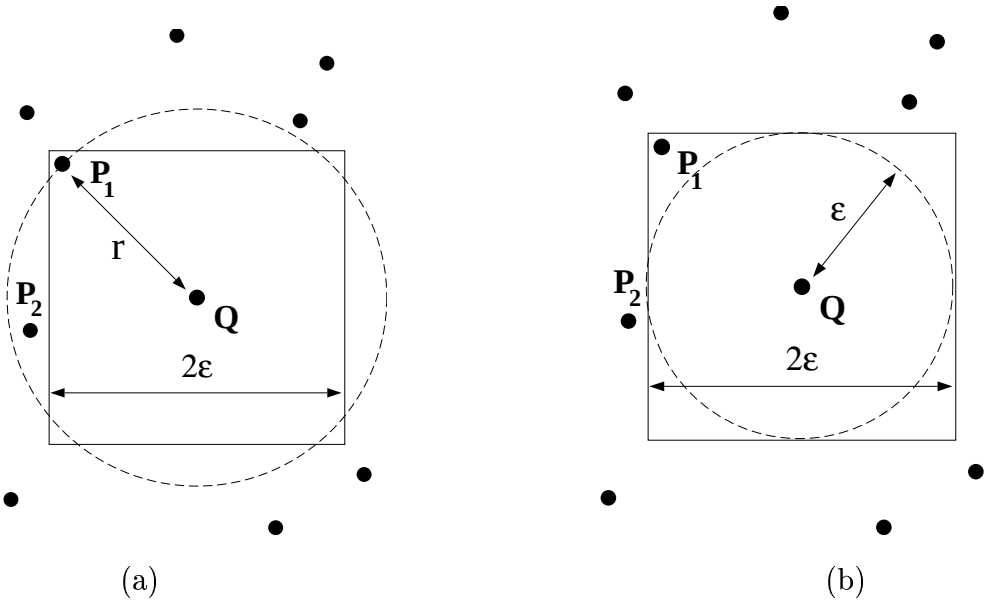


Figure 7: An exhaustive search within a hypercube may yield an incorrect result. (a) $\boldsymbol{P_2}$ is closer to $\boldsymbol{Q}$ than $\boldsymbol{P_1}$, but just an exhaustive search within the cube will incorrectly identify $\boldsymbol{P_1}$ as the closest point. (b) This can be remedied by imposing the constraint that the exhaustive search should consider only points within an L_2 distance ϵ from $\boldsymbol{Q}$ (given that the length of a side of the hypercube is ϵ).

5.1 Smallest Hypersphere Method

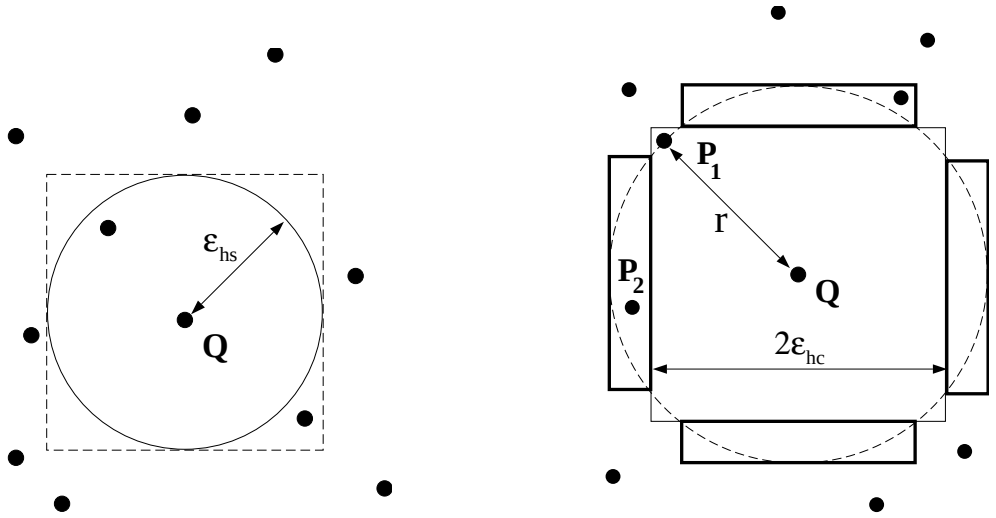
Let us now see how to analytically compute the minimum size of a hypersphere given that we want to be able guarantee that it is non empty with probability p. Let the radius of such a hypersphere be ϵ_{hs}. Let M be the total number of points within this hypersphere. Let $\boldsymbol{Q}$ be the novel point and define $\|\boldsymbol{Z}\|$ to be the distance between $\boldsymbol{Q}$ and any point in the point set. Once again, M is binomially distributed with the density

$$P\{M = k \mid \boldsymbol{Q}\} = (P\{\|\boldsymbol{Z}\| \leq \epsilon_{hs} \mid \boldsymbol{Q}\})^k (1 - P\{\|\boldsymbol{Z}\| \leq \epsilon_{hs} \mid \boldsymbol{Q}\})^{n-k} \begin{pmatrix} n \\ k \end{pmatrix}$$

Now, the probability p that there is at least one point in the hypersphere is simply

$$\begin{aligned} p &= P\{M > 0 \mid \boldsymbol{Q}\} = 1 - P\{M = 0 \mid \boldsymbol{Q}\} \\ &= 1 - (1 - P\{\|\boldsymbol{Z}\| \leq \epsilon_{hs} \mid \boldsymbol{Q}\})^n \end{aligned}$$

The above equation suggests that if we know $\boldsymbol{Q}$, the density $f_{\boldsymbol{Z}|\boldsymbol{Q}}(\boldsymbol{z})$, and the probability p, we can solve for ϵ_{hs}. For example, consider the case when the point set is uniformly



(a) (b)

Figure 8: ϵ can be computed using two methods: (a) By finding the radius of the smallest hypersphere that will contain at least one point with high probability. A search is performed by setting ϵ to this radius and constraining the exhaustive search within ϵ. (b) By finding the size of the smallest hypercube that will contain at least one point with high probability. When searching, ϵ is set to the length of a side. Additional searches have to be performed in the areas marked in bold.

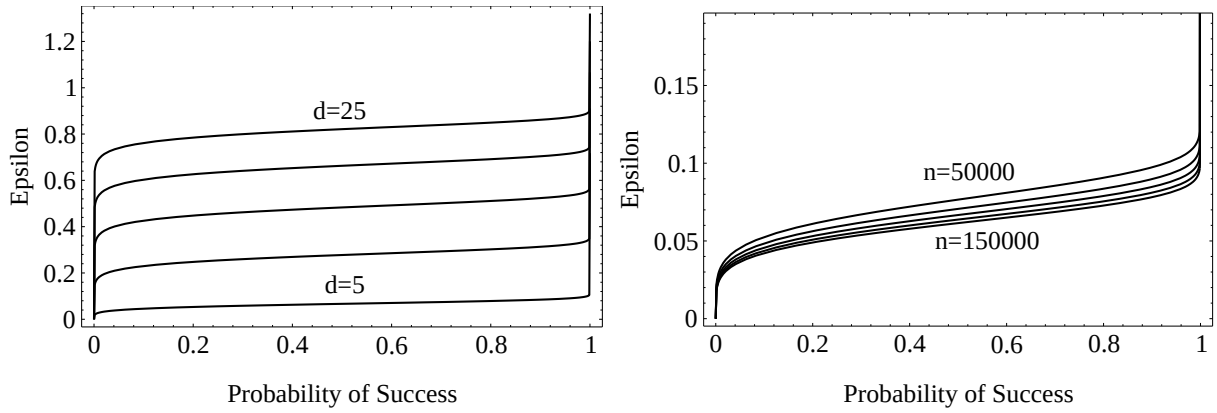
distributed with density given by equation (6). The cumulative distribution function of $\|\boldsymbol{Z}\|$ is the uniform distribution integrated within a hypersphere; which is simply it's volume. Thus,

$$P\{\|\boldsymbol{Z}\| \leq \epsilon_{hs} \mid \boldsymbol{Q}\} = \frac{2{\epsilon_{hs}}^d \pi^{d/2}}{l^d d\Gamma(d/2)}$$

Substituting the above in equation 9 and solving for ϵ_{hs}, we get

$$\epsilon_{hs} = \left(\frac{l^d d\Gamma(d/2)}{2\pi^{d/2}} \left(1 - (1-p)^{1/n}\right) \right)^{1/d} \tag{9}$$

Using equation (9), ϵ_{hs} is plotted against probability for two cases. In figure 9(a), d is fixed to different values between 5 to 25 with n is fixed to 100000, and in figure 9(b), n is fixed to different values between 50000 to 150000 with d fixed to 5. Both the figures illustrate an important property, which is that large changes in the probability p result in very small changes in ϵ_{hs}. This suggests that ϵ_{hs} can be set to the right hand "knee" of both the curves where probability is very close to unity. In other words, it is easy to guarantee that at least one point is within the hypersphere. A search can now be performed by setting the length



(a) (b)

Figure 9: The radius ϵ necessary to find a point inside a hypersphere varies very little with probability. This means that ϵ can be set to the knee where probability is close to unity. (a) Uniformly distributed point set containing 100000 points in 5, 10, 15, 20 and 25 dimensional space. (b) Uniformly distributed 5-D point set containing 50000, 75000, 100000, 125000 and 150000 points.

of a side of the circumscribing hypercube to ϵ_{hs} and by imposing an additional constraint during exhaustive search that only points within an L_2 distance ϵ_{hs} should be considered.

5.2 Smallest Hypercube Method

As before, we attempt to analytically compute the size of the smallest hypercube given that we want to be able guarantee that it is non empty with probability p. Let M be the number of points within a hypercube of size $2\epsilon_{hc}$. Define Z_c to be the distance between the c^{th} coordinate of a point set point and the novel point $\boldsymbol{Q}$. Once again, M is binomially distributed with the density

$$P\{M = k \mid \boldsymbol{Q}\} = \left(\prod_{c=1}^{d} P\{-\epsilon_{hc} \leq Z_c \leq \epsilon_{hc} \mid Q_c\}\right)^k \left(1 - \prod_{c=1}^{d} P\{-\epsilon_{hc} \leq Z_c \leq \epsilon_{hc} \mid Q_c\}\right)^{n-k} \begin{pmatrix} n \\ k \end{pmatrix}$$

Now, the probability p that there is at least one point in the hypercube is simply

$$\begin{aligned} p &= P\{M > 0 \mid \boldsymbol{Q}\} = 1 - P\{M = 0 \mid \boldsymbol{Q}\} \\ &= 1 - \left(1 - \prod_{c=1}^{d} P\{-\epsilon_{hc} \leq Z_c \leq \epsilon_{hc} \mid Q_c\}\right)^n \end{aligned} \tag{10}$$

Again, above equation suggests that if we know $\boldsymbol{Q}$, the density $f_{Z_c|Q_c}(z)$, and the probability p, we can solve for ϵ_{hc}. If the point set is uniformly distributed, an expression for ϵ_{hc} can be

obtained in closed form as follows. Let the density of the uniform distribution be given by equation (6). Using equation (7) we get,

$$\prod_{c=1}^{d} P\{-\epsilon_{hc} \leq Z_c \leq \epsilon_{hc} \mid Q_c\} = \left(\frac{2\epsilon_{hc}}{l}\right)^d$$

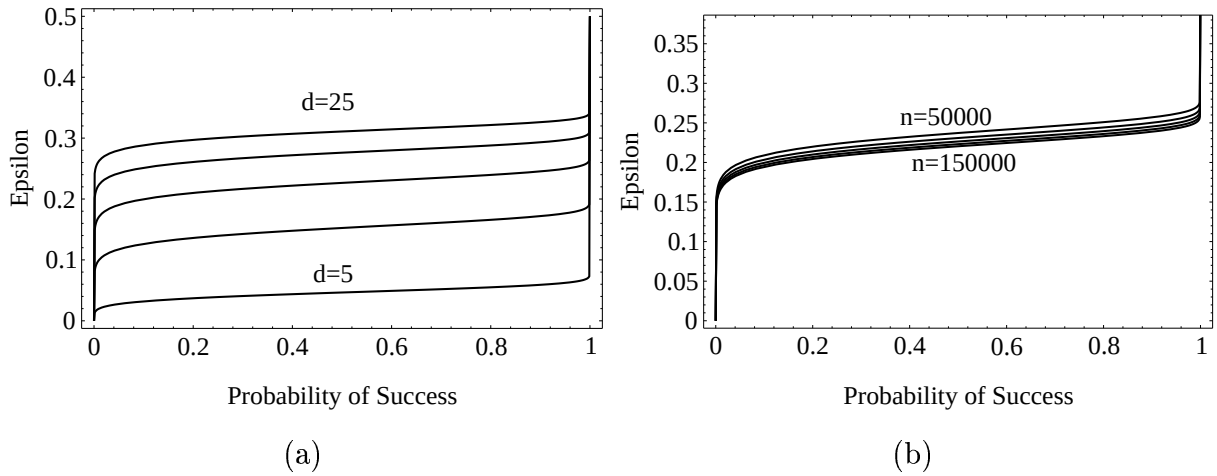
Substituting the above in equation (10) and solving for ϵ_{hc}, we get

$$\epsilon_{hc} = \frac{l}{2}\left(1 - (1-p)^{1/n}\right)^{1/d} \tag{11}$$

Using equation (11), ϵ_{hc} is plotted against probability for two cases. In figure 10(a), d is fixed to different values between 5 to 25 with n is fixed to 100000, and in figure 10(b), n is fixed to different values between 50000 to 150000 with d fixed to 5. These are similar to the graphs obtained in the case of a hypersphere and again, ϵ_{hc} can be set to the right hand "knee" of both the curves where probability is very close to unity. Notice that the value of ϵ_{hc} required for the hypercube is much smaller than that required for the hypersphere, especially in high d. This is precisely the reason why we prefer the second method.

Recall that it is not sufficient to simply search for the closest point within a hypercube because a point outside can be closer than a point inside. To remedy this problem, we suggest the following technique. First, an exhaustive search is performed to compute the L_2 distance to the closest point within the hypercube. Call this distance r. In figure 8(b), the closest point $\boldsymbol{P_1}$ within the hypercube is at a distance of r from $\boldsymbol{Q}$. Clearly, if a closer point exists, it can only be within a hypersphere of radius r. Since parts of this hypersphere lie outside the original hypercube, we also search in the hyper-rectangular regions shown in bold (by performing additional list trimmings). When performing an exhaustive search in each of these hyper-rectangles, we impose the constraint that a point is considered only if it is *less* than distance r from $\boldsymbol{Q}$. In figure 8(b), $\boldsymbol{P_2}$ is present in one such hyper-rectangular region and happens to be closer to $\boldsymbol{Q}$ than $\boldsymbol{P_1}$. Although this method is more complicated, it gives excellent performance in sparsely populated high dimensional spaces (such as a high dimensional uniform distribution).

To conclude, we wish to emphasize that both the hypercube and hypersphere methods can be used interchangeably and both are guaranteed to find the closest point within ϵ. However, the choice of which one of these methods to use should depend on the dimensionality of the space and the local density of points. In densely populated low dimensional spaces, the hypersphere method performs quite well and searching the hyper-rectangular regions



(a) (b)

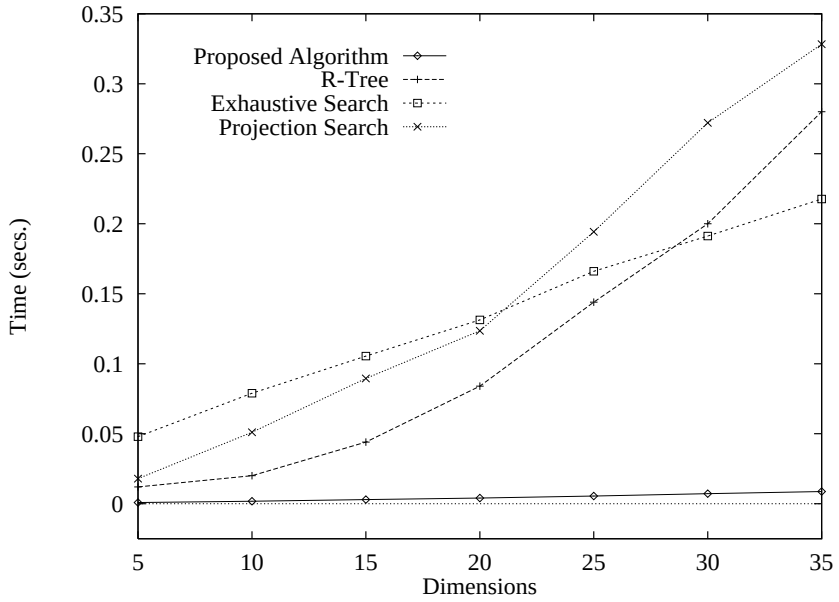
Figure 10: The value of ϵ necessary to find a point inside a hypercube varies very little with probability. This means that ϵ can be set to the knee where probability is close to unity. (a) Uniformly distributed point set containing 100000 points in 5, 10, 15, 20 and 25 dimensional space. (b) Uniformly distributed 5-D point set containing 50000, 75000, 100000, 125000 and 150000 points.

is not worth the additional overhead. In sparsely populated high dimensional spaces, the effort needed to exhaustively search the huge circumscribing hypercube is far more than the overhead of searching the hyper-rectangular regions. Finally, although the above discussion is relevant only for the L_2 norm, an equivalent analysis can be easily performed for any other norm.

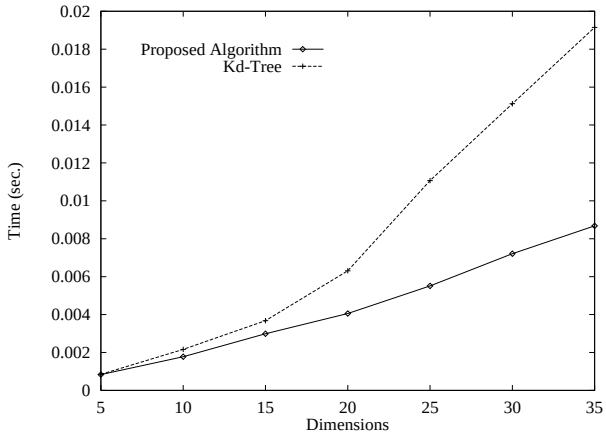
6 Benchmarks

We have performed an extensive set of benchmarks on the proposed algorithm. A pseudo code version of the implementation we used for the benchmarks is given in Appendix A. We looked at two representative classes of search problems that may benefit from the algorithm. In the first class, the data has some structure. This is the case, for instance, when points are uniformly or normally distributed. The second class of problems are unstructured, for instance, when points lie in a high dimensional multivariate manifold, and it is difficult to say anything about their distribution.

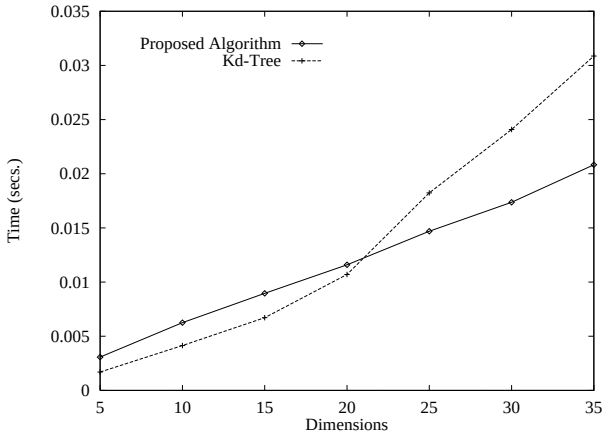
The benchmarks were conducted first for the unstructured case, and next for the structured case. In the first case, we used points that lie on a high dimensional trivariate manifold with complex structure. This manifold was used in the appearance matching experiments



(a)



(b)



(c)

Figure 11: The average execution time of the proposed algorithm is benchmarked for an unstructured problem. The point set is constructed by sampling a high dimensional trivariate manifold. (a) The manifold is sampled to obtain 31,752 points. The proposed algorithm is more than two orders of magnitude faster than the other algorithms. (b) The manifold is sampled as before to obtain 31,752 points. (c) The manifold is sampled to obtain 107,163 points. The k-d tree algorithm is slightly faster in low dimension but degrades rapidly with increase in dimension.

conducted by Nayar and Murase [Nayar *et al.*-1994]. In Figure 11(a) the proposed algorithm is compared with exhaustive, projection and R-tree search algorithms, for fixed n and d varying from 5 to 35. The manifold was sampled at 42x54x14 equally spaced locations to obtain 31,752 discrete points. The execution time per search was found by averaging the total execution time required to perform 10,000 closest point searches. The test set of 10,000 points was obtained by sampling the manifold at random locations and adding white noise with standard deviation .01 to emulate a realistic setting. It can be seen that in high dimensions the proposed algorithm is at least two orders of magnitude faster than all the other search techniques. Notice that the time taken by the R-tree algorithm grows very rapidly with d.

Figures 11(b) and 11(c) compare the proposed algorithm with the k-d tree algorithm with d varying from 5 to 25. The exhaustive, projection and R-tree algorithms are not included because they are far slower. In Figure 11(b), the manifold was sampled as before at 42x54x14 equally spaced locations to obtain 31,752 points. In Figure 11(c), the manifold was sampled at 63x81x21 equally spaced locations to obtain 107,163 points. In both cases, a test set of 10,000 randomly sampled manifold points and white noise with standard deviation .01 was used and the execution time averaged over this test set. In Figure 11(b), the proposed algorithm is faster than the k-d tree for all d, while in Figure 11(c), the proposed algorithm is faster for all $d > 21$. This can be explained by the fact that the cost of our algorithm has a linear dependence on n, but with a lower dependence on d.

In the second set of benchmarks, we tested the algorithm with structured data for d varying from 5 to 25. Two commonly occurring distributions, normal and uniform were used. The proposed algorithm was compared with the k-d tree and exhaustive search algorithms. Figures 12(a) and 12(b) show the execution times when the point set is normally distributed with variance 1.0 and containing 30,000 and 100,000 points, respectively. The execution time was calculated by averaging over the total time required to perform 10,000 closest point searches. This test set of 10,000 points was also normally distributed with variance 1.0. In both cases, ϵ was computed using the analysis in Section 5. Observe that the proposed algorithm is faster than the k-d tree algorithm for all d in Figure 12(a). In Figure 12(b), the proposed algorithm is faster for $d > 12$. Also notice that the k-d tree algorithm actually runs slower than exhaustive search for $d > 15$. In high dimensions, the space is so sparsely populated that the radius of the query hypersphere is very large. Consequently,

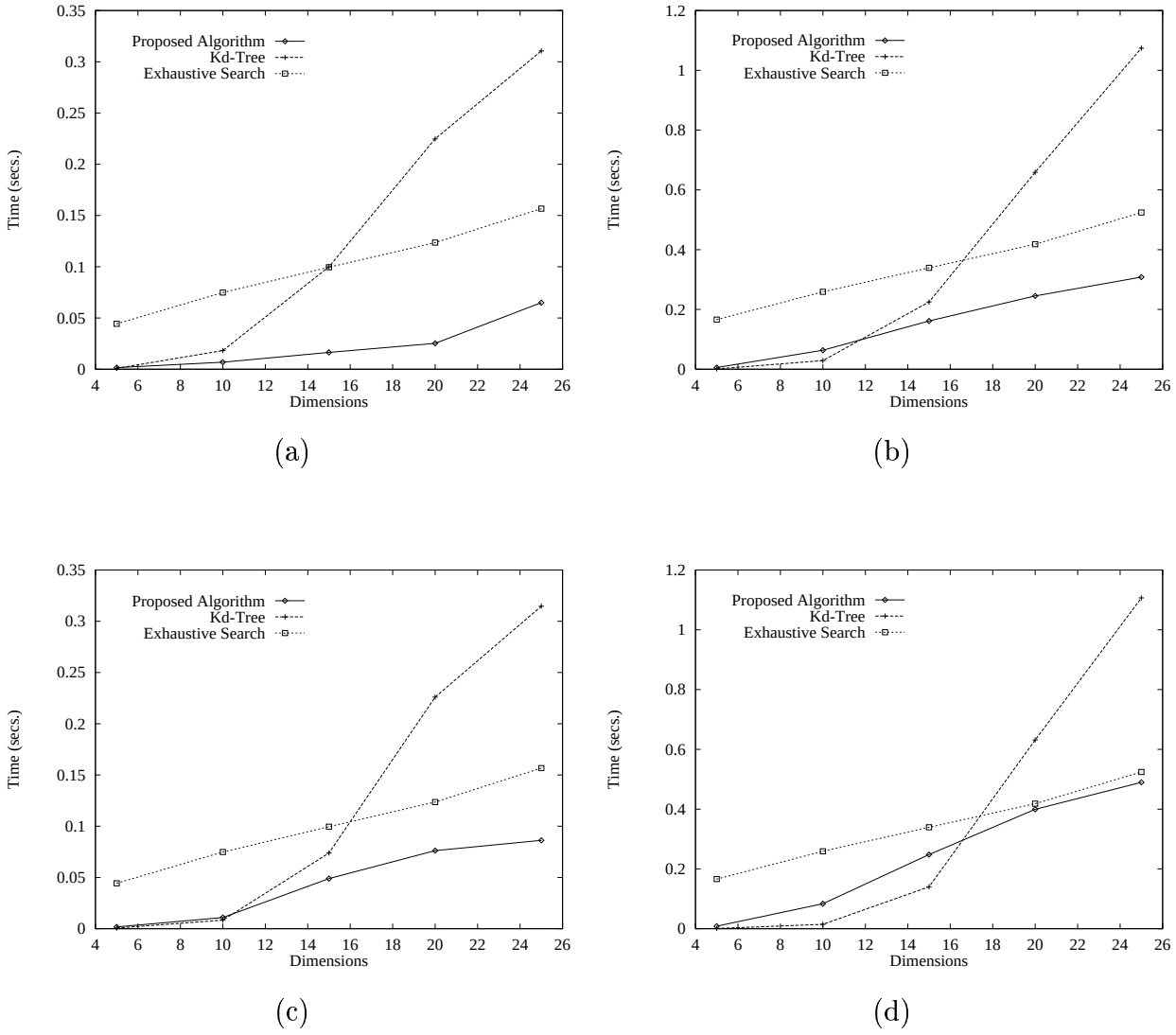


Figure 12: The average execution time of the proposed algorithm is benchmarked for structured problems. (a) Point set is normally distributed and contains 30,000 points with variance 1.0. (b) Point set is normally distributed and contains 100,000 points with variance 1.0. The proposed algorithm is clearly faster in high d. (c) Point set is uniformly distributed and contains 30,000 points with extent 1.0. (d) Point set is uniformly distributed and contains 100,000 points with extent 1.0. The proposed algorithm does not perform as well for uniform distributions due to the extreme sparseness of the point set in high d.

the hypersphere intersects almost all the buckets and thus a large number of points are examined. This, along with the additional overhead of traversing the tree structure, makes the k-d tree algorithm run slower than exhaustive search.

Figures 12(c) and 12(d) show execution times when the point set is uniformly distributed with unit extent ($l = 1$) and containing 30,000 and 100,000 points, respectively. As before, the execution time was calculated by averaging over the total time required to perform 10,000 closest point searches. This test set of 10,000 points was also uniformly distributed with unit extent. In both cases, ϵ was computed based on the analysis in the Section 5. For uniform distribution, the proposed algorithm does not perform as well, although, it does appear to be slightly faster than the k-d tree and exhaustive search algorithms. The reason is that the high dimensional space is very sparsely populated and hence requires ϵ to be quite large. As a result, the algorithm examines almost all points, thereby approaching exhaustive search.

7 Applications

We now demonstrate two applications where a fast and efficient high dimensional search technique is desirable. The first, real time object recognition, requires the closest point to be found among 36,000 points in a 35-D space. In the second, motion estimation for MPEG coding, the closest point needs to be found from 961 points in a 256-D space.

7.1 Real Time Object Recognition

We used the Columbia Object Image Library [Nayar *et al.*-1996] along with the SLAM software package for appearance matching [Nene and Nayar-1994] to compute 100 univariate manifolds in a 35-D eigenspace. These manifolds correspond to appearance models of the 100 objects shown in Figure 13(a) [Murase and Nayar-1995]. Object recognition is performed by first projecting a novel image to eigenspace to obtain a single point and then searching for the closest manifold (object identity) and the closest point on that manifold (object pose). Each of the 100 manifolds were sampled at 360 equally spaced points to obtain 36,000 discrete points in 35-D space. Figure 13(b) shows the time taken to search by the different algorithms. The search time was calculated by averaging the total time taken to perform 100,000 closest point searches. This set of 100,000 recognition test points were obtained by sampling the 100 manifolds at random points and adding white noise with standard deviation .01 to account

(a)

Algorithm	Time (secs.)
Proposed Algorithm	.0025
k-d tree	.0045
Exhaustive Search	.1533
Projection Search	.2924

(b)

Figure 13: The proposed algorithm was used to recognize and estimate pose of hundred objects using the Columbia Object Image Library. (a) Twenty of the hundred objects are shown. The point set consisted of 36,000 points (360 for each object) in 35-D eigenspace. (b) The average execution time per search is compared with other algorithms.

for affine distortions in image projection and image sensor noise. It can be seen that the proposed algorithm outperforms all the other techniques.

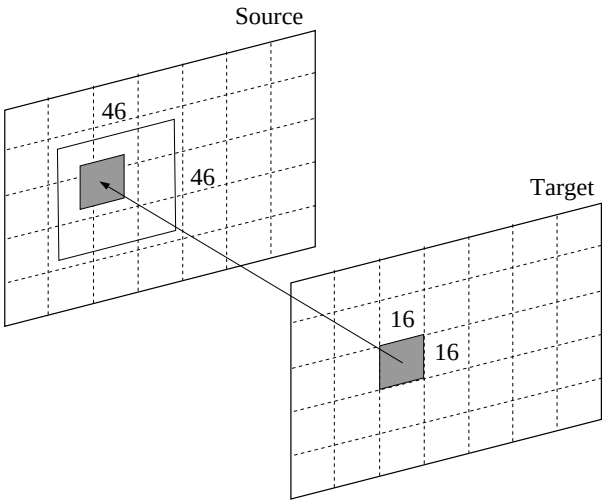
7.2 Motion Vector Estimation in MPEG Coding

MPEG (Motion Pictures Expert Group) coding is an important tool for compression of digital movies. It finds use in a variety of commercial applications, such as, Digital Satellite TV, Video on Demand systems, and High Definition Television (HDTV). It is well known that MPEG coding exploits the temporal redundancy between adjacent frames to achieve very high compression [Netravali-1995]. Temporal redundancy implies the high degree of similarity that exists between consecutive image frames. Hence, rather than transmit an entire frame, only the differences need to be transmitted. However, one cannot simply transmit the difference because even slight motion can produce a very large difference image that can be as expensive to transmit as the original frame.

MPEG deals with the above problem in the following manner. A *target* frame is divided into 16x16 regions known as Macro-Blocks (MBs) (see Figure 14(b)). The position of each of these MBs in the *source* frame is found by searching in a fixed size region (typically 46x46) around the MB. Once these positions are known, one can simply transmit the MB motion vectors rather than intensity images. Obviously 3-D effects such as occlusion and intensity variations will result in imperfect reconstruction. Thus a difference image computed *after* compensating for motion is also transmitted.

Estimating motion vectors is the biggest computational bottleneck in MPEG coding. In the past, substantial effort has been dedicated to overcoming this problem. The most effective techniques are based on gradient based algorithms. Although these techniques are fast, it is hard to guarantee that they will work under any circumstance, and it has been observed that simple exhaustive search based algorithms for finding the motion vectors yield the best result.

We used our search algorithm to perform motion estimation and compared the results with those obtained using the traditional exhaustive search based approach. Figure 14(a) shows the motion vectors computed for each MB using the proposed algorithm for the widely used flower garden sequence. Figure 14(b) shows the motion vectors obtained by an exhaustive search. Discrepancies between a few of the vectors arise due to the fact that we use the L_∞ norm as opposed to the L_2 norm in exhaustive search. Our algorithm ran almost an order



(a)

(b)

(c)

Algorithm	Time (secs.)
Proposed Algorithm	1.5
Correlation (SSD)	11

(d)

Figure 14: MPEG coding achieves very high compression by exploiting temporal redundancy. This involves dividing each frame into 16x16 Macro Blocks (MBs) and performing motion estimation (visual correspondence) on the MBs between consecutive frames. (a) A motion vector is computed for every MB by searching in a 46x46 region in the preceding frame. Motion vectors obtained by running (b) the proposed algorithm and (c) exhaustive search. The slight differences in the vectors are due to the use of L_∞ as opposed to L_2. (d) The average execution time per frame for the proposed algorithm is almost an order of magnitude faster than exhaustive search.

of magnitude faster than exhaustive search, as shown in Figure 14(e). Note that we were able to compare only exhaustive search because it takes too much time to construct a k-d tree or an R-tree for each MB. On the other hand, we are able to construct the ordered set and the forward and backward maps extremely quickly (linear time) by exploiting the spatial redundancy between adjacent search regions [Nene and Nayar-1996]. We conclude by noting that the above solution can also be used in other areas where visual correspondence needs to be established, such as, binocular stereo and structure from motion in computational vision.

8 Hardware Architecture

A major advantage of our algorithm is its simplicity. Recall that the main computations performed by the algorithm are simple integer map lookups (backward and forward maps) and two integer comparisons (to see if a point lies within hyperplane boundaries). Consequently, it is possible to implement the algorithm in hardware using off-the-shelf, inexpensive components. This is hard to envision in the case of any competitive techniques such as k-d trees or R-trees, given the difficulties involved in constructing parallel stack machines.

The proposed architecture is shown in Figure 15. A Field Programmable Gate Array (FPGA) acts as an algorithm state machine controller and performs I/O with the CPU. The Dynamic RAMs (DRAMs) hold the forward and backward maps which are downloaded from the CPU during initialization. The CPU initiates a search by performing a binary search to obtain the hyperplane boundaries. These are then passed on to the search engine and held in the Static RAMs (SRAMs). The FPGA then independently begins the candidate list construction and trimming. A candidate is looked up in the backward map and each of the forward maps. The integer comparator returns a true if the candidate is within range, else it is discarded. After trimming all the candidate points by going through the dimensions, the final point list (in the form of point set indices) is returned to the CPU for exhaustive search and/or further processing. Note that although we have described an architecture with a single comparator, any number of them can be added and run in parallel with a near linear performance scaling in the number of comparators. While the search engine is trimming the candidate list, the CPU is of course free to carry out other tasks in parallel.

We have begun implementation of the proposed architecture. The result is intended to be a small low-cost SCSI based module that can be plugged in to any standard workstation

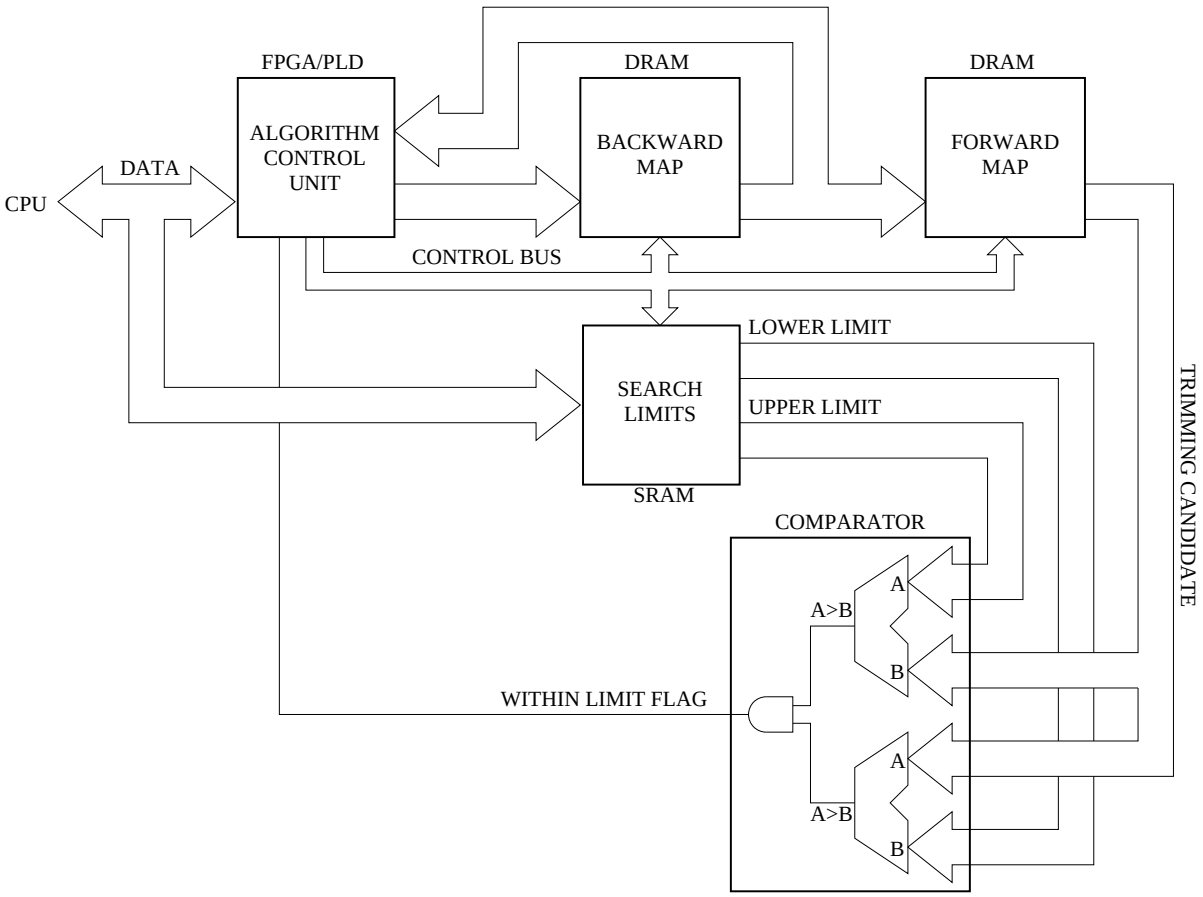


Figure 15: Architecture for an inexpensive hardware search engine that is based on the proposed algorithm.

or PC. We estimate the module to result in a 100 fold speedup over an optimized software implementation.

Acknowledgements

We wish to thank Simon Baker and Dinkar Bhat for their detailed comments, criticisms and suggestions on the paper.

This research was conducted at the Center for Research on Intelligent Systems at the Department of Computer Science, Columbia University. It was supported in parts by ARPA Contract DACA-76-92-C-007, DOD/ONR MURI Grant N00014-95-1-0601, and a NSF National Young Investigator Award.

A Pseudo Code Implementation

```
/* This function performs pre-processing on Point Set to obtain an Ordered Set
   along with Forward & Backward maps */

Preprocess(double PointSet[d][n], double OrderedSet[d][n], int BMap[d],
   int FMap [d][n])
{
  int TempMap[n];

  /* Create Backward map for first dimension by sorting first dimension in
     Point Set */

  Sort(PointSet[1], TempMap);
  for i = 1 to n {
    OrderedSet[1][i] = PointSet[1][TempMap[i]];
    BMap[i] = TempMap[i];
    FMap[1][TempMap[j]] = j;
  }

  /* Create Forward map for each dimension by sorting corresponding coordinate
     in the Point Set */

  for i = 2 to d {
    Sort(PointSet[i], TempMap);
    for j = 1 to n {
      OrderedSet[i][j] = PointSet[i][TempMap[j]];
      FMap[i][TempMap[j]] = j;
    }
  }
}

/* This function is called by Preprocess() for sorting individual dimensions in
```

```
   the Point Set */

Sort(double PointSet[n], int TempMap[n])
{
  /* Initialize TempMap */

  for i = 1 to n
    TempMap[i] = i;

  /* Bubble sort */

  for i = 1 to n
    for j = 1 to n - i
      if PointSet[TempMap[j]] > PointSet[TempMap[j + 1]] {
        t = PointSet[TempMap[j + 1]];
        PointSet[TempMap[j + 1]] = PointSet[TempMap[j]];
        PointSet[TempMap[j]] = t;
      }
}

/* This function is called to perform closest point search. At the  end  of
   the search, it returns an index into the PointSet array which represents
   the closest point */

int Closest(double P[d], double Epsilon, double OrderedSet[d][n], int BMap[n],
    int FMap[d][n])
{
  /* Perform binary Search on first dimension */

  Bottom = BinarySearch(OrderedSet[1], P[1] - Epsilon);
  Top = BinarySearch(OrderedSet[1], P[1] + Epsilon);

  /* Create list */
```

```
  ListElem = 0;
  for i = Bottom to Top
    List[++ListElem] = BMap[i];

  /* Trim list with binary searches on other dimensions along with lookups */

  for i = 2 to d {
    Bottom = BinarySearch(OrderedSet[i], P[i] - Epsilon);
    Top = BinarySearch(OrderedSet[i], P[i] + Epsilon);

    m = ListElem;
    ListElem = 0;
    for j = 1 to m
      if FMap[i][List[j]] >= Top && FMap[i][List[j]] <= Bottom
        List[++ListElem] = List[j];
  }

  /* Perform exhaustive search on remaining points */

  max = MAXDOUBLE;
  for i = 1 to ListElem {
    t = 0;
    for j = 1 to d
      t += (P[j] - OrderedSet[j][FMap[j][List[i]]]) ** 2;
    if t < max {
      max = t;
      pos = List[i];
    }
  }

  return pos;
}

/* This function performs a binary search on an array. It returns the index of
```

```
   the element which is the result of the search. */

int BinarySearch(double OrderedSet[n], double v)
{
  Bottom = 0;
  Top = n;

  while Top > Bottom + 1 {
    Center = (Bottom + Top) / 2;
    if v < OrderedSet[Center]
      Top = Center;
    else
      Bottom = Center;
  }

  return Bottom;
}
```

References

[Aho *et al.*, 1974] A. V. Aho, J. E. Hopcroft and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[Arya, 1995] S. Arya. Nearest neighbor searching and applications. Technical Report CS-TR-3490, University of Maryland, College Park, MD 20742-3275, June 1995.

[Aurenhammer, 1991] F. Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, September 1991.

[Beckmann *et al.*, 1990] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD*, pages 322–331, Atlantic City, NJ, May 1990.

[Bentley and Friedman, 1979] J. L. Bentley and J. H. Friedman. Data structures for range searching. *Computing Surveys*, 11(4):397–409, December 1979.

[Bentley and Weide, 1980] J. L. Bentley and B. W. Weide. Optimal expected-time algorithms for closest point problems. *ACM Transactions on Mathematical Software*, 6(4):563–580, December 1980.

[Bentley, 1975] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[Bentley, 1979] J. L. Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering*, SE-5(4):333–340, July 1979.

[Califano and Mohan, 1991] A. Califano and R. Mohan. Multidimensional indexing for recognizing visual shapes. In *Proc. IEEE Conference on Computer Vision and Pattern Recognition*, pages 28–34, June 1991.

[Clarkson, 1988] K. L. Clarkson. A randomized algorithm for closest-point queries. *SIAM J. Computing*, 17(4):830–847, August 1988.

[Dobkin and Lipton, 1976] D. Dobkin and R. J. Lipton. Multidimensional searching problems. *SIAM J. Comuting*, 5(2):181–186, June 1976.

[Edelsbrunner, 1987] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. Springer, Berlin-Heidelberg, 1987.

[Friedman *et al.*, 1975] J. H. Friedman, F. Baskett and L. J. Shustek. An algorithm for finding nearest neighbors. *IEEE Transactions on Computers*, pages 1000–1006, October 1975.

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

[Fukunaga and Narendra, 1975] K. Fukunaga and P. M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Transactions on Computers*, pages 750–753, July 1975.

[Gargantini, 1982] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, December 1982.

[Guttman, 1984] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD*, pages 47–57, June 1984.

[Klee, 1980] V. Klee. On the complexity of d-dimensional voronoi diagrams. *Arch. Math*, 34:75–80, 1980.

[Knuth, 1973] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Mass., 1973.

[Lomet and Salzberg, 1990] D. B. Lomet and B. Salzberg. The hb-tree: a multiattribute indexing method with good guaranteed performance. *ACM TODS*, 15(4):625–658, December 1990.

[Murase and Nayar, 1995] H. Murase and S. K. Nayar. Visual learning and recognition of 3d objects from appearance. *International Journal of Computer Vision*, 14(1):5–24, January 1995.

[Nayar *et al.*, 1994] S. K. Nayar, H. Murase and S. A. Nene. Learning, positioning, and tracking visual appearance. In *Proceedings of IEEE International Conference on Robotics and Automation*, San Diego, May 1994.

[Nayar *et al.*, 1995] S. K. Nayar, S. Baker and H. Murase. Parametric feature detection. Technical Report CUCS-028-95, Columbia University, 1995.

[Nayar *et al.*, 1996] S. K. Nayar, S. A. Nene and H. Murase. Real-time 100 object recognition system. In *Proceedings of IEEE International Conference on Robotics and Automation*, Twin Cities, May 1996.

[Nene and Nayar, 1994] S. A. Nene and S. K. Nayar. Slam: A software library for appearance matching. In *Proceedings of ARPA Image Understanding Workshop*, Monterey, November 1994. Also Tech. Rep. CUCS-019-94.

[Nene and Nayar, 1996] S. A. Nene and S. K. Nayar. A simple algorithm for efficient motion estimation in mpeg coding. Technical report, Department of Computer Science, Columbia University, 1996. In preparation.

[Netravali, 1995] A. N. Netravali. *Digital Pictures : Representation, Compression, and Standards.* Plenum Press, New York, 2nd edition, 1995.

[Petrakis and Faloutsos, 1994] E. G. M. Petrakis and C. Faloutsos. Similarity searching in large image databases. Technical Report CS-TR-3388, Department of Computer Science, University of Maryland, December 1994.

[Preparata and Shamos, 1985] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer, New York, 1985.

[Robinson, 1981] J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *ACM SIGMOD*, pages 10–18, 1981.

[Roussopoulos and Leifker, 1985] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed r-trees. In *ACM SIGMOD*, May 1985.

[Sellis *et al.*, 1987] T. Sellis, N. Roussopoulos and C. Faloutsos. The r+-tree: A dynamic index for multidimensional objects. In *Proceedings of 13th International Conference on VLDB*, pages 507–518, September 1987.

[Sproull, 1991] R. F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6:579–589, 1991.

[Wolfson, 1990] H.J. Wolfson. Model-based object recognition by geometric hashing. In *Proc. 1st European Conf. Comp. Vision*, pages 526–536, April 1990.

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

[Yunck, 1976] T. P. Yunck. A technique to identify nearest neighbors. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-6(10):678–683, October 1976.