A Simple Algorithm for Nearest Neighbor Search in High Dimensions

Sameer A. Nene and Shree K. Nayar

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 is demonstrated as an example application. 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. A C++ implementation of our algorithm is available upon request to `search@cs.columbia.edu/CAVE/`.

Index Terms—Pattern classification, nearest neighbor, searching by slicing, benchmarks, object recognition, 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" [21]. 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 [24], visual positioning, tracking, and inspection [26], and feature detection [25]. 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 ($d > 256$) [29], 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, for small ϵ, grows very slowly with 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 [14]. Friedman's 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 1D 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 in high d.

This simple projection search was improved upon by Yunck [40]. He utilizes a precomputed data structure which maintains a mapping from the sorted to the unsorted (original) coordinate arrays. In addition to this mapping, an indicator array of n elements is used. Each element of the

• *The authors are with the Department of Computer Science, Columbia University, New York. E-mail: {sameer, nayar}@cs.columbia.edu.*

Manuscript received 5 Feb. 1996; revised 29 May 1997. Recommended for acceptance by A. Webb.
For information on obtaining reprints of this article, please send e-mail to: tpami@computer.org, and reference IEEECS Log Number 105305.

indicator array, henceforth called an *indicator*, corresponds to a point. At the beginning of a search, all indicators are initialized to the number "1." As before, a small offset ϵ is subtracted from and added to each of the novel point $\boldsymbol{Q}$'s coordinates to obtain two values. Two binary searches are performed on each of the d sorted arrays to locate the positions of both the values. The mapping from sorted to unsorted arrays is used to find the points corresponding to the coordinates *in between* these values. Indicators corresponding to these points are (binary) shifted to the left by one bit and the entire process repeated for each of the d dimensions. At the end, points whose indicators have the value 2^d must lie within an 2ϵ hypercube. An exhaustive search can now be performed on the hypercube points to find the nearest neighbor.

With the above data structure, Yunck was able to find points within the hypercube using primarily integer operations. However, the *total* number of machine operations required (integer and floating point) to find points within the hypercube are similar to that of Friedman's algorithm (roughly $O(nd\epsilon)$). Due to this, and the fact that most modern CPUs do not significantly penalize floating point operations, the improvement is only slight (benchmarked in a later section). We propose a data structure that significantly reduces the total number of machine operations required to locate points within the hypercube to roughly $O\left(n\epsilon + n\left(\frac{1-\epsilon^d}{1-\epsilon}\right)\right)$. 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,
(e) Randomized algorithms.

The algorithm described in this paper falls into category (d). The algorithms can be further categorized into those that work in vector spaces and those that work in metric spaces. Categories (b)-(d) fall into the former, while category (a) falls into the later. Metric space search techniques are used when it is possible to somehow compute a distance measure between sample "points" or pieces of data but the space in which the points reside lacks an explicit coordinate structure. In this paper, we focus only on vector space techniques. For a detailed discussion on searching in metric spaces, refer to [13], [23], and [37].

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 (one or two) dimensional space and then indexing in this low dimensional space have been proposed. Such a dimensionality reduction is called geometric hashing [38], [9]. 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 multidimensional 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 2D 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 [3], [12], [20], and [31]. Another algorithm of interest is the 1D binary search generalized to d dimensions [11]. 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 [5], [6]. The k-d tree is a data structure that 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 [7], [8], [15].

k-d trees are extremely versatile and efficient to use in low dimensions. However, the performance degrades ex-

ponentially[1] 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 [15]. Sproull suggests using a truncated distance computation to increase efficiency in high dimensions [36]. Variants of the *k*-d tree have been used to address specific search problems [2], [33].

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 [18]. This hierarchical rectangular structure is useful in applications such as searching by image content [30] 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 [34], R+-trees [35], and R*-trees [4]. 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 [16], quad-trees [17], vp-trees [39], and hB-trees [22], 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 [10]. 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 if $d > 3$.

3 The Algorithm

3.1 Searching by Slicing

We illustrate the proposed high dimensional search algorithm using a simple example in 3D space, shown in Fig. 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 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 Fig. 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 on these points. If there are no points inside the cube, we know that there are no points within ϵ.

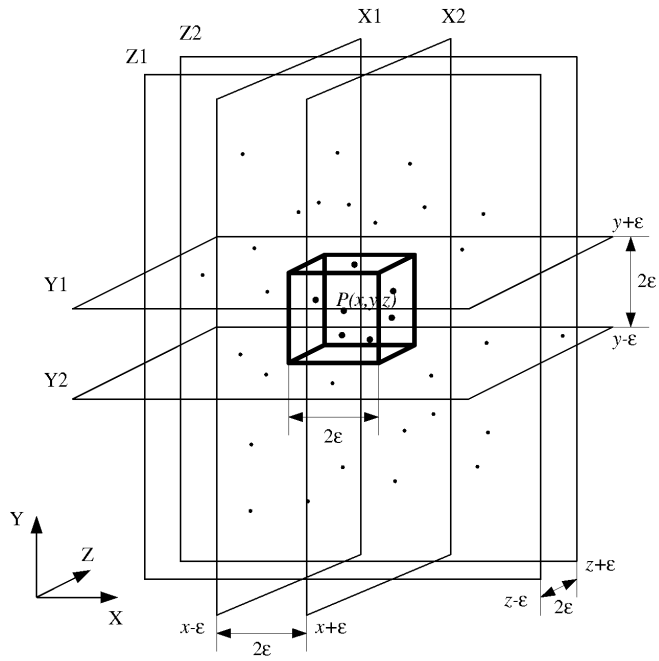


Fig. 1. The proposed algorithm efficiently finds points inside a cube of size 2ϵ around the novel query point $\boldsymbol{Q}$. The closest point is then found by performing an exhaustive search within the cube using the Euclidean distance metric.

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 Fig. 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 $\boldsymbol{Q}$.

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.

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 1D binary searches [1] 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 Fig. 2. It is assumed that the point set is static and hence, for a given

1. Although this appears contradictory to the previous statement, the claim of $O(\log_2 n)$ complexity is made assuming fixed d and varying n [7], [8], [15]. The exact relationship between d and complexity has not yet been established, but it has been observed by us and many others that it is roughly exponential.

point set, the data structure needs to be constructed only once. The point set is stored as a collection of d 1D arrays, where the jth array contains the jth 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 Fig. 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 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.

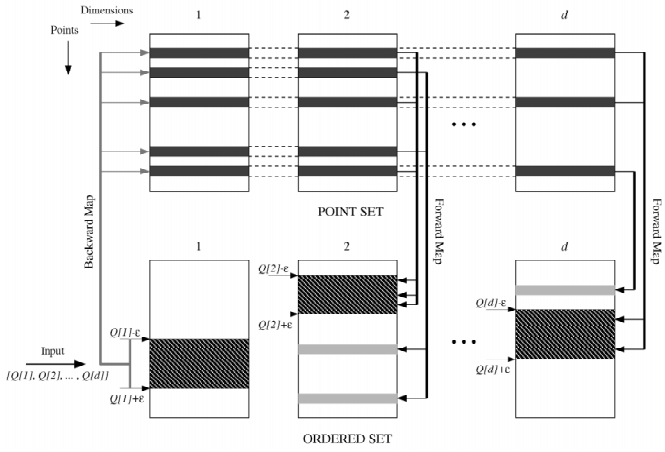


Fig. 2. Data structures used for constructing and trimming the candidate list. The point set corresponds to the raw list of data points, while in the ordered set each coordinate is sorted. The forward and backward maps enable efficient correspondence between the point and ordered sets.

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 $P[i][B[i][j]] = O[i][j]$.

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 $k = 2, 3, \ldots, d$, as follows. In iteration k, we check every point in the candidate list, by using the forward map, to see if its kth coordinate lies within the limits $Q_k - \epsilon$ and $Q_k + \epsilon$. Each of these limits is also obtained by binary search. Points with kth coordinates that lie outside this range (shown in light gray) 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, …, 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.

4 Complexity

In this section, we attempt to analyze the computational complexity of data structure storage, construction, and nearest neighbor search. As we saw in the previous section, constructing the data structure is essentially sorting d arrays of size n. is can be done in $O(dn\log_2 n)$ time. The only additional storage necessary is to hold the forward and backward maps. This requires space $O(nd)$. For nearest neighbor search, 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 e, 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 kth 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.

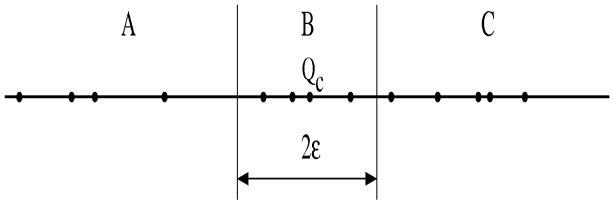


Fig. 3. The projection of the point set and the novel point onto one of the dimensions of the search space. The number of points inside bin B is given by the binomial distribution.

Fig. 3 shows the novel point $\boldsymbol{Q}$ and a set of n points in 2D 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 cth dimension. This corresponds to the points inside bin B in Fig. 3, where the entire point set and $\boldsymbol{Q}$ are projected to the cth 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 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,

$$P_c = P\{-\epsilon \le Z_c \le \epsilon \mid Q_c\} \tag{1}$$

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[2]:

$$P\{M_c = k \mid Q_c\} = P_c^k (1 - P_c)^{n-k} \binom{n}{k} \tag{2}$$

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

$$\begin{aligned} E[M_c \mid Q_c] &= \sum_{k=0}^{n} kP\{M_c = k \mid Q_c\} \\ &= nP_c \end{aligned} \tag{3}$$

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, (3) 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_kth hyperplane pair. How many points on the candidate list lie between this hyperplane pair? Once again, (3) 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,

$$\begin{aligned} N_k &= P_{c_k} N_{k-1}, \qquad N_0 = n \\ &= n \prod_{i=1}^{k} P_{c_i} \end{aligned} \tag{4}$$

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 (4) as

$$\begin{aligned} N &= N_1 + 3N_1 + 3N_2 + \ldots + 3N_{d-1} \\ &= N_1 + 3\sum_{k=1}^{d-1} N_k \\ &= n\left(P_{c_1} + 3\sum_{k=1}^{d-1} \prod_{i=1}^{k} P_{c_i} \right) \end{aligned} \tag{5}$$

which on the average is

$$E[N \mid \boldsymbol{Q}] = nE\left[P_{c_1} + 3\sum_{k=1}^{d-1} \prod_{i=1}^{k} P_{c_i} \right] \tag{6}$$

Equation (6) 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:

- $\boldsymbol{Z}$ is uniformly distributed, and
- $\boldsymbol{Z}$ is normally distributed.

Note that we have left out the cost of exhaustive search on points within the final hypercube. The reason is that the cost of an exhaustive search is dependent on the distance metric used. This cost is however very small and can be neglected in most cases when $n \gg d$. If it needs to be considered, it can be added to (6).

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, (5) can be rewritten as

$$N = n\left(P_{c_1} + 3\left(P_{c_1}\left(1 + P_{c_2}\left(1 + P_{c_3}\left(1 + \ldots\right)\right)\right)\right)\right) \tag{7}$$

It is immediate that the value of N is minimum when

$$P_{c_1} < P_{c_2} < \ldots < P_{c_{d-1}}$$

In other words, $c_1, c_2, \ldots, c_d$ 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

2. This is equivalent to the elementary probability problem: given that a success (a point is within bin B) can occur with probability P_c, the number of successes that occur in n independent trials (points) is binomially distributed.

there are only d such numbers, which can be obtained in time $O(d)$ by simply taking the difference of the indices to the ordered set returned by each pair of binary searches. Further, the cost of sorting these numbers is $O(d\log_2 d)$ by heap sort [1]. Clearly, both these costs are 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 its coordinates as

$$f_{X_c}(x) = \begin{cases} 1/l & \text{if } -l/2 \le x \le l/2 \\ 0 & \text{otherwise} \end{cases}, \forall c \tag{8}$$

Using (8), and the fact that $Z_c = X_c - Q_c$, an expression for the density of Z_c can be written as

$$f_{Z_c|Q_c}(z) = \begin{cases} 1/l & \text{if } -l/2 - Q_c \le z \le l/2 - Q_c \\ 0 & \text{otherwise} \end{cases}, \forall c \tag{9}$$

P_c can now be written as

$$\begin{aligned} P_c &= P\{-\epsilon \le Z_c \le \epsilon | Q_c\} = \int_{-\epsilon}^{\epsilon} f_{Z_c|Q_c}(z)dz \\ &\le \int_{-\epsilon}^{\epsilon} \frac{1}{l} dz \\ &\le \frac{2\epsilon}{l} \end{aligned} \tag{10}$$

Substituting (10) in (6), and considering the upper bound (worst case), we get

$$\begin{aligned} E[N] &= n\left(\frac{2\epsilon}{l} + 3\left(\frac{2\epsilon}{l} + \left(\frac{2\epsilon}{l}\right)^2 + \ldots + \left(\frac{2\epsilon}{l}\right)^{d-1}\right)\right) \\ &= n\left(\frac{2\epsilon}{l} + 3\left(\frac{1 - \left(\frac{2\epsilon}{l}\right)^d}{1 - \frac{2\epsilon}{l}} - 1\right)\right) \end{aligned} \tag{11}$$

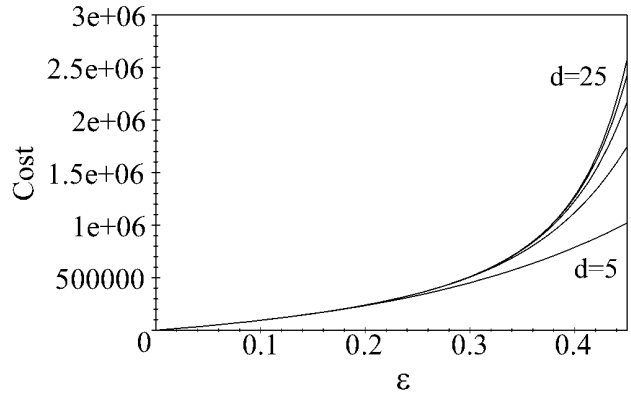
By neglecting constants, we write

$$E[N] = O\left(n\epsilon + n\frac{1-\epsilon^d}{1-\epsilon}\right) \tag{12}$$

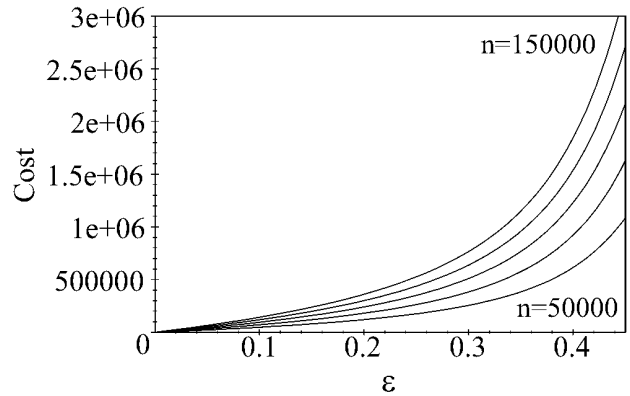
For small ϵ, we observe that $\epsilon^d \approx 0$, because of which cost is independent of d:

$$E[N] \approx O\left(n\epsilon + n\frac{1}{1-\epsilon}\right) \tag{13}$$

In Fig. 4, (11) is plotted against ϵ for different d (Fig. 4a) and different n (Fig. 4b) for $l = 1$. Observe that as long as $\epsilon < .25$, the cost varies little with d, and is linearly proportional to n. This also means that keeping ϵ small is crucial to the performance of the algorithm. As we shall see later, ϵ can, in fact, be kept small for many problems. Hence, even though the cost of our algorithm grows linearly with n, ϵ is small enough that in many real problems, it is better to pay this price of linearity, rather than an exponential dependence on d.



(a)



(b)

Fig. 4. The average cost of the algorithm is independent of d and grows only linearly for small ϵ. The point set in both cases is assumed to be uniformly distributed with extent $l = 1$. (a) The point set contains 100,000 points in 5D, 10D, 15D, 20D, and 25D spaces. (b) The point set is 15D and contains 50,000, 75,000, 100,000, 125,000, and 150,000 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 its coordinates:

$$f_{X_c}(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\frac{-x^2}{2\sigma^2} \tag{14}$$

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

$$f_{Z_c|Q_c}(z) = \frac{1}{\sqrt{2\pi}\sigma} \exp\frac{-(z-Q_c)^2}{2\sigma^2} \tag{15}$$

P_c can then be written as

$$\begin{aligned} P_c &= P\{-\epsilon \le Z_c \le \epsilon | Q_c\} = \int_{-\epsilon}^{\epsilon} f_{Z_c|Q_c}(z)\,dz \\ &= \frac{1}{2}\left(\operatorname{erf}\frac{\epsilon - Q_c}{\sigma\sqrt{2}} + \operatorname{erf}\frac{\epsilon + Q_c}{\sigma\sqrt{2}}\right) \end{aligned} \tag{16}$$

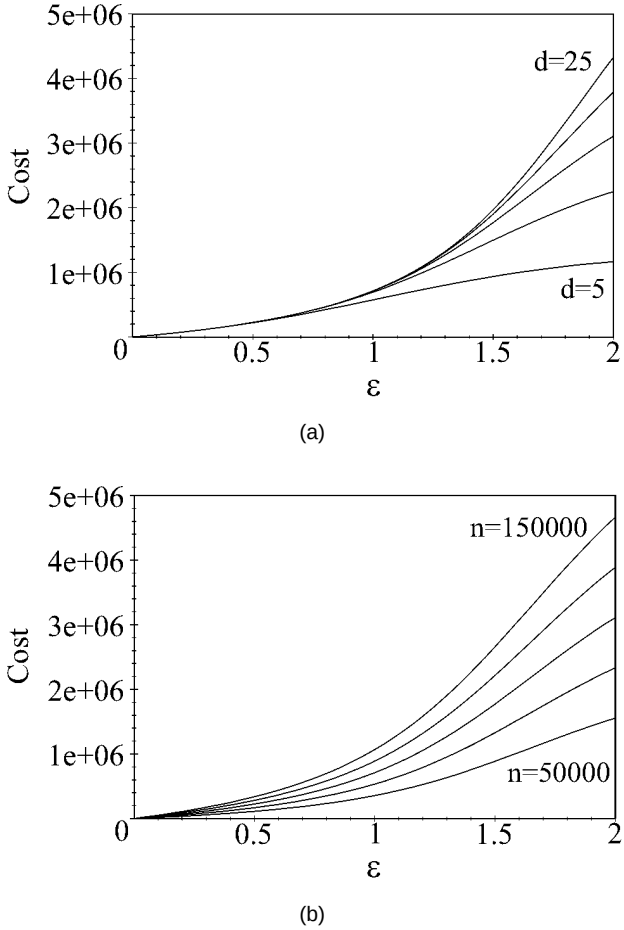


Fig. 5. The average cost of the algorithm is independent of d and grows only linearly for small ϵ. The point set in both cases is assumed to be normally distributed with variance $s = 1$. (a) The point set contains 100,000 points in 5D, 10D, 15D, 20D, and 25D spaces ($\boldsymbol{Q} = \boldsymbol{0}$). (b) The point set is 15D and contains 50,000, 75,000, 100,000, 125,000, and 150,000 points ($\boldsymbol{Q} = \boldsymbol{0}$).

This expression can be substituted into (6) and evaluated numerically to estimate cost for a given $\boldsymbol{Q}$. Fig. 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.

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. Fig. 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

$$L_p(\boldsymbol{a}, \boldsymbol{b}) = \left[\sum_k |a_k - b_k|^p \right]^{1/p} \tag{17}$$

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 Fig. 7a, this does not always correctly find the 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 Fig. 7b). 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.

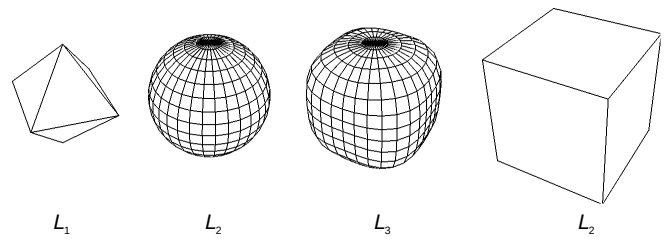


Fig. 6. An illustration of various norms, also known as Minkowski p-metrics. All points on these surfaces are equidistant from the central point. The L_∞ metric bounds L_p for all p.

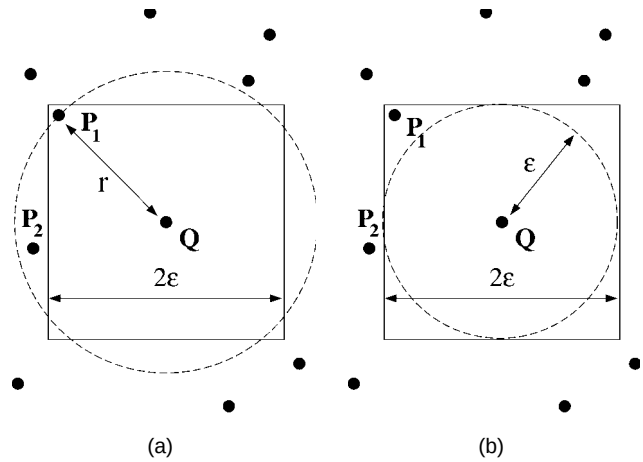


Fig. 7. An exhaustive search within a hypercube may yield an incorrect result. (a) $\boldsymbol{P_2}$ is closer to $\boldsymbol{Q}$ than $\boldsymbol{P_1}$, but just an exhaustive search within the cube will incorrectly identify $\boldsymbol{P_1}$ as the closest point. (b) This can be remedied by imposing 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 2ϵ).

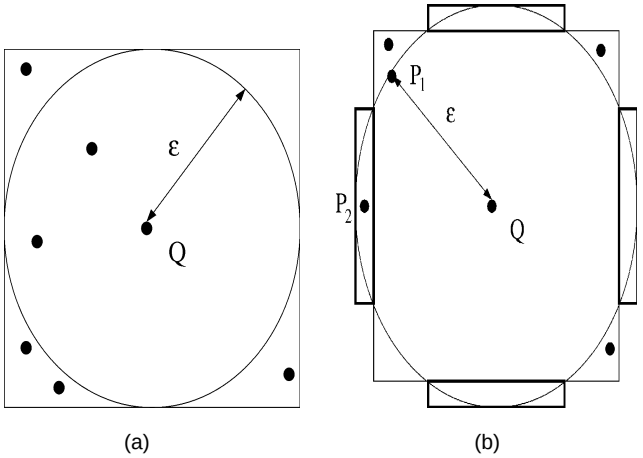


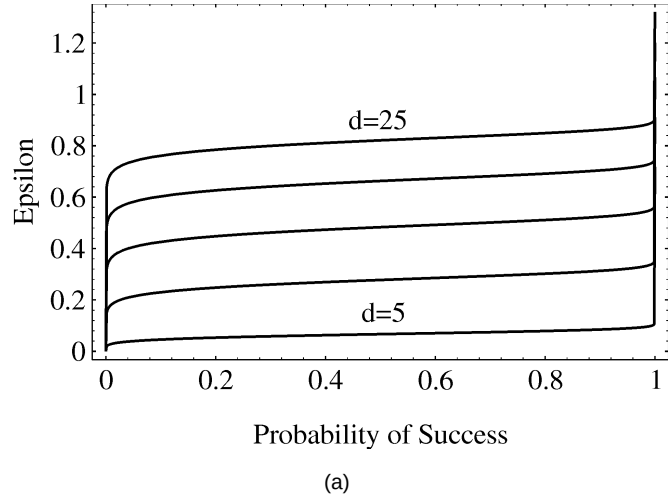
Fig. 8. ϵ can be computed using two methods. (a) By finding the radius of the smallest hypersphere that will contain at least one point with high probability. A search is performed by setting ϵ to this radius and constraining the exhaustive search within ϵ. (b) By finding the size of the smallest hypercube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

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 2ϵ. 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 half 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.

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 L_2 distance between $\boldsymbol{Q}$ and any point in the point set. Once again, M is binomially distributed with the density

$$P\{M = k|\boldsymbol{Q}\} = \left(P\{\|\boldsymbol{Z}\| \le \epsilon_{hs}|\boldsymbol{Q}\}\right)^k \left(1 - P\{\|\boldsymbol{Z}\| \le \epsilon_{hs}|\boldsymbol{Q}\}\right)^{n-k}\binom{n}{k} \quad (18)$$



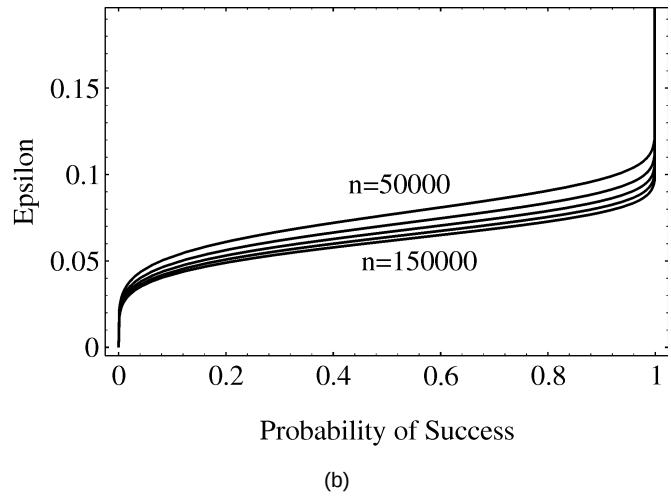


Fig. 9. The radius ϵ necessary to find a point inside a hypersphere varies very little with probability. This means that ϵ can be set to the knee where probability is close to unity. The point set in both cases is uniformly distributed with extent $l = 1$. (a) The point set contains 100,000 points in 5, 10, 15, 20, and 25 dimensional space. (b) The point is 5D and contains 50,000, 75,000, 100,000, 125,000, and 150,000 points.

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

$$p = P\{M > 0|\boldsymbol{Q}\} = 1 - P\{M = 0|\boldsymbol{Q}\} = 1 - \left(1 - P\{\|\boldsymbol{Z}\| \le \epsilon_{hs}|\boldsymbol{Q}\}\right)^n \quad (19)$$

The above equation suggests that if we know $\boldsymbol{Q}$, the density $f_{\boldsymbol{Z}|\boldsymbol{Q}}(z)$, and the probability p, we can solve for ϵ_{hs}.

For example, consider the case when the point set is uniformly distributed with density given by (9). The cumulative distribution function of $\|\boldsymbol{Z}\|$ is the uniform distribution integrated within a hypersphere; which is simply its volume. Thus,

$$P\{\|\boldsymbol{Z}\| \le \epsilon_{hs}|\boldsymbol{Q}\} = \frac{2\epsilon_{hs}{}^d \pi^{d/2}}{l^d d\Gamma(d/2)} \quad (20)$$

Substituting the above in (19) and solving for ϵ_{hs}, we get

$$\epsilon_{hs} = \left(\frac{l^d d\Gamma(d/2)}{2\pi^{d/2}} \left(1 - (1-p)^{1/n}\right) \right)^{1/d} \quad (21)$$

Using (21), ϵ_{hs} is plotted against probability for two cases. In Fig. 9a, d is fixed to different values between five and 25 with n fixed to 100,000, and in Fig. 9b, n is fixed to different values between 50,000 to 150,000 with d fixed to five. 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 of a side of the circumscribing hypercube to $2\epsilon_{hs}$ and by imposing an additional constraint during exhaustive search that only points within an L_2 distance ϵ_{hs} 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 cth coordinate of a point set point and the novel point $\boldsymbol{Q}$. Once again, M is binomially distributed with the density

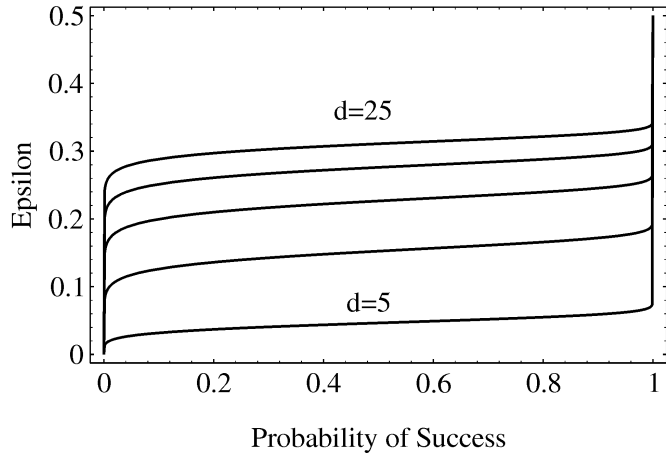
$$P\{M = k|\boldsymbol{Q}\} = \left(\prod_{c=1}^{d} P\{-\epsilon_{hc} \le Z_c \le \epsilon_{hc}|Q_c\} \right)^k \left(1 - \prod_{c=1}^{d} P\{-\epsilon_{hc} \le Z_c \le \epsilon_{hc}|Q_c\} \right)^{n-k} \binom{n}{k} \quad (22)$$

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

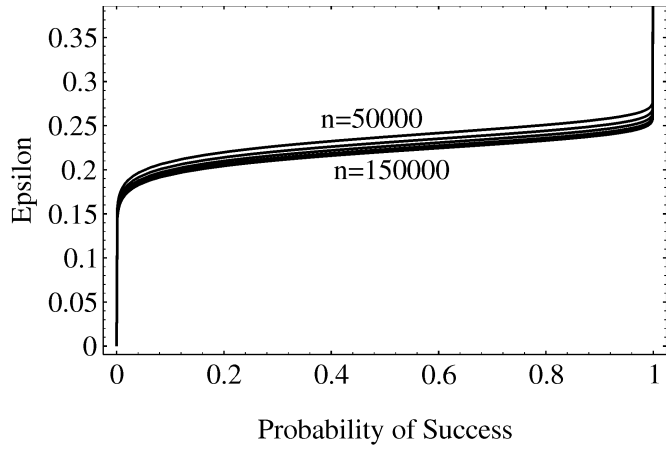
$$\begin{aligned} p &= P\{M > 0|\boldsymbol{Q}\} \\ &= 1 - P\{M = 0|\boldsymbol{Q}\} \\ &= 1 - \left(1 - \prod_{c=1}^{d} P\{-\epsilon_{hc} \le Z_c \le \epsilon_{hc}|Q_c\} \right)^n \end{aligned} \quad (23)$$

Again, the 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 for the specific case that 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 (9). Using (10) we get,

$$\prod_{c=1}^{d} P\{-\epsilon_{hc} \le Z_c \le \epsilon_{hc}|Q_c\} = \left(\frac{2\epsilon_{hc}}{l} \right)^d \quad (24)$$



(a)



(b)

Fig. 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. The point set in both cases is uniformly distributed with extent $l = 1$. (a) The point set contains 100,000 points in 5, 10, 15, 20, and 25 dimensional space. (b) The point set is 5D and contains 50,000, 75,000, 100,000, 125,000, and 150,000 points.

Substituting the above in (23) and solving for ϵ_{hc}, we get

$$\epsilon_{hc} = \frac{l}{2} \left(1 - (1-p)^{1/n}\right)^{1/d} \quad (25)$$

Using (25), ϵ_{hc} is plotted against probability for two cases. In Fig. 10a, d is fixed to different values between five and 25, with n is fixed at 100,000, and in Fig. 10b, n is fixed to different values between 50,000 and 150,000 with d fixed at five. 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 (smallest hypercube) 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 Fig. 8b, 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 Fig. 8b, $\boldsymbol{P_2}$ 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 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. It is, however, difficult to analytically predict which one of these methods suits a particular class of data. Hence, we encourage the reader to implement both the methods and use the one which performs the best. 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. We looked at two representative classes of search problems that may benefit from the algorithm.

- In the first class, the data has statistical structure. This is the case, for instance, when points are uniformly or normally distributed.
- The second class of problems are statistically unstructured, for instance, when points lie on a high dimensional multivariate manifold, and it is difficult to say anything about their distribution.

In this section, we will present results for benchmarks performed on statistically structured data. For benchmarks on statistically unstructured data, we refer the reader to Section 7.

We tested two commonly occurring distributions, normal and uniform. The proposed algorithm was compared with the *k*-d tree and exhaustive search algorithms. Other algorithms were not included in this benchmark because they did not yield comparable performance. For the first set of benchmarks, two normally distributed point sets containing 30,000 and 100,000 points with variance 1.0 were used. To test the per search execution time, another set of points, which we shall call the test set, was constructed. The test set contained 10,000 points, also normally distributed with variance 1.0. For each algorithm, the execution time was calculated by averaging the total time required to perform a nearest neighbor search on each of the 10,000 points in the test set. To determine ϵ, we used the "smallest hypercube" method described in Section 5.2. Since the point set is normally distributed, we cannot use a closed form solution for ϵ. However, it can be numerically computed as follows. Substituting (16) into (23), we get

$$p = 1 - \left(1 - \prod_{c=1}^{d} \frac{1}{2}\left(\operatorname{erf}\frac{\epsilon - Q_c}{\sigma\sqrt{2}} + \operatorname{erf}\frac{\epsilon + Q_c}{\sigma\sqrt{2}}\right)\right)^n \qquad (26)$$

By setting p (the probability that there is at least one point in the hypercube) to .99 and σ (the variance) to 1.0, we computed ϵ for each search point $\boldsymbol{Q}$ using the fast and simple bisection technique [32].

Figs. 11a and 11b show the average execution time per search when the point set contains 30,000 and 100,000 points respectively. These execution times include the time taken for search, computation of ϵ using (26), and the time taken for the few (1 percent) additional[3] searches necessary when a point was not found within the hypercube. Although ϵ varies for each $\boldsymbol{Q}$, values of ϵ for a few sample points are as follows.

- For n = 30,000, the values of ϵ at the point $\boldsymbol{Q}$ = (0, 0, ...) were ϵ = 0.22, 0.54, 0.76, 0.92, and 1.04, corresponding to d = 5, 10, 15, 20, and 25, respectively. At the point $\boldsymbol{Q}$ = (0.5, 0.5, ...), the values of ϵ were ϵ = 0.24, 0.61, 0.86, 1.04, and 1.17, corresponding to d = 5, 10, 15, 20, and 25, respectively.
- For n = 100,000, the values of ϵ at the point $\boldsymbol{Q}$ = (0, 0, ...) were ϵ = 0.17, 0.48, 0.69, 0.85, and 0.97, corresponding to d = 5, 10, 15, 20, and 25, respectively. At the point $\boldsymbol{Q}$ = (0.5, 0.5, ...), the values of ϵ were ϵ = 0.19, 0.54, 0.78, 0.96, and 1.09, corresponding to d = 5, 10, 15, 20, and 25, respectively.

Observe that the proposed algorithm is faster than the *k*-d tree algorithm for all d in Fig. 11a. In Fig. 11b, the proposed algorithm is faster for $d > 12$. Also notice that the *k*-d tree algorithm actually runs slower than exhaustive search for $d > 15$. The reason for this observation is as follows. In high dimensions, the space is so sparsely populated that the radius of the query hypersphere is very large. Consequently, 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 it very inefficient to search the sparse high dimensional space.

For the second set of benchmarks, we used uniformly distributed point sets containing 30,000 and 100,000 points with extent 1.0. The test set contained 10,000 points, also uniformly distributed with extent 1.0. The execution time per search was calculated by averaging the total time required to perform a closest point search on each of the 10,000 points in the test set. As before, to determine ϵ, the

3. When a point was not found within the hypercube, we incremented ϵ by 0.1 and searched again. This process was repeated till a point was found.

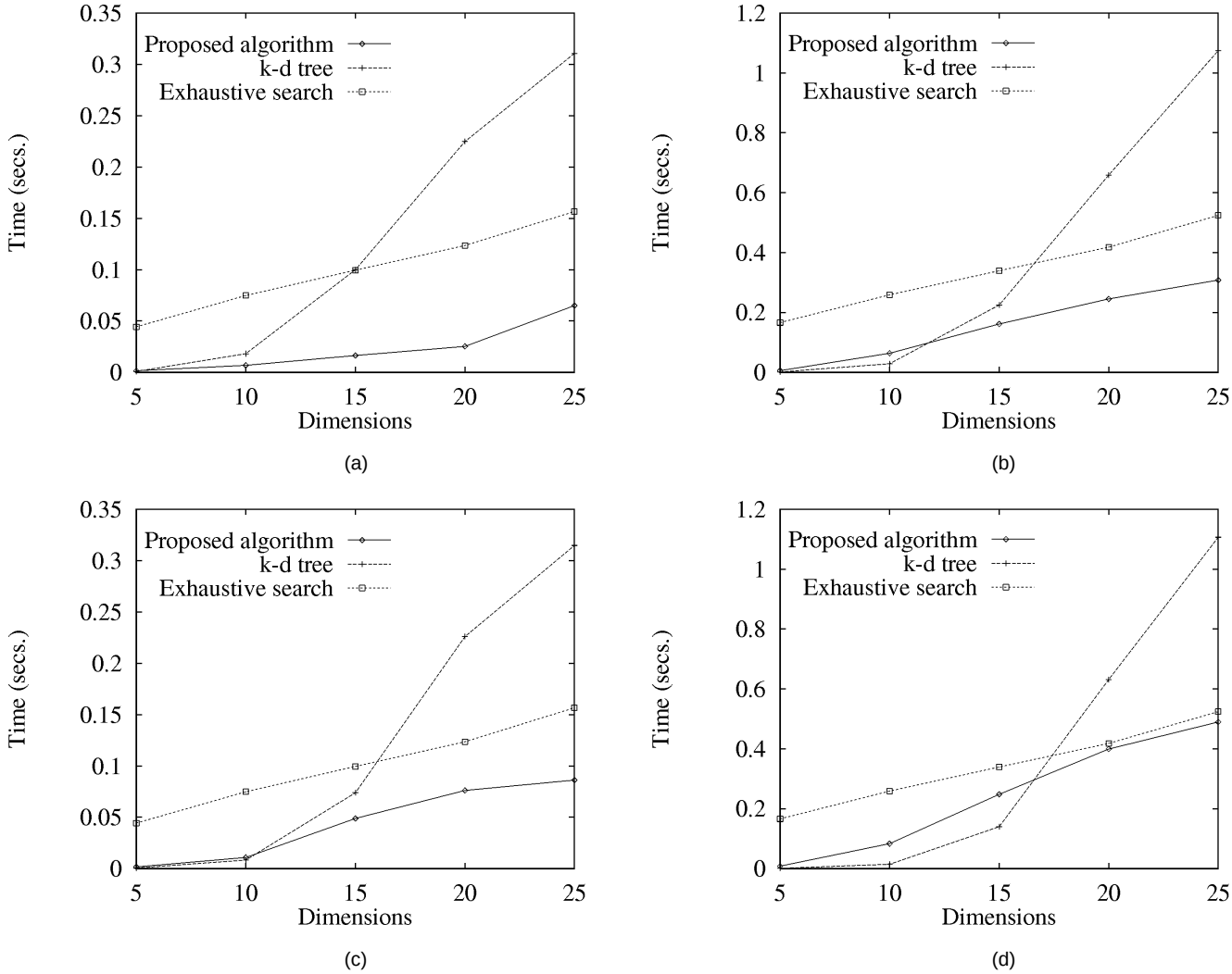


Fig. 11. The average execution time of the proposed algorithm is benchmarked for statistically structured problems. (a) The point set is normally distributed with variance 1.0 and contains 30,000 points. (b) The point set is normally distributed with variance 1.0 and contains 100,000 points. The proposed algorithm is clearly faster in high *d*. (c) The point set is uniformly distributed with extent 1.0 and contains 30,000 points. (d) The point set is uniformly distributed with extent 1.0 and contains 100,000 points. The proposed algorithm does not perform as well for uniform distributions due to the extreme sparseness of the point set in high *d*.

"smallest hypercube" method, described in Section 5.2 was used. Recall, that for uniformly distributed point sets, ϵ can be computed in the closed form using (25). Figs. 11c and 11d show execution times when the point set contains 30,000 and 100,000 points, respectively.

- For n = 30,000, the values of ϵ were ϵ = 0.09, 0.21, 0.28, 0.32, and 0.35, corresponding to d = 5, 10, 15, 20, and 25, respectively.
- For n = 100,000, the values of ϵ were ϵ = 0.07, 0.18, 0.26, 0.30, and 0.34, corresponding to d = 5, 10, 15, 20, and 25, respectively.

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 ends up examining almost all points, thereby approaching exhaustive search.

7 An Example Application: Appearance Matching

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 35D space. In the second, the closest point is required to be found from points lying on a multivariate high dimensional manifold. Both these problems are examples of statistically unstructured data.

Let us briefly review the object recognition technique of Murase and Nayar [24]. Object recognition is performed in two phases:

- appearance learning phase, and
- appearance recognition phase.

In the learning phase, images of each of the hundred objects in all poses are captured. These images are used to compute a high dimensional subspace, called the eigenspace. The

images are projected to eigenspace to obtain discrete high dimensional points. A smooth curve is then interpolated through points that belong to the same object. In this way, for each object, we get a curve (or a univariate manifold) parameterized by its pose. Once we have the manifolds, the second phase, object recognition, is easy. An image of an object is projected to eigenspace to obtain a single point. The manifold closest to this point identifies the object. The closest point on the manifold identifies the pose. Note that the manifold is continuous, so in order to find the closest point on the manifold, we need to finely sample it to obtain discrete closely spaced points.

For our benchmark, we used the Columbia Object Image Library [27] along with the SLAM software package [28] to compute 100 univariate manifolds in a 35D eigenspace. These manifolds correspond to appearance models of the 100 objects (20 of the 100 objects shown in Fig. 12a). Each of the 100 manifolds were sampled at 360 equally spaced points to obtain 36,000 discrete points in 35D space. It was impossible to manually capture the large number of object images that would be needed for a large test set. Hence, we automatically generated a test set of 100,000 points by sampling the manifolds at random locations. This is roughly equivalent to capturing actual images, but, without image sensor noise, lens blurring, and perspective projection effects. It is important to simulate these effects because they cause the projected point to shift away from the manifold and hence, substantially affect the performance of nearest neighbor search algorithms[4].

Unfortunately, it is very difficult to relate image noise, perspective projection, and other distortion effects to the location of points in eigenspace. Hence, we used a simple model where we add uniformly distributed noise with extent[5] .01 to each of the coordinates of points in the test set. We found that this approximates real-world data. We determined that setting $\epsilon = 0.1$ gave us good recognition accuracy. Fig. 12b shows the time taken per search by the different algorithms. The search time was calculated by averaging the total time taken to perform 100,000 closest point searches using points in the test set. It can be seen that the proposed algorithm outperforms all the other techniques. ϵ was set to a predetermined value such that a point was found within the hypersphere all the time. For object recognition, it is useful to search for the closest point within ϵ because this provides us with a means to reject points that are "far" from the manifold (most likely from objects not in the database).

Next, we examine another case when data is statistically unstructured. Here, the closest point is required to be found from points lying on a single smooth multivariate high dimensional manifold. Such a manifold appears frequently in appearance matching problems such as visual tracking [26], visual inspection [26], and parametric feature detection [25]. As with object recognition, the manifold is a representation of visual appearance. Given a novel appearance (point),

(a)

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

(b)

Fig. 12 The proposed algorithm was used to recognize and estimate pose of 100 objects using the Columbia Object Image Library. (a) Twenty of the 100 objects are shown. The point set consisted of 36,000 points (360 for each object) in 35D eigenspace. (b) The average execution time per search is compared with other algorithms.

matching involves finding a point on the manifold closest to that point. Given that the manifold is continuous, to pose appearance matching as a nearest neighbor problem, as before, we sample the manifold densely to obtain discrete closely spaced points.

The trivariate manifold we used in our benchmarks was obtained from a visual tracking experiment conducted by Nayar et al. [26]. In the first benchmark, the manifold was sampled to obtain 31,752 discrete points. In the second benchmark, it was sampled to obtain 107,163 points. In both cases, a test set of 10,000 randomly sampled manifold points was used. As explained previously, noise (with extent .01) was added to each coordinate in the test set. The execution time per search was averaged over this test set of 10,000 points. For this point set, it was determined that $\epsilon = 0.07$ gave good recognition accuracy. Fig. 13a shows the algorithm to be more than two orders of magnitude faster than the other algorithms. Notice the exponential behavior of the R-tree algorithm. Also notice that Yunck's algorithm is only slightly faster than Friedman's; the difference is due to use of integer operations. We could only benchmark Yunck's algorithm till $d = 30$ due to use of a 32-bit word in the indicator array. In Fig. 13b, it can be seen that the proposed algorithm is faster than the *k*-d tree for all *d*, while in Fig. 13c, the proposed algorithm is faster for all $d > 21$.

4. For instance, in the *k*-d tree, a large query hypersphere would result in a large increase in the number of adjacent buckets that may have to be searched.

5. The extent of the eigenspace is from –1.0 to +1.0. The maximum noise amplitude is hence about 0.5 percent of the extent of eigenspace.

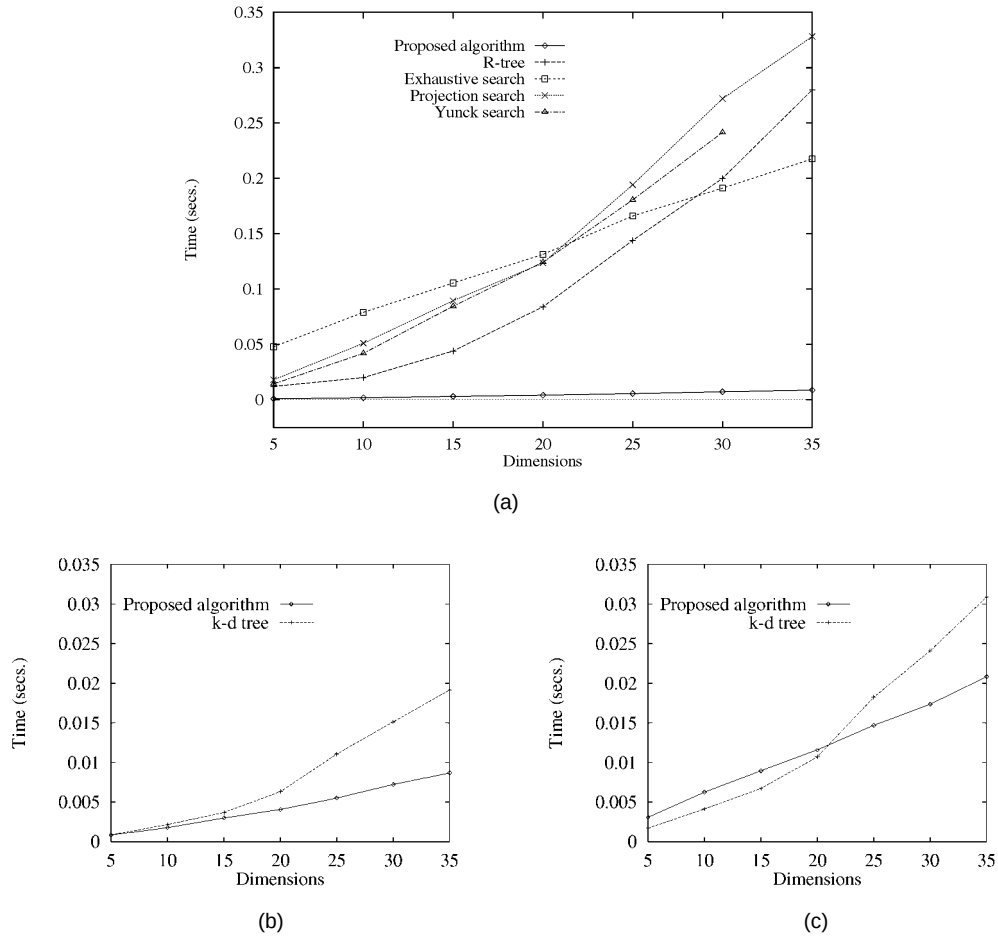


Fig. 13. The average execution time of the proposed algorithm is benchmarked for an unstructured problem. The point set is constructed by sampling a high dimensional trivariate manifold. (a) The manifold is sampled to obtain 31,752 points. The proposed algorithm is more than two orders of magnitude faster 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.

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 Fig. 14. 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, otherwise 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 or PC. We estimate the module to result in a 100 fold speedup over an optimized software implementation.

9 Discussion

9.1 *k* Nearest Neighbor Search

In Section 5, we saw that it is possible to determine the minimum value of ϵ necessary to ensure that at least one point is found within a hypercube or hypersphere with high probability. It is possible to extend this notion to ensure that at least *k* points are found with high certainty.

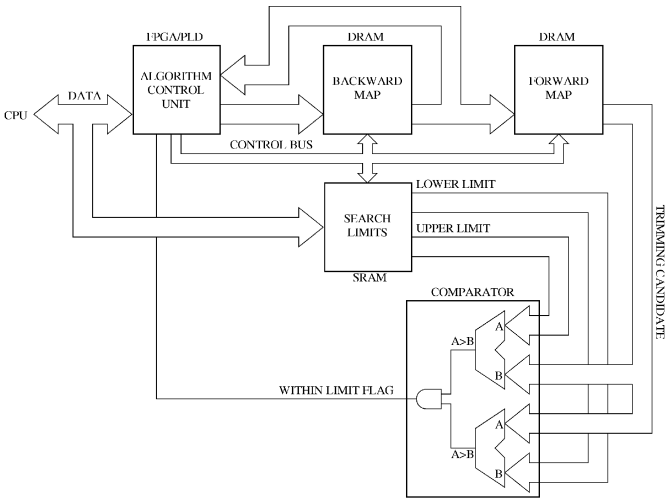


Fig. 14. Architecture for an inexpensive hardware search engine that is based on the proposed algorithm.

Recall that the probability that there exists at least one point in a hypersphere of radius ϵ is given by (19). Now define p_k to be the probability that there are at least k points within the hypersphere. We can then write p_k as

$$\begin{aligned} p_k &= P\{M \geq k|\mathbf{Q}\} \\ &= 1 - \left(P\{M = 0|\mathbf{Q}\} + P\{M = 1|\mathbf{Q}\} + \ldots + P\{M = k-1|\mathbf{Q}\}\right) \\ &= 1 - \sum_{i=0}^{k-1} P\{M = i|\mathbf{Q}\} \end{aligned} \quad (27)$$

The above expression can now be substituted in (18) and given p_k, numerically solved for ϵ_{hs}. Similarly, it can be substituted in (22) to compute the minimum value of ϵ_{hc} for a hypercube.

9.2 Dynamic Point Insertion and Deletion

Currently, the algorithm uses d floating point arrays to store the ordered set, and $2d$ integer arrays to store the backward and forward maps. As a result, it is not possible to efficiently insert or delete points in the search space. This limitation can be easily overcome if the ordered set is not stored as an array but as a set of d binary search trees (BST) (each BST corresponds to an array of the ordered set). Similarly, the d forward maps have to be replaced with a single linked list. The backward maps can be done away with completely as the indices can be made to reside within a node of the BST. Although BSTs would allow efficient insertion and deletion, nearest neighbor searches would no longer be as efficient as with integer arrays. Also, in order to get maximum efficiency, the BSTs would have to be well balanced (see [19] for a discussion on balancing techniques).

9.3 Searching With Partial Data

Many times, it is required to search for the nearest neighbor in the absence of complete data. For instance, consider an application which requires features to be extracted from an image and then matched against other features in a feature space. Now, if it is not possible to extract all features, then the matching has to be done partially. It is trivial to adapt our algorithm to such a situation: while trimming the list, you need to only look at the dimensions for which you have data. This is hard to envision in the case of k-d trees for example, because the space has been partitioned by hyperplanes in *particular* dimensions. So, when traversing the tree to locate the bucket that contains the query point, it is not possible to choose a traversal direction at a node if data corresponding to the partitioning dimension at that node is missing from the query point.

ACKNOWLEDGMENTS

We wish to thank Simon Baker and Dinkar Bhat for their detailed comments, criticisms and suggestions that have helped greatly in improving 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 National Science Foundation National Young Investigator Award.

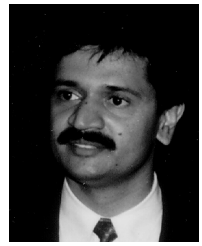
REFERENCES

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
[2] S. Arya, "Nearest Neighbor Searching and Applications," no. CS-TR-3490, Univ. of Maryland, June 1995.
[3] F. Aurenhammer, "Voronoi Diagrams—A Survey of a Fundamental Geometric Data Structure," *ACM Computing Surveys*, vol. 23, no. 3, pp. 345-405, Sept. 1991.
[4] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD*, pp. 322-331, Atlantic City, NJ, May 1990.
[5] J.L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Comm. ACM*, vol. 18, no. 9, pp. 509-517, Sept. 1975.
[6] J.L. Bentley, "Multidimensional Binary Search Trees in Database Applications," *IEEE Trans. Software Engineering*, vol. 5, no. 4, pp. 333-340, July 1979.
[7] J.L. Bentley and B.W. Weide, "Optimal Expected-Time Algorithms for Closest Point Problems," *ACM Trans. Mathematical Software*, vol. 6, no. 4, pp. 563-580, Dec. 1980.
[8] J.L. Bentley, "Multidimensional Divide-and-Conquer, *Comm. ACM*, vol. 23, no. 4, pp. 214-229, Apr. 1980.
[9] A. Califano and R. Mohan, "Multidimensional Indexing for Recognizing Visual Shapes," *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, pp. 28-34, June 1991.
[10] K. L. Clarkson, "A Randomized Algorithm for Closest-Point Queries," *SIAM J. Computing*, vol. 17, no. 4, pp. 830-847, Aug. 1988.
[11] D. Dobkin and R.J. Lipton, "Multidimensional Searching Problems," *SIAM J. Computing*, vol. 5, no. 2, pp. 181-186, June 1976.
[12] H. Edelsbrunner, *Algorithms in Combinatorial Geometry*. Berlin-Heidelberg: Springer, 1987.
[13] A. Farago, T. Linder, and G. Lubosi, "Fast Nearest-Neighbor Search in Dissimilarity Spaces," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 15, no. 9, pp. 957-962, Sept. 1993.
[14] J.H. Friedman, F. Baskett, and L.J. Shustek, "An Algorithm for Finding Nearest Neighbors," *IEEE Trans. Computers*, pp. 1,000-1,006, Oct. 1975.
[15] J.H. Friedman, J.L. Bentley, and R.A. Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time," *ACM Trans. Mathematical Software*, vol. 3, no. 3, pp. 209-226, Sept. 1977.
[16] K. Fukunaga and P. M. Narendra, "A Branch and Bound Algorithm for Computing k-Nearest Neighbors," *IEEE Trans. Computers*, pp. 750-753, July 1975.
[17] I. Gargantini, "An Effective Way to Represent Quadtrees," *Comm. ACM*, vol. 25, no. 12, pp. 905-910, Dec. 1982.
[18] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD*, pp. 47-57, June 1984.

[19] E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, 2nd ed. Rockville, Md.: Computer Science Press, 1987.
[20] V. Klee, "On the Complexity of d-Dimensional Voronoi Diagrams," *Arch. Math*, vol. 34, pp. 75-80, 1980.
[21] D.E. Knuth, "Sorting and Searching," *The Art of Computer Programming*, vol. 3. Reading, Mass.: Addison-Wesley, 1973.
[22] D.B. Lomet and B. Salzberg, "The hb-Tree: A Multiattribute Indexing Method With Good Guaranteed Performance," *Proc. ACM TODS*, vol. 15, no. 4, pp. 625-658, Dec. 1990.
[23] M.L. Mico, J. Oncina, and E. Vidal, "A New Version of the Nearest-Neighbor Approximating and Eliminating Search Algorithm (AESA) With Linear Preprocessing Time and Memory Requirements," *Pattern Recognition Letters*, no. 15, pp. 9-17, 1994.
[24] H. Murase and S.K. Nayar, "Visual Learning and Recognition of 3D Objects From Appearance," *Int'l J. Computer Vision*, vol. 14, no. 1, pp. 5-24, Jan. 1995.
[25] S.K. Nayar, S. Baker, and H. Murase, "Parametric Feature Detection," *Proc. IEEE CS Conf. Computer Vision and Pattern Recognition* (*CVPR*), pp. 471-477, San Francisco, Calif., June 1996.
[26] S.K. Nayar, H. Murase, and S.A. Nene, "Learning, Positioning, and Tracking Visual Appearance," *Proc. IEEE Int'l Conf. Robotics and Automation*, San Diego, Calif., May 1994.
[27] S.K. Nayar, S.A. Nene, and H. Murase, "Real-Time 100 Object Recognition System," *Proc. IEEE Int'l Conf. Robotics and Automation*, Twin Cities, May 1996.
[28] S.A. Nene and S.K. Nayar, "SLAM: A Software Library for Appearance Matching," *Proc. ARPA Image Understanding Workshop*, Monterey, Calif., Nov. 1994. Also Technical Report CUCS-019-94.
[29] A.N. Netravali, *Digital Pictures: Representation, Compression, and Standards*, 2nd ed. New York: Plenum Press, 1995.
[30] E.G.M. Petrakis and C. Faloutsos, "Similarity Searching in Large Image Databases," Technical Report CS-TR-3388, Dept. Computer Science, Univ. of Maryland, Dec. 1994.
[31] F.P. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*. New York: Springer, 1985.
[32] W.H. Press, S. A. Teukolsky, W.T. Vetterling, and B.P. Flannery, *Numerical Recipes in C*, 2nd ed. Cambridge Univ. Press, 1992.
[33] T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proc. ACM SIGMOD*, pp. 10-18, 1981.
[34] N. Roussopoulos and D. Leifker, "Direct Spatial Search on Pictorial Databases Using Packed R-Trees," *Proc. ACM SIGMOD*, May 1985.
[35] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-Tree: A Dynamic Index for Multidimensional Objects, *Proc. 13th Int'l Conf. VLDB*, pp. 507-518, Sept. 1987.
[36] R.F. Sproull, "Refinements to Nearest-Neighbor Searching in k-Dimensional Trees," *Algorithmica*, vol. 6, pp. 579-589, 1991.
[37] J.M. Vilar, "Reducing the Overhead of the AESA Metric-Space Nearest Neighbour Searching Algorithm," *Information Processing Letters*, 1996.
[38] H. Wolfson, "Model-Based Object Recognition by Geometric Hashing," *Proc. First European Conf. Comp. Vision*, pp. 526-536, Apr. 1990.
[39] P.N. Yianilos, "Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces," *Proc. ACM-SIAM Symp. Discrete Algorithms*, pp. 311-321, 1993.
[40] T.P. Yunck, "A Technique to Identify Nearest Neighbors," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 6, no. 10, pp. 678-683, Oct. 1976.

Sameer A. Nene received the BE degree in computer engineering from the University of Pune, Pune, India, in 1992 and the MS degree in electrical engineering from Columbia University, New York, in 1994. He is currently pursuing the PhD in electrical engineering at Columbia University, New York.

His research interests include appearance matching, nearest neighbor search and its applications to computer vision, image processing and computer graphics, uncalibrated stereo, stereo using mirrors, image-based rendering, and high-performance computer graphics.



Shree K. Nayar is a professor at the Department of Computer Science, Columbia University. He received his PhD degree in electrical and computer engineering from the Robotics Institute at Carnegie-Mellon University in 1990.

His primary research interests are in computational vision and robotics, with emphasis on physical models for early visual processing, sensors, and algorithms for shape recovery, pattern learning and recognition, vision-based manipulation and tracking, and the use of machine vision for computer graphics and virtual reality.

Dr. Nayar has authored and coauthored papers that have received the David Marr Prize at the 1995 International Conference on Computer Vision (ICCV'95) held in Boston, Mass., Siemens Outstanding Paper Award at the 1994 IEEE Computer Vision and Pattern Recognition Conference (CVPR'94) held in Seattle, 1994 Annual Pattern Recognition Award from the Pattern Recognition Society, Best Industry Related Paper Award at the 1994 International Conference on Pattern Recognition (ICPR'94) held in Jerusalem, and the David Marr Prize at the 1990 International Conference on Computer Vision (ICCV'90) held in Osaka. He holds several U.S. and international patents for inventions related to computer vision and robotics. Dr. Nayar was the recipient of the David and Lucile Packard Fellowship for Science and Engineering in 1992 and the National Young Investigator Award from the National Science Foundation in 1993.