A Fast Nearest Neighbor Algorithm Based on a Principal Axis Search Tree

James McNames, Member, IEEE

Abstract— **A new fast nearest neighbor algorithm is described that uses principal component analysis to build an efficient search tree. At each node in the tree, the data set is partitioned along the direction of maximum variance. The search algorithm efficiently uses a depth-first search and a new elimination criterion. The new algorithm was compared to sixteen other fast nearest neighbor algorithms on three types of common benchmark data sets including problems from time series prediction and image vector quantization. This comparative study illustrates the strengths and weaknesses of all of the leading algorithms. The new algorithm performed very well on all of the data sets and was consistently ranked among the top three algorithms.**

Keywords— **Nearest Neighbor, Vector Quantization Encoding, Principal Components Analysis, Closest Point, Intrinsic Dimension, Post Office Problem**

I. Introduction

GIVEN a data set of n_p points, $\{x_1, x_2, \ldots, x_{n_p}\}$, the k nearest neighbors problem is to find the k points that are closest to a query point, q, where $q, x_i \in \Re^{n_d}\ \forall i$. This problem is encountered in a wide range of applications including density estimation, pattern recognition [1], clustering [2], function approximation [3], time series prediction [4], document retrieval [5], optical character recognition [6], and vector quantization [2, 7]. In many of these applications, the computational cost of finding the nearest neighbors imposes practical limits on the data set size and the rate at which the application can operate. This has motivated the development of many fast nearest neighbor algorithms.

This article introduces a new algorithm based on principal axis trees called PAT, and compares it with other leading algorithms. Only algorithms that use the Euclidean metric,

$$D^2(x,q) \equiv \sum_{i=1}^{n_d} (x_i - q_i)^2,$$

to measure the distance between two points, are included in this study.

The performance of nearest neighbor algorithms is usually specified by the preprocessing time, memory required, and average query time. For most applications, a moderate amount of preprocessing and memory allocation are acceptable if the average query time is small enough.

The following section reviews some of the most popular and effective methods of reducing query time. Section III discusses how the dimension of the data set affects these methods. Section IV describes principal axis trees (PAT) in detail. Section V specifies the other algorithms included in a comparative empirical study, discusses how the query time was measured, and describes how user-specified parameters were chosen. Section VI reports the performance of PAT and the other algorithms on a variety of benchmark problems. PAT is shown to have excellent performance across a broad range of data sets.

Electrical & Computer Engineering, Portland State University, P.O. Box 751, Portland State University, Portland, OR 97207-0751. Email: mcnames@pdx.edu. Phone: 503.725.5390. Fax: 503.725.3807.

II. Elimination Criteria

Many nearest neighbor algorithms have been proposed to overcome the large computational cost of the obvious brute force approach. Typically, these algorithms employ one or more elimination criteria. Each estimates a lower bound on the distance between a query point and a point, or set of points, in the data set. If the lower bound is greater than the distance to the kth nearest neighbor found so far, the point can be eliminated without explicitly calculating the distance to that point. This section describes three of these elimination criteria that are included in principal axis trees (PAT).[1]

A. Partial Distance Search

Full-search improvements apply an elimination criterion to every point in the data set. Cheng *et al.* [7] originally proposed an algorithm called partial distance search (PDS) that is arguably the most popular full-search improvement. The algorithm consists of a simple modification of the brute force search: during the calculation of the distance, if the partial sum of square differences exceeds the distance to the nearest neighbor found so far, the calculation is aborted. Like the brute force search, PDS does not require any preprocessing or storage and the performance of PDS is almost always substantially better.

B. Search Trees

During preprocessing, search trees divide the data set into distinct subsets. Each subset is recursively subdivided until the number of points in each terminal node is acceptably small. The search for the nearest neighbors begins with the root node and works toward the terminal nodes. At each step, a lower bound on the minimum distance to each subset of points is calculated. If the lower bound is greater than the distance to the kth nearest neighbor found

[1] More thorough reviews of common elimination criteria are given in [8, 9].

so far, then the entire subset can be eliminated without calculating the distance to each point explicitly [10–14]. Because search trees are capable of eliminating entire groups of points, query times of $O(\log n_p)$ are frequently achieved, in low dimensions [11, 12].

C. Projection Methods

Projection methods reduce computation by projecting the data set and the query point into a linear subspace, which is usually spanned by the first few principal components [2, 5, 15, 16]. Section IV describes a new algorithm that uses the principal component projection to construct efficient search trees.

III. The Problem of Dimensionality

Many of the elimination criteria employed by fast nearest neighbor algorithms require $O(n_p \log n_p)$ preprocessing time, $O(n_p)$ storage space, and achieve $O(\log n_p)$ mean query time or better in low dimensions, where n_p is the number of points in the data set [11–13]. However, if the data set variables are independently distributed, the elimination criteria become less effective as the dimension increases and the average query times of the fastest algorithms scale exponentially with dimension, $O(c^{n_d})$ [9]. When the dimension becomes large enough, the elimination criteria become mostly ineffective and the distance is calculated for nearly every point in the data set. At this point, the average query time transitions from an exponential growth to linear growth, which is equivalent to a full-search algorithm. Moreover, in high dimensions "fast algorithms" can have a larger query time than full-search algorithms due to the computational overhead of the elimination criteria.

Fortunately, much better performance is typically observed in practice because real data sets are rarely composed of independently distributed variables. The intrinsic dimension, defined as the minimum number of parameters needed to account for the observed properties of a data set, is often much less than the spatial dimension [9]. Conceptually, this occurs when all of the points in a data set lie on a n_i-dimensional nonlinear surface, or manifold, in a n_d-dimensional space, where $n_i \ll n_d$.

The performance of fast algorithms on data sets with an intrinsic dimension that is significantly less than the spatial dimension is sensitive to the type of elimination criterion used [9]. The new algorithm described in the next section uses principal component analysis to adapt to the data set distribution and it performs well when the intrinsic dimension is less than the spatial dimension.

IV. Principal Axis Trees

This section describes a new method of using principal component analysis (PCA) to build an efficient search tree. Although PCA has been used in other nearest neighbor algorithms, it has not been combined with the efficient structure of a search tree or the elimination criteria described here. The combination of the search tree and PCA enables PAT to find the k nearest neighbors efficiently.

A. Search Tree Construction

Search tree construction begins by projecting the entire data set along the principal axis. Next, the data set is partitioned along the principal axis into n_c distinct regions such that each region contains roughly the same number of points. The process is repeated for each subset of points recursively until each subset contains fewer than n_c points. The algorithm for the search tree construction is described in Appendix B. This algorithm requires $O(n_p \log n_p)$ preprocessing and $O(n_p)$ storage.

Although the principal axis in the second step could be found by performing a complete principal components analysis, it is much faster to use the power method [18]. This technique has also been used in clustering algorithms and for codebook generation in vector quantization [17]. The implementation of the power method used in this study is described in [9], though a standard implementation would also work. A discussion of the method's convergence properties can be found in [18].

A two-dimensional example of the type of partition generated by this type of search tree is shown in Fig. 1. This illustrates the ability of PAT to efficiently partition the data set into regions that have approximately the same density locally.

B. Search Algorithm

The depth-first search process begins with the root node and uses a binary search to determine which region the query point is in. The child node that contains this region is then searched, and the process is repeated recursively until a terminal node is reached. Partial distance search (PDS) is used to calculate the distance to points in the terminal node [7]. The algorithm then moves to the parent of the terminal node. The elimination criterion (described below) is applied to the closest sibling node. If it is not satisfied, the sibling node is searched. If the criterion is satisfied, one or more sibling nodes are eliminated. The algorithm then works its way back up to the root node, either searching or eliminating sibling nodes along the way.

The elimination criterion is based on a lower bound of the distance to the points assigned to sibling nodes. If the lower bound is greater than the distance to the kth nearest neighbor found so far, none of the points assigned to the sibling is a neighbor and the node does not need to be searched. Other siblings can also be eliminated that have greater lower bounds.

The calculation of the lower bound is illustrated in Fig. 2. During the construction of the search tree, the points assigned to each node are subdivided by hyperplanes located along the principal axis. At the root node, the distance from the query point, q, to the boundary of a child node, b_2, is shown by the chord labeled d_{q2}. This distance is a lower bound on the distance to all of the points in the gray region. If this distance is greater than the distance to the kth nearest neighbor found so far, then all points in this region can be eliminated. Similarly, all of the regions that are further along the principal axis can be eliminated including, for example, Region 5.

If a region cannot be eliminated by the lower bound, a different lower bound on the distance to a point within the region can be calculated. For example, consider the point x in Region 2. Since the chord $\overline{qb_2}$ is a normal of the hyperplane separating Region 1 from Region 2, the angle $\angle qb_2x$ is no smaller than 90°. A lower bound on the distance between q and x is given by the law of cosines:

$$\begin{aligned} d_{qx}^2 &= d_{q2}^2 + d_{2x}^2 - 2d_{q2}d_{2x}\cos\angle qb_2x, \\ &\geq d_{q2}^2 + d_{2x}^2. \end{aligned} \quad (1)$$

This method can also be used to establish a lower bound for subregions of the gray region. For example, if the boundary point between Region 2 and Region 3 is calculated (shown as b_3 in Fig. 2), a lower bound on the distance from b_2 to the points in Region 3 can be calculated by projecting b_2 onto the principal axis of the gray region and calculating the distance from b_2 to b_3. Since d_{23}^2 is a lower bound on the distance from b_2 to any point in Region 3, Equation 1 can be used to find a lower bound on the distance from q to any point in Region 3:

$$\begin{array}{lcll} D^2(q,x) & \geq & d_{q2}^2 + d_{2x}^2 & \forall x \in \text{ Gray Region}, \\ d_{2x}^2 & \geq & d_{23}^2 & \forall x \in \text{ Region 3, and therefore}, \\ D^2(q,x) & \geq & d_{q2}^2 + d_{23}^2 & \forall x \in \text{ Region 3}. \end{array}$$

The same reasoning can be extended to other subregions. For example, a lower bound on the distance from q to all points in Region 4 is given by

$$D^2(q,x) \geq d_{q2}^2 + d_{23}^2 + d_{34}^2 \quad \forall x \in \text{ Region 4}.$$

The boundary points, b, and distance lower bounds, d_{LB}^2, are calculated recursively in the search algorithm. The search algorithm is described in detail in Appendix C.

V. Experimental Design

This section lists the other algorithms included in this study, describes how the user-specified parameters were chosen, and describes how the mean query time was estimated.

A. Algorithms

Section IV described the new algorithm PAT. Table I lists the other sixteen algorithms included in this study. All of the algorithms find the exact nearest neighbors of a query point. Most of these algorithms also calculate the distance to the nearest neighbors in the process, but for this study the algorithms were only required to find the indices of the neighboring points.

Two of the algorithms, K-D Tree [11] and OPS [12], were modified for this study such that they incorporate the partial distance search [7] (PDS) in the terminal nodes, instead of the originally proposed brute force search (Brute). This significantly improves performance. A modification to OPS described in [19] was also implemented to guarantee the exact nearest neighbors are found.

B. User-Specified Parameters

User-specified parameters enable the user to tune the performance of an algorithm to a particular data set. However, this process requires additional computation, analysis, and user expertise that is usually impractical. Instead, users often pick a value that is recommended by the authors of the algorithm or simply pick a value that has worked well in the past.

Some of the nearest neighbor algorithms included in this study have a single user-specified parameter. For example, PAT, FN, and FNM all require the user to specify the number of partitions of each non-terminal node, K-D Trees require that the user specify the number of points contained in each terminal node, and Bak and TLAESA require the user to specify the number of anchor points for the triangle-inequality. It is possible to generalize some of the other algorithms to include similar parameters.

User-specified parameters make it difficult to compare the performance of algorithms fairly. For all of the results reported here, the user-specified parameters were fixed at a constant value. This ensured that the algorithms with user-specified parameters could not be tuned to a particular data set, giving them an unfair advantage over algorithms that could not be likewise optimized.

Since a broad range of data sets were included in this study, the user-specified parameters were selected to ensure good overall performance. Specifically, the values were based on the algorithms' average query time on twenty-four simulated data sets. Twelve of the data sets were drawn from a uniform distribution with one to twelve dimensions. Twelve other data sets were drawn from a normal distribution over the same range of dimensions. Each data set contained $2,000$ points. The parameter values were selected from a range of two to fifty.

In most cases, the best parameter value varied only slightly with dimension and distribution. For example, Fig. 3 shows the average query time for PAT for various dimensions and parameter values. For PAT, the best parameter values were in the range of five to eight. In this case, the parameter value was fixed at seven. Table II lists the values chosen for each of the algorithms that had a user-specified parameter. These values were selected prior to the study described in Section VI.

C. Measures of Performance

The goal of this study was to compare the average query times of the leading algorithms, the most critical consideration for many applications. However, there is no consensus on how to measure query time and researchers have used a wide variety of different measures in previous studies [9].

For this study, the average query time was measured directly. The disadvantage of this approach is that the results depend on the characteristics of the hardware and software[2], but, for lack of a better measure, it is assumed

[2]The results reported here were generated using a Pentium 200 MHz MMX processor with 512 KB Cache, 128 MB RAM, Windows NT 4.0 SP5, and Visual C++ 6.0 SP1.

that similar results would be obtained with other implementations and systems.

VI. Empirical Performance

This section reports the algorithms' empirical performance on several benchmark data sets. The accuracy of each algorithm was confirmed by comparing the nearest neighbor indices with those found by Brute. All of the algorithms found the same neighbors as Brute except in cases where different data set points were equidistant to the query point. A more thorough discussion and a more detailed description of the data sets[3] and results are given in [9].

A. Uniform Distribution

The average query times were calculated for uniformly distributed data sets with dimensions ranging from one to twenty. Each data set contained $5,000$ points and the average query times were calculated using $5,000$ queries. For each dimension, the algorithms were ranked in order of increasing average query times. Fig. 4 shows a plot of the average query time versus dimension for the top five algorithms. An exponential growth in low dimensions and the linear growth in high dimensions is apparent, as discussed in Section III.

For this type of data set, PDS performed best in high dimensions because the elimination criteria of the other algorithms are ineffective in high dimensions and have more overhead than PDS. In low dimensions, algorithms that used a search tree performed better than PDS.

According to an average ranking, PAT and DOSVQ were tied for best performance. However, as shown in Fig. 4, DOSVQ performed better in high dimensions due to a smaller overhead and PAT performed better in low dimensions.

The average query times were also measured for various data set sizes. Each data set contained eight variables and the average query times were calculated using $5,000$ queries. For each data set size, the algorithms were ranked in order of increasing average query times. Fig. 5 shows the average query time versus the data set size for the best five algorithms. In this case the average query time of PAT scaled as $O(\log n_p)$, whereas DOSVQ and scaled as $O(n_p)$. This is an important advantage of PAT. Although the slower algorithms may also eventually scale as $O(\log n_p)$ for larger data sets or lower dimensions, it is significant that PAT achieves logarithmic query time more readily.

B. Normal Distribution

The analysis performed using a uniform distribution was repeated using a normal distribution. Fig. 6 shows a plot of the average query time versus dimension for the top five algorithms.

[3]The time series and vector quantization data sets were available at *http://www.ece.pdx.edu/∼mcnames/DataSets* at the time of writing.

The results are very similar to those obtained using a uniform distribution, which suggests that the relative performance of the algorithms is insensitive to the marginal distributions of data set variables when they are independently distributed.

As with the uniform distribution, PAT had the best overall performance in low and moderate dimensional spaces; other search trees also performed well. PDS had the best performance in high dimensions, though other methods with small overhead had comparable performance.

Fig. 7 shows the average query times of the best five algorithms for various data set sizes. The averages were calculated using eight dimensional data sets and $5,000$ queries for each data set. As with the uniform distribution, PAT achieved logarithmic query time, $O(\log n_p)$. The average query times of DOSVQ scaled roughly linearly with the data set size, $O(n_p)$.

C. Chaotic Time Series

Local models are commonly used for time series prediction [9]. These models use a data set that is constructed from overlapping segments of the time series. A simple model is constructed for every query using only a portion of the points neighboring the input vector. The neighbors are usually found by a fast nearest neighbor algorithm. The models are often constructed to predict one step ahead using a window of previous values as the input. For example, if the time series is represented by $[y_1, y_2, \ldots, y_n]$, an input at time t is given by

$$x_t \equiv [\, y_t, y_{t-1}, y_{t-2}, \ldots, y_{t-(n_d-1)} \,].$$

Usually these models are applied to time series generated by nonlinear dynamic systems. If the systems have no external inputs and they meet other very general criteria, the time series are chaotic and all of the points in the data set lie on a low dimensional manifold [9]. Consequently, the intrinsic dimension of these data sets does not change as the spatial dimension, n_d, is increased.

The Lorenz time series is a common benchmark for measuring the accuracy of time series prediction algorithms [4]. Using this time series, the average query times were calculated for various dimensions using a data set containing $5,000$ points and averaging over $5,000$ queries. Fig. 8 shows the average query times of the top five algorithms.

In moderate to high dimensional spaces, DOSVQ had the best performance, though PAT also performed very well. Elimination criteria that are insensitive to the spatial dimension of the data set performed much better on this problem than on the independently distributed data sets. For example, algorithms such as DWCS, TLAESA and FN, which rely on the triangle inequality, improved significantly in rank. The query time of K-D Tree grew much more quickly with dimension than the top algorithms because it is less able to adapt to the data set distribution, especially when the intrinsic dimension is much smaller than the spatial dimension.

This analysis was repeated using the Santa Fe laser time series [20], another common time series prediction bench-

mark. The average query times were measured for various dimensions using a data set containing $5,000$ points and averaging over $5,000$ queries. Fig. 9 shows the average query times of the top five algorithms. In this case, PAT performed slightly better than DOSVQ.

D. Vector Quantization

Vector quantization is a popular and powerful method of compression. It begins with the construction of a codebook, which is a sample data set of the signals to be compressed. The codebook is typically constructed by applying a clustering algorithm to a large representative sample of the signals to be transmitted. Once the codebook is created, the signal to be compressed is divided into blocks, or vectors, of a user-specified length. The nearest neighbor in a codebook is found for each vector and the index of the nearest neighbor is used to encode, or represent, the vector. In a communications application, the sender and the receiver are both given a copy of the codebook. Only the indices of the nearest neighbors are transmitted. Once the receiver receives an index, the corresponding vector in the codebook is retrieved and used to represent the original vector. This technique is commonly used to compress speech and images. One of the practical limitations of this method is the large computational cost of finding the nearest neighbor in the codebook.

To test the performance of the nearest neighbor algorithms in vector quantization encoding applications, a range of different codebooks were constructed for a speech signal and three gray-scale images that are commonly used as benchmarks for vector quantization algorithms. The codebooks were constructed using the clustering algorithm described by Franti *et al.* [17].

Table III shows the average query times, calculated using $35,000$ queries, to find the nearest neighbor for the speech signal. Tables IV, V, and VI show the average query times for the benchmark images *Lena*, *Boat*, and *Baboon*[4], respectively. For the images, the average query times were calculated using $20,000$ queries for the 2×2 image blocks and $15,000$ queries for the 4×4 image blocks. In all cases, the query vectors were taken from the original signals used to construct the codebook. The results are similar to those obtained for chaotic time series: PAT and DOSVQ were statistically tied for the best performance.

VII. Conclusions

This work introduced a new nearest neighbor algorithm based on principal axis trees called PAT. The performance of this algorithm was compared to sixteen other of the leading fast nearest neighbor algorithms on a variety of benchmark problems, including independently-distributed simulated data sets, chaotic time series, and vector quantization codebooks.

Some of the algorithms used in this study had a single user-specified parameter. These parameters were fixed at the value that optimized the overall performance on independently-distributed data sets ranging from one to twelve dimensions. Better performance may be achievable on other data sets if different user-specified parameter values are used.

The algorithms would certainly have performed better if general purpose improvements had been used. For example, in many applications nearest neighbor algorithms are applied to a set of query points that are highly correlated. Many researchers have independently proposed using the nearest neighbors of the previous query as initial candidates for the nearest neighbors of the current query [1, 21]. This type of improvement can be applied to any nearest neighbor algorithm, but it was not used here.

The algorithms were applied to three different types of problems and many different data sets. One of the important conclusions from this study is that the performance of nearest neighbor algorithms varies substantially with the properties of the data set. Thus, when the properties of a data set is known, such as the intrinsic dimension, this should be taken into account when choosing an algorithm. If the properties are not known, an algorithm with good overall performance, such as PAT, should be chosen.

The algorithms that performed best in this study incorporated search trees and partial distance search (PDS). It was surprising to find that PDS performed best on independently-distributed data sets in high dimensions. Since the elimination criteria of all algorithms become ineffective in high dimensional spaces, the good performance of PDS in this case is due to the minimal overhead of its elimination criteria.

One of the most popular algorithms included in this study, K-D Trees, shares some similarities with PAT and also had good overall performance in this study. Both algorithms construct balanced search trees and use elimination criteria to eliminate entire subtrees during a recursive search. There are also important differences between the two algorithms. PAT requires the user to specify the number of branches at each node whereas K-D Tree uses a binary tree, which is generally narrower and deeper. PAT partitions the search space into hyper-polygons whereas K-D Tree partitions the space into hyper-rectangles. The elimination criteria for the algorithms is very different due to the different geometry of the partitions. Although both algorithms used PDS to search terminal nodes for this study, the original description of K-D Tree used Brute [11].

Overall, PAT and DOSVQ [14] had the best overall performance. In many cases, the average query times of PAT and DOSVQ were statistically indistinguishable. In other cases, the difference was significant. For example, DOSVQ had better performance on independently-distributed data sets in high dimensions, mostly due to the smaller overhead of its elimination criteria. PAT had superior performance on large independently-distributed data sets and scaled as $O(\log n_p)$ as opposed to $O(n_p)$ for DOSVQ, where n_p is the number of points in the data set. PAT also requires less memory, $O(n_p)$, than DOSVQ, $O(n_p \log n_p)$. Consequently, PAT has a significant advantage over DOSVQ on large data sets.

[4]This image is sometimes called *Mandrill*.

PAT was designed to combine the best features of the fastest algorithms, such as search trees and PDS, with a new method of partitioning the data set along the directions of maximum variance and a new elimination criterion. These new features enable PAT to construct a partition of the data set that can be searched efficiently. This type of partitioning, as illustrated by Fig. 1, helps PAT perform especially well when the spatial dimension of the data set is significantly higher than the intrinsic dimension.

Acknowledgments

The author is grateful for thorough and thoughtful suggestions of the anonymous reviewers and the editorial staff.

References

[1] S. H. Chen and J. S. Pan, "Fast search algorithm for VQ-based recognition of isolated words," *IEE Proceedings I (Communications, Speech and Vision)*, vol. 136, no. 6, pp. 391–396, Dec. 1989.

[2] Tung-Shou Chen and Chin-Chen Chang, "Diagonal axes method (DAM): A fast search algorithm for vector quantization," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 7, no. 3, pp. 555–559, June 1997.

[3] Christopher G. Atkeson, Andrew W. Moore, and Stefan Schaal, "Locally weighted learning," *Artificial Intelligence Review*, vol. 11, no. 1–5, pp. 11–73, Feb. 1997.

[4] James McNames, "A nearest trajectory strategy for time series prediction," in *Proceedings of the International Workshop on Advanced Black-Box Techniques for Nonlinear Modeling*, Katholieke Universiteit Leuven, Belgium, July 1998, pp. 112–128.

[5] C.Y. Chen, C. C. Chang, and R. C. T. Lee, "A near pattern-matching scheme based upon principal component analysis," *Pattern Recognition Letters*, vol. 16, pp. 339–345, Apr. 1995.

[6] Stelios G. Bakamidis, "An exact fast nearest neighbor identification technique," in *IEEE International Conference on Acoustics, Speech and Signal Processing*, 1993, vol. 5, pp. 658–661.

[7] De-Yuan Cheng, Allen Gersho, Bhaskar Ramamurthi, and Yair Shoham, "Fast search algorithms for vector quantization and pattern matching," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, Mar. 1984, vol. 1, pp. 9.11.1–9.11.4.

[8] Sameer A. Nene and Shree K. Nayar, "A simple algorithm for nearest neighbor search in high dimensions," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 19, no. 9, pp. 989–1003, Sept. 1997.

[9] James McNames, *Innovations in Local Modeling for Time Series Prediction*, Ph.D. thesis, Stanford University, 1999.

[10] Keinosuke Fukunaga and Patrenahalli M. Narendra, "A branch and bound algorithm for computing k-nearest neighbors," *IEEE Transactions on Computers*, vol. C-24, pp. 750–753, July 1975.

[11] Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel, "An algorithm for finding best matches in logarithmic expected time," *ACM Transactions on Mathematical Software*, vol. 3, no. 3, pp. 209–226, Sept. 1977.

[12] Baek S. Kim and Song B. Park, "A fast k nearest neighbor finding algorithm based on the ordered partition," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 761–766, Nov. 1986.

[13] Luisa Micó, José Oncina, and Rafael C. Carrasco, "A fast branch & bound nearest neighbor classifier in metric spaces," *Pattern Recognition Letters*, vol. 17, pp. 731–739, 1996.

[14] Ioannis Katsavounidis, C.-C. Jay Kuo, and Zhen Zhang, "Fast tree-structured nearest neighbor encoding for vector quantization," *IEEE Transactions on Image Processing*, vol. 5, no. 2, pp. 398–404, Feb. 1996.

[15] S. C. Tai, C. C. Lai, and Y. C. Lin, "Two fast nearest neighbor searching algorithms for image vector quantization," *IEEE Transactions on Communications*, vol. 44, no. 12, pp. 1623–1628, Dec. 1996.

[16] Yih-Chuan Lin and Shen-Chuan Tai, "Dynamic windowed codebook search algorithm in vector quantization," *Optical Engineering*, vol. 35, no. 10, pp. 2921–2929, Oct. 1996.

[17] Pasi Fränti, Timo Kaukoranta, and Olli Nevalainen, "On the splitting method for vector quantization codebook generation," *Optical Engineering*, vol. 36, no. 11, pp. 3043–3051, Nov. 1997.

[18] Gene H. Golub and Charles F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, third edition, 1996.

[19] Stephane Lubiarz and Philip Lockwood, "Evaluation of fast algorithms for finding the nearest neighbor," in *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, Apr. 1997, vol. 2, pp. 1491–1494.

[20] Andreas S. Weigend and Neil A. Gershenfeld, *Time Series Prediction*, Addison-Wesley Publishing Company, 1994.

[21] G. Poggi, "Fast algorithm for full-search VQ encoding," *Electronics Letters*, vol. 29, no. 12, pp. 1141–1142, June 1993.

[22] SeongJoon Baek, BumKi Jeon, and Koeng-Mo Sung, "A fast encoding algorithm for vector quantization," *IEEE Signal Processing Letters*, vol. 4, no. 12, pp. 325–327, Dec. 1997.

[23] Mohammad Reza Soleymani and Salvatore D. Morgera, "An efficient nearest neighbor search method," *IEEE Transactions on Communications*, vol. 35, no. 6, pp. 677–679, June 1987.

[24] Kuang-Shyr Wu and Ja-Chen Lin, "An efficient nearest neighbor searching algorithm with application to LBG codebook generation," *Journal of the Chinese Institute of Engineers*, vol. 19, no. 6, pp. 719–724, 1996.

Appendices

A. Notation

This section uses the following notation. The variable n_p denotes the number of points in the data set, $\{x_1, x_2, \ldots, x_{n_p}\}$, where each point exists in a n_d-dimensional space, $x_i \in \Re^{n_d}$ for $i = 1, 2, \ldots, n_p$. The vector $d \in \Re^k$ represents the distance to the nearest neighbors found so far and d^2 represents a vector of the square of these distances.

The element of each data set vector, x_i, is specified by a second subscript. For example, x_{ij} represents the jth element of the vector x_i. The elements of other vectors are specified by a single subscript. For example, p_i represents the ith element of the vector p. The distinction between scalars and vectors is made clear from the context.

B. Search Tree Construction

Principal Axis Tree Construction

1. Define n_y as the number of points assigned to a node N and define $y_i \in \Re^{n_d}$ as the ith point. Define n_c as the number of child nodes assigned to each non-terminal node.
2. $d_i^2 := \infty$ for $i = 1, 2, \ldots, k$.
3. $N :=$ the root node.
4. Assign all of the points in the data set to N.
5. BuildTree(N).

Function BuildTree(N)

1. If $n_y < n_c$, then N is a terminal node. Exit function.
2. Calculate the principal axis, p, for the points assigned to N.
3. Calculate the projections of all of the points assigned to N onto the principal axis: $g_i := \sum_{j=1}^{n_d} y_{ij} p_j$ for $i = 1, 2, \ldots, n_y$.
4. Sort the projected values in increasing order.
5. Divide the principal axis into n_c distinct regions such that the number of projected values in each region is either $\lfloor n_y/n_c \rfloor$ or $\lfloor n_y/n_c \rfloor + 1$. Define G_i as the set of points contained in the ith region for $i = 1, 2, \ldots, n_c$.
6. Calculate the smallest and largest projection for each subset of points, G_i:

$g_{i,\min} := \min_{y_j \in G_i} g_j$ and
$g_{i,\max} := \max_{y_j \in G_i} g_j$.

7. Store the principal axis, p, and subset projection boundaries, $g_{i,\min}$ and $g_{i,\max}$.
8. Create a child node for each subset of points.
   For $i = 1$ to n_c,
   - 8.1 Create a child node C_i and assign the points in the subset G_i to this node.
   - 8.2 BuildTree(C_i).
   - 8.3 Next i.

C. Search Algorithm

Search Algorithm

1. Define q as the query vector and define N as a node in the tree. If N is a non-terminal node, define n_c as the number of child nodes contained in N, C_i as the ith child node, and G_i as the group of points assigned to C_i. Define p as the projection vector of N and define $g_{i,\max}$ and $g_{i,\min}$ as the maximum and minimum values of the points in G_i projected onto p. If N is a terminal node, define n_y as the number of points assigned to N and $y_i \in \Re^{n_d}$ as the ith point. Define L as the list of nearest neighbors.
2. $d_i^2 := \infty$ for $i = 1, 2, \ldots, k$.
3. $N :=$ the root node.
4. $L =$ Search(N,q,q,0).

Function $L =$ **Search**(N,q,b,d^2_{LB})

1. If N is a terminal node, then
   - 1.1 Perform a Partial Distance Search on the points assigned to this node.
   - 1.2 Return the updated list of nearest neighbors, L, and exit function.
2. Calculate the projection of the boundary point onto p: $\sigma := \sum_{i=1}^{n_d} b_i p_i$.
3. Initialize stopping-criteria variables:
   LowerDone := false and
   UpperDone := false.
4. Find the closest group of points using the upper boundaries. Begin by checking end conditions.
   If $\sigma < g_{1,\max}$, then $i_u := 1$ and LowerDone := true.
   Else if $\sigma > g_{n_c-1,\max}$, then $i_u := n_c$, $i_\ell := n_c - 1$, and UpperDone := true.
   Else, perform a binary search:
   - 4.1 $i_\ell := 1$.
   - 4.2 $i_u := n_c$.
   - 4.3 While $i_u - i_\ell > 1$,
     - $i := \lfloor (i_u + i_\ell)/2 \rfloor$.
     - If $\sigma < g_{i,\max}$, then $i_u := i$.
       Else, $i_\ell := i$.
5. Search the child node that contains the region that b is in.
   Search(C_{i_u},q,b,d^2_{LB}) and
   $i_u := i_u + 1$.
6. Initalize variables for main loop:
   If LowerDone = false, then $d_\ell := \sigma - g_{i_\ell,\max}$.
   If UpperDone = false, then $d_u := g_{i_u,\min} - \sigma$.
7. Search the remaining child nodes starting with the closest until the bounding conditions apply.

While UpperDone = false or LowerDone = false,

- 7.1 If (UpperDone = true or $d_\ell < d_u$) and LowerDone = false, then check C_{i_ℓ}.
  - Calculate lower bound on distance to points in G_{i_ℓ}: $d^2_{\mathrm{LB}_\ell} := d^2_{\mathrm{LB}} + d^2_\ell$.
  - If $d_k^2 < d^2_{\mathrm{LB}_\ell}$, then none of the points in G_i for $i = 1, 2, \ldots, i_\ell$ is a nearest neighbor of q. LowerDone := true. Goto 7.
  - If C_{i_ℓ} is not a terminal node, then calculate the boundary point: $b_\ell := b - d_\ell p$.
  - Search(C_{i_ℓ},q,b_ℓ,$d^2_{\mathrm{LB}_\ell}$).
  - $i_\ell := i_\ell - 1$.
  - If $i_\ell = 0$, then LowerDone := true. Else, $d_\ell := \sigma - g_{i_\ell,\max}$.
  - Goto 7.
- 7.2 Else, check node C_{i_u}
  - Calculate lower bound on distance to points in G_{i_u}: $d^2_{\mathrm{LB}_u} := d^2_{\mathrm{LB}} + d^2_u$.
  - If $d_k^2 < d^2_{\mathrm{LB}_u}$, then none of the points in G_i for $i = i_u, i_u + 1, \ldots, n_c$ is a nearest neighbor of q. UpperDone := true. Goto 7.
  - If C_{i_u} is not a terminal node, then calculate the boundary point: $b_u := b + d_u p$.
  - Search(C_{i_u},q,b_u,$d^2_{\mathrm{LB}_u}$).
  - $i_u := i_u + 1$.
  - If $i_u = n_c + 1$, then UpperDone := true. Else, $d_u := g_{i_u,\min} - \sigma$.
  - Goto 7.

Algorithm	Source
Bak	Bakamidis [6]
Brute	Not Applicable
BJS	Baek *et al.* [22]
Cube	Soleymani and Morgera [23]
DOPAT	McNames [9]
DOSVQ	Katsavounidis *et al.* [14]
DWCS	Tai and Lin [15, 16]
FN	Fukunaga and Narendra [10]
FNM	McNames [9]
K-D Tree	Friedman *et al.* [11]
MiniMax	Cheng *et al.* [7]
OPS	Kim and Park [12]
PDS	Cheng *et al.* [7]
Poggi	Poggi [21]
TLAESA	Micó *et al.* [13]
WL-AESA	Wu and Lin [24]

TABLE I

List of algorithms implemented for this study.

Algorithm	Value
PAT	7
FN	16
FNM	50
TLAESA	3
K-D Tree	40
Bak	2

TABLE II

Values chosen for algorithms that have a user-specified parameter.

Algorithm	8 Dimensions				16 Dimensions				Rank
	1024	2048	4096	8192	1024	2048	4096	8192	
PAT	0.056	0.076	0.108	0.122	0.214	0.300	0.385	0.442	1.4
DOSVQ	0.057	0.081	0.132	0.159	0.185	0.272	0.375	0.478	1.6
DOPAT	0.068	0.101	0.175	0.221	0.242	0.382	0.579	0.867	3.0
DWCS	0.103	0.174	0.290	0.417	0.297	0.500	0.773	1.163	4.1
K-D Tree	0.191	0.231	0.327	0.357	0.647	0.940	1.221	1.471	5.0
FNM	0.329	0.538	0.925	1.337	0.601	0.968	1.601	2.669	5.9
FN	0.544	0.828	1.438	2.065	1.363	1.991	3.089	4.906	7.6
PDS	0.590	1.118	2.265	4.378	0.895	1.684	3.159	5.975	8.4
BJS	0.581	1.210	2.340	4.524	1.088	2.161	3.980	7.585	9.3
Cube	0.718	1.326	2.607	4.955	1.490	2.661	4.693	8.433	10.9
Poggi	0.789	1.595	3.615	7.171	1.166	2.289	4.566	9.223	11.1
OPS	0.839	1.100	1.514	1.982	2.655	5.399	11.005	22.855	11.6
TLAESA	1.118	2.722	2.991	3.983	2.017	4.479	7.739	11.589	12.5
WL-AESA	1.044	2.302	4.665	9.889	2.044	4.045	8.204	16.030	13.1
Bak	1.112	2.530	5.500	10.344	2.528	5.663	11.611	21.877	14.5
Brute	1.903	3.769	7.672	15.460	3.581	7.243	14.629	29.606	16.0
MiniMax	3.128	6.250	12.632	25.639	5.867	11.884	23.845	48.220	17.0

TABLE III

ALGORITHMS' AVERAGE QUERY TIMES (MILLISECONDS) TO FIND THE NEAREST NEIGHBOR FOR A CODEBOOK CONSTRUCTED FROM A *speech signal*. THE RESULTS ARE SHOWN FOR FOUR CODEBOOK SIZES AND TWO VECTOR SIZES.

Algorithm	2×2				4×4				Rank
	1024	2048	4096	8192	1024	2048	4096	8192	
DOSVQ	0.028	0.034	0.045	0.055	0.075	0.105	0.134	0.159	1.5
PAT	0.026	0.035	0.043	0.048	0.090	0.130	0.168	0.186	2.3
DWCS	0.027	0.039	0.059	0.084	0.077	0.116	0.156	0.178	2.8
DOPAT	0.028	0.038	0.053	0.062	0.085	0.132	0.184	0.243	3.5
K-D Tree	0.054	0.069	0.088	0.106	0.237	0.349	0.457	0.526	5.0
FNM	0.134	0.257	0.406	0.562	0.288	0.567	0.849	1.287	6.5
FN	0.215	0.367	0.558	1.006	0.610	0.976	1.432	2.062	8.1
Bak	0.156	0.343	0.772	1.577	0.563	1.158	1.871	2.451	8.6
TLAESA	0.247	0.469	0.770	1.371	0.546	1.112	1.911	2.552	8.9
PDS	0.456	0.942	1.867	3.720	0.609	1.184	2.369	4.607	10.4
OPS	0.080	0.103	0.142	0.143	2.386	4.984	10.129	21.602	10.5
Cube	0.495	1.014	2.005	3.982	0.873	1.618	3.086	5.747	11.8
WL-AESA	0.474	1.047	2.153	4.553	0.971	1.884	3.392	5.666	12.3
BJS	0.669	1.351	2.685	5.671	1.362	2.727	5.561	12.346	13.5
Poggi	0.876	1.761	3.541	7.167	2.328	4.774	9.991	20.699	14.5
Brute	1.013	2.079	4.150	8.352	3.504	7.102	14.674	29.761	16.0
MiniMax	1.755	3.498	7.121	14.336	5.690	11.575	23.633	47.709	17.0

TABLE IV

ALGORITHMS' AVERAGE QUERY TIMES (MILLISECONDS) TO FIND THE NEAREST NEIGHBOR FOR A CODEBOOK CONSTRUCTED FROM THE IMAGE *Lena*. THE RESULTS ARE SHOWN FOR FOUR CODEBOOK SIZES AND TWO VECTOR SIZES.

Algorithm	2 × 2				4 × 4				Rank
	1024	2048	4096	8192	1024	2048	4096	8192	
DOSVQ	0.026	0.034	0.043	0.053	0.084	0.110	0.139	0.154	1.5
PAT	0.025	0.032	0.040	0.044	0.097	0.132	0.174	0.194	1.8
DWCS	0.028	0.042	0.063	0.090	0.092	0.138	0.184	0.207	3.4
DOPAT	0.028	0.037	0.049	0.064	0.096	0.138	0.194	0.262	3.4
K-D Tree	0.054	0.066	0.083	0.108	0.274	0.393	0.510	0.547	5.0
FNM	0.140	0.254	0.384	0.617	0.318	0.575	0.900	1.361	6.5
FN	0.212	0.330	0.559	0.797	0.621	1.037	1.595	2.327	7.6
TLAESA	0.262	0.560	1.083	1.317	0.672	1.224	2.598	3.756	9.1
Bak	0.171	0.403	0.911	2.030	0.732	1.470	2.723	3.120	9.3
PDS	0.486	0.908	1.841	3.699	0.672	1.266	2.542	4.884	10.0
OPS	0.079	0.101	0.136	0.163	2.416	4.924	10.591	23.157	10.5
Cube	0.525	0.985	1.981	3.957	0.998	1.797	3.401	6.255	11.5
WL-AESA	0.604	1.245	2.589	5.381	1.142	2.101	4.095	7.030	12.6
BJS	0.706	1.384	2.653	5.139	1.431	2.615	5.117	10.409	13.4
Poggi	0.934	1.856	3.692	7.990	2.381	4.661	9.536	19.432	14.5
Brute	1.039	2.013	4.103	8.250	3.549	7.068	14.686	29.536	16.0
MiniMax	1.719	3.412	7.027	14.052	5.779	11.514	23.585	47.598	17.0

TABLE V

ALGORITHMS' AVERAGE QUERY TIMES (MILLISECONDS) TO FIND THE NEAREST NEIGHBOR FOR A CODEBOOK CONSTRUCTED FROM THE IMAGE *Boat*. THE RESULTS ARE SHOWN FOR FOUR CODEBOOK SIZES AND TWO VECTOR SIZES.

Algorithm	2 × 2				4 × 4				Rank
	1024	2048	4096	8192	1024	2048	4096	8192	
PAT	0.033	0.041	0.051	0.054	0.304	0.446	0.553	0.385	1.5
DOSVQ	0.034	0.044	0.061	0.073	0.273	0.431	0.596	0.468	1.8
DOPAT	0.036	0.045	0.065	0.079	0.284	0.443	0.621	0.503	2.8
DWCS	0.052	0.080	0.130	0.182	0.312	0.526	0.748	0.636	4.4
K-D Tree	0.071	0.082	0.107	0.122	0.710	1.129	1.602	1.327	5.0
FNM	0.174	0.255	0.449	0.667	0.587	1.042	1.757	2.313	6.3
FN	0.255	0.382	0.636	0.837	1.182	1.927	3.370	4.679	7.8
PDS	0.482	0.914	1.961	3.872	1.011	1.913	3.601	6.397	9.0
TLAESA	0.385	0.609	1.077	2.624	1.742	3.205	6.315	10.311	9.3
OPS	0.110	0.126	0.159	0.179	2.632	5.617	11.786	22.936	10.1
Cube	0.532	1.006	2.119	4.162	1.917	3.412	5.947	9.945	10.8
Bak	0.427	0.903	2.064	4.668	1.996	4.193	8.829	11.489	11.6
BJS	0.610	1.146	2.333	4.489	1.849	3.550	7.032	12.732	11.9
Poggi	0.732	1.427	3.047	6.511	2.069	4.157	8.536	16.237	13.4
WL-AESA	1.162	2.557	5.516	11.462	3.159	5.975	9.613	13.260	15.1
Brute	0.999	1.967	4.145	8.313	3.514	7.150	14.700	29.636	15.5
MiniMax	1.754	3.395	7.093	14.160	5.801	11.663	23.802	47.870	17.0

TABLE VI

ALGORITHMS' AVERAGE QUERY TIMES (MILLISECONDS) TO FIND THE NEAREST NEIGHBOR FOR A CODEBOOK CONSTRUCTED FROM THE IMAGE *Babbon*. THE RESULTS ARE SHOWN FOR FOUR CODEBOOK SIZES AND TWO VECTOR SIZES.

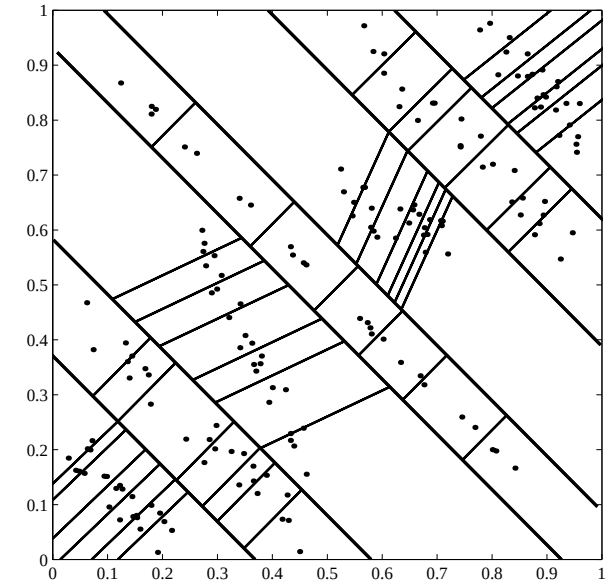

Fig. 1. Partition of a two-dimensional data set using a principal axis tree. Each non-terminal node in the tree subdivided its assigned set of points into seven subsets. In this example, the tree had a depth of two and divided each of the seven primary regions into seven subsets. Note that the direction of each partition is in the direction of maximum variance.

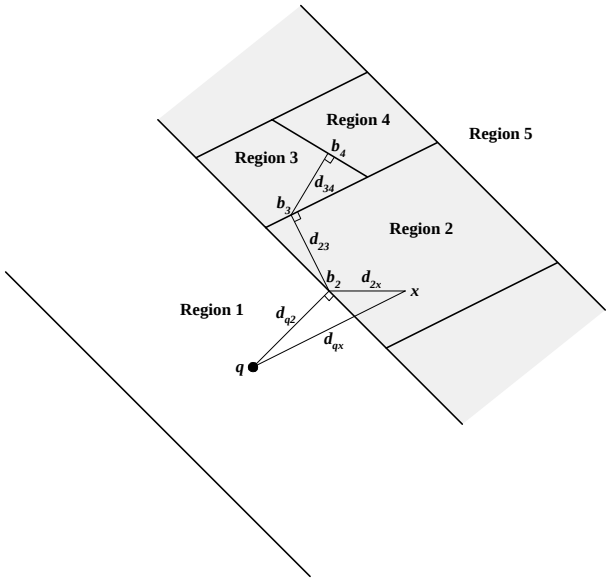

Fig. 2. Illustration of the bounding criteria of principal axis trees. In this example the query point is located in Region 1 and denoted by q. The chords labeled d_{q2}, d_{23}, and d_{34} are used to establish a lower bound on the distance to points in the gray region, Region 3, and Region 4 respectively.

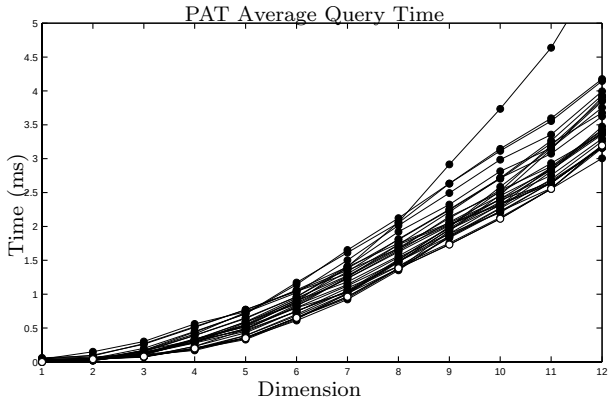

Fig. 3. Average query time of PAT to find five nearest neighbors as a function of dimension for various numbers of children per non-terminal node, the user-specified parameter. The data set consisted of $2,000$ points drawn from a normal distribution. The white circles show the average query times for seven children per non-terminal node.

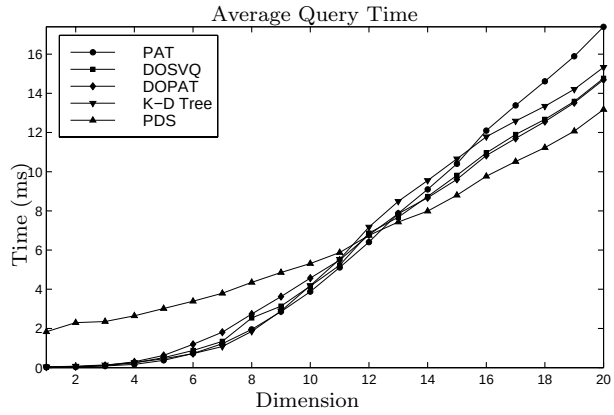

Fig. 4. Average query time to find five nearest neighbors as a function of dimension. The data sets and query points were drawn from a *uniform distribution*. The data set contained $5,000$ points. Only the best five algorithms are shown.

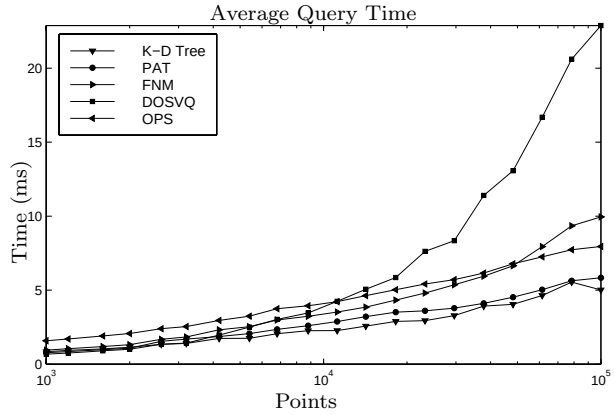

Fig. 5. Average query time to find five nearest neighbors as a function of the data set size. The data sets and query points were drawn from a *uniformly distribution* in eight dimensions. Since the horizontal axis is on a log scale, a straight line indicates that the average query time scales as $O(\log n_p)$ and an exponentially increasing trace scales as $O(n_p)$, where n_p is the number of points in the data set. Only the best five algorithms are shown.

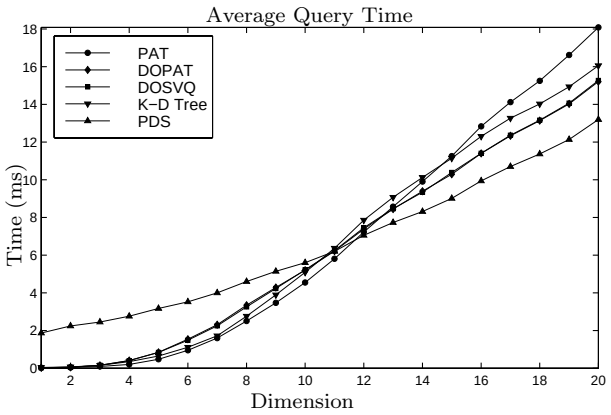

Fig. 6. Average query time to find five nearest neighbors as a function of dimension. The data sets and query points were drawn from a *normal distribution*. The data set contained $5,000$ points. Only the best five algorithms are shown.

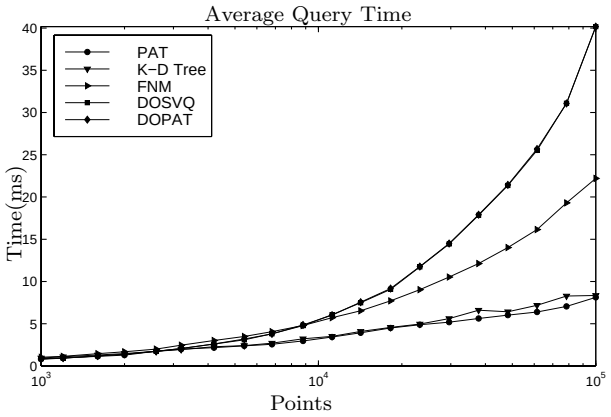

Fig. 7. Average query time to find five nearest neighbors as a function of the data set size. The data sets and query points were drawn from a *normal distribution* in eight dimensions. Only the best five algorithms are shown.

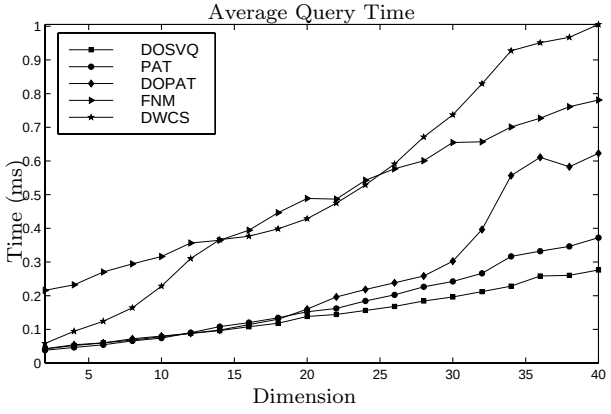

Fig. 8. Average query time to find five nearest neighbors as a function of dimension. The data sets and query points were created from the *Lorenz time series*. The data set contained $5,000$ points. Only the best five algorithms are shown.

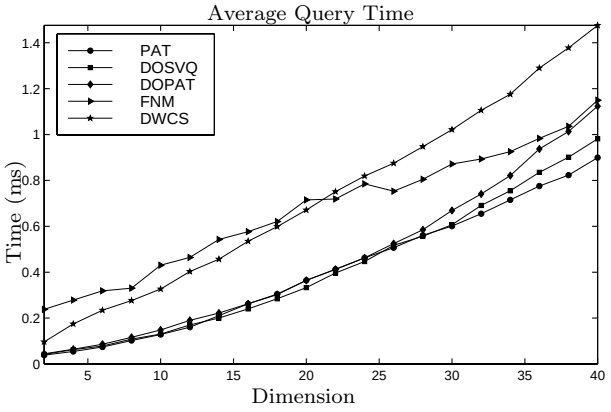

Fig. 9. Average query time to find five nearest neighbors as a function of dimension. The data sets and query points were created from the *Santa Fe time series*. The data set contained $5,000$ points. Only the best five algorithms are shown.