Closest Point Search in High Dimensions *

Sameer A. Nene and Shree K. Nayar

Department of Computer Science, Columbia University
New York, NY 10027

Abstract

The problem of finding the closest point in high-dimensional spaces is common in computational vision. 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. Our algorithm uses a projection search technique along with a novel data structure to dramatically improve performance in high dimensions. A complexity analysis is presented which can help determine ϵ in structured problems. Benchmarks clearly show the superiority of the proposed algorithm for high dimensional search problems frequently encountered in machine vision, such as real-time object recognition.

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 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 [Nayar *et al.*-1996b], visual positioning, tracking and inspection [Nayar *et al.*-1994], feature detection [Nayar *et al.*-1996a], and face recognition [Turk and Pentland-1991]. 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.

*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 an NSF National Young Investigator Award.

In this paper, we propose a simple algorithm to efficiently search for the nearest neighbor within distance ϵ. We shall see that unlike existing techniques, 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 the applications mentioned above and for that matter, for most real world problems[1], because successful recognition generally requires the novel point to be sufficiently close to a database point. At times, it is not possible to assume that ϵ is known, so we suggest a method to automatically choose ϵ. A C language implementation of our algorithm is available by sending mail to the authors at *search@cs.columbia.edu*. A pseudo-code version of the algorithm can be found in [Nene and Nayar-1995].

2 Previous Work

Search algorithms can be broadly divided into the following categories: (a) exhaustive search, (b) hashing and indexing, (c) static space partitioning, (d) dynamic space partitioning, and (e) randomized algorithms. Our algorithm falls in category (d). A detailed description of search algorithms that fall in the above categories can be found in [Nene and Nayar-1995]. Our algorithm is based on the projection search paradigm first used by Friedman [Friedman *et al.*-1975]. Friedman's technique did not however perform well in high dimensions. Our algorithm, while still using the notion of projection search, dramatically improves high dimensional performance. A number of algorithms have been advanced in category (c), which include the widely used k-d tree [Bentley-1975] and R-tree [Guttman-1984] techniques. These algorithms construct a data structure which partitions space into bins that hold the database points. Given a novel point, a search is performed by first using the data structure to find the bin which contains the novel point. An exhaustive search is then performed within this bin (and if necessary, in the surrounding bins) to find the closest point. A detailed description of these techniques can again be found in [Nene and Nayar-1995].

[1] It is actually hard to think of a problem where one needs to find the closest point which is in general far away from the novel point.

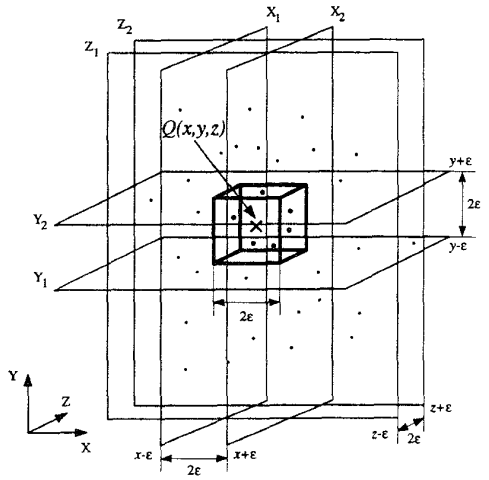


Figure 1: The proposed algorithm efficiently finds points inside a cube of size 2ϵ around the novel query point $\boldsymbol{Q}$. The closest point is then found by performing an exhaustive search within the cube using the Euclidean distance metric.

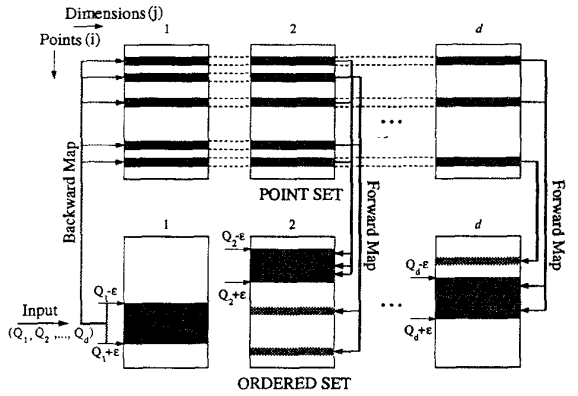


Figure 2: Data structures used for constructing and trimming the candidate list. The point set corresponds to the raw list of data points, while in the ordered set each coordinate is sorted. The forward and backward maps enable efficient correspondence between the point and ordered sets.

3 The Algorithm

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 a Euclidean sense and within a distance ϵ from a novel query point $\boldsymbol{Q}(x, y, z)$ (marked by a cross). 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 is found by performing an exhaustive search on these inside points using the Euclidean distance metric. 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 between the parallel pair of planes Y_1 and Y_2. 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 at $\boldsymbol{Q}$.

The data structure shown in figure 2 helps us to quickly find the sandwiched points. 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 $Q_1, Q_2, \cdots, Q_d$. 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 coordinates lie between the limits $Q_1 - \epsilon$ and $Q_1 + \epsilon$. This can be done with the help of two 1-D binary searches [Aho *et al.*-1974], 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 coordinate in the point set to a coordinate in the ordered set. Notice that the maps are simple integer arrays; if $P[d][n]$ is the point set, $O[d][n]$ the ordered set, $F[d][n]$ and $B[d][n]$ the forward and backward maps respectively, then $O[i][F[i][j]] = P[i][j]$ and $P[i][B[i][j]] = O[i][j]$.

As mentioned above, 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

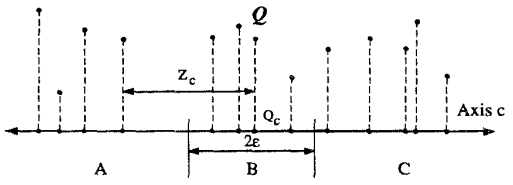


Figure 3: The projection of the point set and the novel point onto one of the dimensions of the search space.

list by iterating through $k = 2, 3, \cdots, d$, as follows. In 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 $\boldsymbol{Q}$.

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 $\boldsymbol{Q}$. For the purpose of 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 and vectors are in bold. 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,

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

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

$$E[M_c \mid Q_c] = nP_c. \tag{1}$$

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. For this, 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

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

which, on the average is

$$E[N \mid \boldsymbol{Q}] = nE\left[P_{c_1} + 3\sum_{k=1}^{d-1}\prod_{i=1}^{k} P_{c_i}\right]. \tag{4}$$

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, which is $N_d d$. This is very small and can be neglected in most cases when $n \gg d$. If it needs to be considered, it is simply added to equation (4).

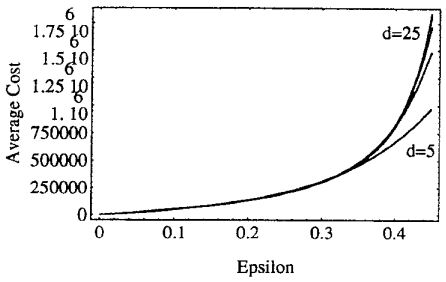


Figure 4: The average cost of the algorithm is independent of d and grows only linearly for small ϵ. Uniformly distributed point set containing 100,000 points in 5-D, 10-D, 15-D, 20-D and 25-D spaces.

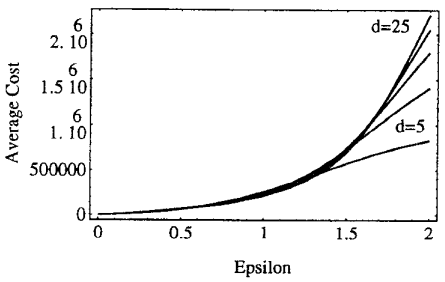


Figure 5: The average cost of the algorithm is independent of d and grows only linearly for small ϵ. Normally distributed point set containing 100,000 points in 5-D, 10-D, 15-D, 20-D and 25-D spaces ($\boldsymbol{Q} = \mathbf{0}$).

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 along each of its coordinates as

$$f_{X_c}(x) = \begin{cases} 1/l & \text{if } -l/2 \le x \le l/2, \\ 0 & \text{otherwise.} \end{cases} \quad (5)$$

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

$$f_{Z_c|Q_c}(z) = \begin{cases} 1/l & \text{if } -l/2 - Q_c \le z \le l/2 - Q_c, \\ 0 & \text{otherwise.} \end{cases} \quad (6)$$

P_c can now be written as

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

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

$$E[N] \le n\left(\frac{2\epsilon}{l} + 3\left(\frac{1-\left(\frac{2\epsilon}{l}\right)^d}{1-\frac{2\epsilon}{l}} - 1\right)\right). \quad (8)$$

If $\epsilon/l \ll 1$, 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 and $l = 1$. 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.

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 along 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

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

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 $\boldsymbol{Q} = \mathbf{0}$ 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 significant 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

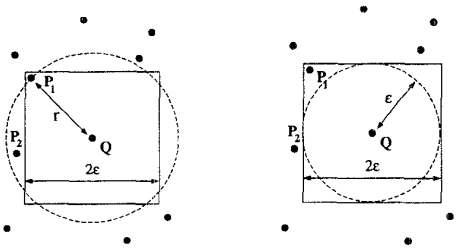


Figure 6: An exhaustive search within a hypercube may yield an incorrect result. (a) $\boldsymbol{P_2}$ is closer to $\boldsymbol{Q}$ than $\boldsymbol{P_1}$, but just an exhaustive search within the cube will incorrectly identify $\boldsymbol{P_1}$ as the closest point. (b) This can be remedied by imposing 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).

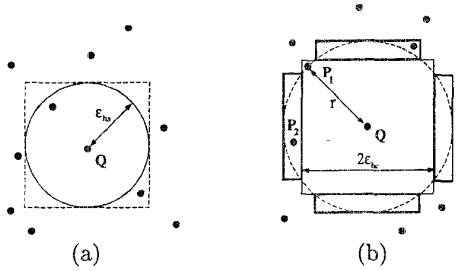


Figure 7: ϵ can be computed using two methods: (a) By finding the radius ϵ_{hs} of the smallest hypersphere that will contain at least one point with high probability. A search is performed by setting ϵ to ϵ_{hs} and constraining the exhaustive search within ϵ. (b) By finding the size ϵ_{hc} of the smallest hypercube that will contain at least one point with high probability. When searching, ϵ is set to ϵ_{hc}. Additional searches are performed in the areas marked in bold.

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.

The L_2 norm occurs most frequently in real world 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 L_2 search inside L_∞. However, as seen in figure 6(a), 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 figure 6(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 ϵ_{hs} of the smallest hypersphere that will contain *at least* one point with some (specified) probability p. ϵ is set to this radius ϵ_{hs}, and the algorithm proceeds to find all points within a *circumscribing* hypercube of side $2\epsilon_{hs}$. This is illustrated in figure 7(a). The following is an expression for ϵ_{hs} obtained from the results of section 4 for a uniformly distributed point set.

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

A detailed derivation of the above expression can be found in [Nene and Nayar-1995]. The method described above 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 ϵ_{hc} of the *smallest* hypercube that will contain *at least* one point with some (specified) probability p. ϵ can then be set to ϵ_{hc}. The following is an expression for ϵ_{hc} obtained from the results of section 4 when the point set is uniformly distributed.

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

A detailed derivation of the above expression can again be found in [Nene and Nayar-1995]. The above method, however, 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. This can be remedied by performing additional searches in the hyper-rectangular regions shown in bold in figure 7(b).

6 Benchmarks

We have performed an extensive set of benchmarks on the proposed algorithm. A C language implementation of the algorithm is available by sending mail to the authors at *search@cs.columbia.edu*. 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

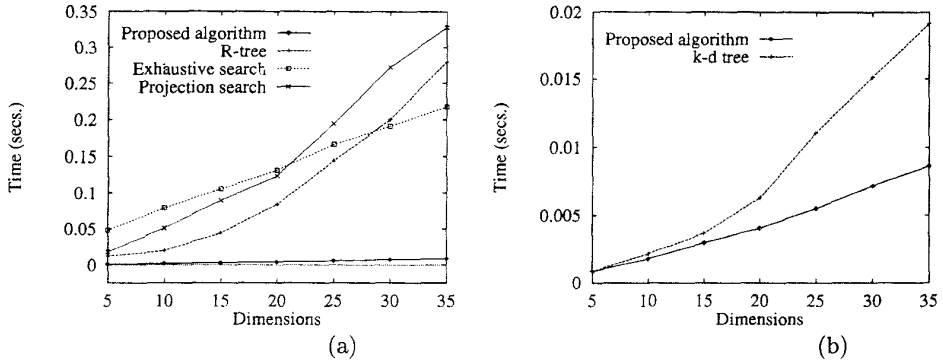


Figure 8: The average execution time of the proposed algorithm is benchmarked for an unstructured problem. The point set is constructed by sampling a high dimensional trivariate manifold to obtain 31,752 points. (a) The proposed algorithm is compared with the R-tree, exhaustive, and projection search techniques and found to be more than two orders of magnitude faster. (b) The proposed algorithm is compared with the k-d tree algorithm and found to perform significantly faster in high dimensions.

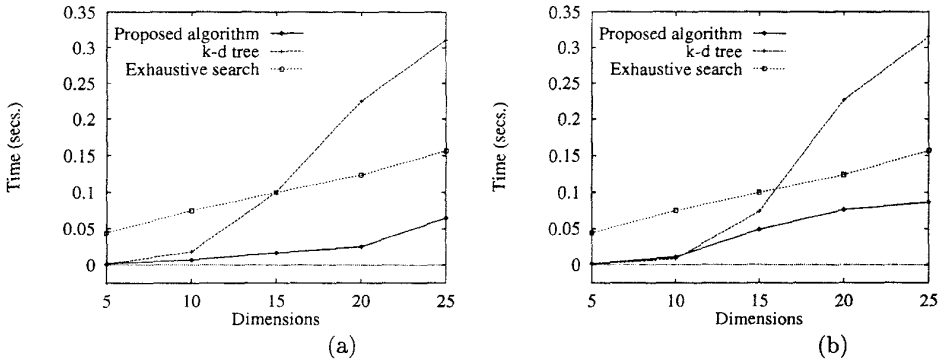
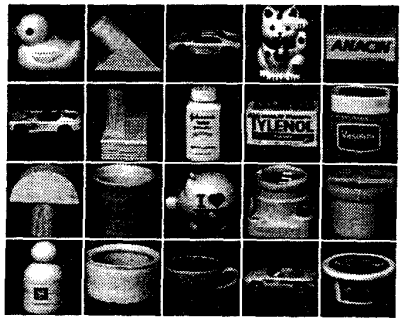


Figure 9: The average execution time of the proposed algorithm is benchmarked for structured problems. (a) Point set is normally distributed and contains 30,000 points with variance 1.0. (b) Point set is uniformly distributed and contains 30,000 points with extent 1.0. In both cases, the proposed algorithm performs better than the k-d tree and exhaustive search techniques.

first case, we used a high dimensional trivariate manifold with complex structure. This manifold was used in the appearance matching experiments conducted by Nayar *et al.* [Nayar *et al.*-1994]. In figure 8(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. Figure 8(b) compares the proposed algorithm with the k-d tree algorithm with d varying from 5 to 25. The proposed algorithm is significantly faster than the k-d tree algorithm in high dimensions.

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. Figure 9(a) shows 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. 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 9(a).



(a)

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

(b)

Figure 10: The proposed algorithm was used to recognize and estimate pose of hundred objects. (a) Twenty of the hundred objects from the Columbia Object Image Library [Nene *et al.*-1996] 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.

Figure 9(b) shows the execution times when the point set is uniformly distributed with unit extent ($l = 1$) and contains 30,000 points. 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. It can be seen that our algorithm performs better in high dimensions. The performance is, however, not as good as with the trivariate manifold because of the extreme sparseness of a uniform distribution of 30,000 points in high dimensional space. The k-d tree algorithm performs worse than exhaustive search because of the overhead associated with traversing the k-d tree to look for points in surrounding bins.

7 Real-time Object Recognition

We used the Columbia Object Image Library [Nene *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 [Murase and Nayar-1995] of the 100 objects shown in Figure 10(a). 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 10(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 for affine distortions in image projection and image sensor noise. It can be seen that the proposed algorithm outperforms all the other techniques.

Acknowledgements

We wish to thank Simon Baker and Dinkar Bhat of Columbia University for their detailed comments and suggestions which have helped to improve this paper.

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.

[Bentley, 1975] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[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.

[Guttman, 1984] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *ACM SIGMOD*, pages 47–57, June 1984.

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

[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.*, 1996a] S. K. Nayar, S. Baker and H. Murase. Parametric Feature Detection. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, June 1996. Also Technical Report CUCS-028-95.

[Nayar *et al.*, 1996b] 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*, Minneapolis, April 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 Technical Report CUCS-019-94.

[Nene and Nayar, 1995] S. A. Nene and S. K. Nayar. A Simple Algorithm for Nearest Neighbor Search in High Dimensions. Technical Report CUCS-030-95, Columbia University, October 1995.

[Nene *et al.*, 1996] S. A. Nene, S. K. Nayar and H. Murase. Columbia Object Image Library: COIL-100. Technical Report CUCS-006-96, Department of Computer Science, Columbia University, February 1996.

[Turk and Pentland, 1991] M. A. Turk and A. P. Pentland. Face Recognition Using Eigenfaces. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 586–591, June 1991.