Algorithmica (1991) 6: 579–589

Algorithmica


# Refinements to Nearest-Neighbor Searching in $k$-Dimensional Trees

Robert F. Sproull[1]

**Abstract.** This note presents a simplification and generalization of an algorithm for searching $k$-dimensional trees for nearest neighbors reported by Friedman *et al.* [3]. If the distance between records is measured using $L_2$, the Euclidean norm, the data structure used by the algorithm to determine the bounds of the search space can be simplified to a single number. Moreover, because distance measurements in $L_2$ are rotationally invariant, the algorithm can be generalized to allow a partition plane to have an arbitrary orientation, rather than insisting that it be perpendicular to a coordinate axis, as in the original algorithm. When a $k$-dimensional tree is built, this plane can be found from the principal eigenvector of the covariance matrix of the records to be partitioned. These techniques and others yield variants of $k$-dimensional trees customized for specific applications.

It is wrong to assume that $k$-dimensional trees guarantee that a nearest-neighbor query completes in logarithmic expected time. For small $k$, logarithmic behavior is observed on all but tiny trees. However, for larger $k$, logarithmic behavior is achievable only with extremely large numbers of records. For $k = 16$, a search of a $k$-dimensional tree of 76,000 records examines almost every record.

**Key Words.** $k$-dimensional tree, Searching, Nearest-neighbor search.

**1. Introduction.** A $k$-dimensional tree is an elegant data structure for organizing records of multidimensional data points [1]. For example, a two-dimensional tree might hold records of two-dimensional coordinates of points on a map; a three-dimensional tree might hold coordinates of atoms in a molecule; and higher-dimensional trees might hold records of multivariate data recorded as part of a scientific experiment or features for a pattern-recognition task. The nearest-neighbor search problem is to find the record in the tree closest to a given *query record.* In this note we assume that the measure of distance in the search is the Euclidean norm, $L_2$, the square root of the sum of the squares of the differences along each coordinate direction.

A $k$-dimensional tree is a binary tree in which each nonterminal node has two descendants, a *left son* and a *right son.* Each nonterminal node partitions the records into two groups according to a record's position with respect to a $k$-dimensional partition hyperplane: records that lie on the "left" side of the hyperplane are stored in the left son subtree, while records that lie on the "right" side of the hyperplane are stored in the right son subtree. The tree can be elaborated to any depth, ending with terminal nodes that hold up to $b$ coordinate records. The reader should see Bentley's original paper [1] for a lucid exposition of $k$-dimensional trees.

[1] Sutherland, Sproull & Associates, P.O. Box 1160, Palo Alto, CA 94302, USA.

Received July, 1988; revised October, 1989. Communicated by Jon Bentley.

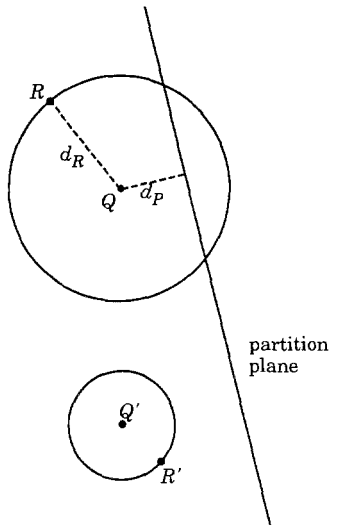


**Fig. 1.** Relationship between a query record, $Q$, and the nearest record, $R$. Because $d_P < d_R$, the record nearest $Q$ may lie to the right of the partition plane.

**2. Nearest-Neighbor Searching.** The nearest-neighbor search algorithm descends the nonterminal nodes of the $k$-dimensional tree, choosing at each node to investigate either the left or right son according to whether the query record, $Q$, lies on the left or right side of the partition plane. When the search reaches a terminal node, the records in the terminal node are examined exhaustively to find the one closest to the query record. The result of this examination is a *current nearest record*, $R$, and a *current nearest distance*, $d_R$, the distance between the query record and the current nearest record.

Unfortunately, the search is not complete at this point, as can be explained with the aid of Figure 1. The figure shows a single partition plane in a two-dimensional space: the query record, $Q$, lies to the left of the plane, as does the current nearest record, $R$. If there is a record that is closer to the query record than the current nearest record, it lies within a circle of radius $d_R$ centered at $Q$. Since part of this circle lies to the right of the partition, the algorithm must search records on the right of the partition before it can guarantee having found the nearest neighbor of the query record. Of course, it is not always necessary to search the other side of the partition, as shown by query record $Q'$ and current nearest match $R'$: no part of the circle lies on the other side of the partition. To decide whether the other side must be searched, the algorithm need only compare the radius of the circle, $d_R$, to the distance, $d_P$, from the query record to the partition plane. Although this reasoning has been illustrated in two dimensions, it applies equally well for any number of dimensions.

The first refinement to the algorithm of Friedman *et al.* [3] is to observe that the determination of whether the other side of a partition plane must be examined can be based on a simple comparison of $d_R$ and $d_P$. Moreover, the algorithm can avoid computing the square root associated with the $L_2$ norm by comparing the squares

of the two distances. The original algorithm requires maintaining a more complex data structure to bound the search along each coordinate axis and complex "bounds overlap ball" and "ball within bounds" tests to determine whether to search on the other side of a partition plane. The paper observes "The overhead required to search the tree is dominated by the bounds-overlap-ball calculation." The refinement described here drastically reduces this and other overheads.

**3. Arbitrary Partition Planes.** A further refinement to the algorithm is to allow partition planes to be arbitrary $k$-dimensional hyperplanes, whereas the original algorithm requires partition planes to be perpendicular to a coordinate axis. This refinement is possible because distances measured with $L_2$ are invariant under rotation. However, the refinement is not always advisable because more computation is required to determine $d_P$ when the partition is an arbitrary plane.

Before showing how to find arbitrary partition planes, it is instructive to recall Bentley's method for finding partition planes [1]:

1. Compute the variance of the records' coordinate values along each coordinate axis. Find the coordinate axis with maximum variance, which we shall call the *partition axis.* The partition plane will be perpendicular to this axis, so the data will be split along the direction of maximum variation.
2. Erect a provisional partition plane that passes through the origin and is perpendicular to the partition axis. Then determine $d_{\mathrm{m}}$, the median of the distances from the records to this plane. Finally, translate the plane along the partition axis by a distance $d_{\mathrm{m}}$ so that the plane now partitions the set of records into two subsets of very nearly equal size.

To use arbitrary partition planes, we need to modify only the first step. The partition axis is obtained by computing the principal eigenvector of the covariance matrix of the coordinate records. Informally, this axis points along the direction of maximum variation in the data, regardless of how the data are oriented with respect to the coordinate axes.

The value of using arbitrary partition planes depends on properties of the data held in the $k$-dimensional tree. If the data records are homogeneously distributed in all dimensions, arbitrary partition planes offer no advantage; data sets with this distribution are termed *random.* To illustrate the benefits of arbitrary planes, the algorithm was tested with data sets that are highly skewed, obtained by taking $k$-element sequences from a digitally sampled signal with high autocorrelation; these data sets are called *signal.* In *signal* data, adjacent elements of the $k$-vectors are usually about equal, but the differences between sample values are significant. Arbitrary hyperplanes will effectively categorize and split such data.

Table 1 shows the improvement that arbitrary partition planes can yield. Although little difference is shown for random records or small $k$, improvements of more than a factor of two appear for *signal* records with $k = 16$. More information on the performance of this technique, including tradeoffs between time spent building and searching the tree, is shown in Sections 6 and 7.

**Table 1.** The number of records examined when searching a $k$-dimensional tree containing $N = 1047$ records with a maximum of $b = 5$ records per terminal node. The column labeled VMS uses partition axes parallel to a coordinate axis; the column labeled PMS uses arbitrary partition axes.

| | | Number of records examined | |
|---|---|---|---|
| Data | $k$ | VMS | PMS |
| Random | 2 | 9.3 | 9.5 |
| | 4 | 37.9 | 43.4 |
| | 8 | 418 | 432 |
| | 16 | 1045 | 1032 |
| Signal | 2 | 8.0 | 9.2 |
| | 4 | 32.5 | 33.2 |
| | 8 | 96 | 63.6 |
| | 16 | 238 | 104 |

**4. Optimizing the Exhaustive Search.** A classic technique can be used to speed up the exhaustive search of records held in a terminal node of the tree: as the sum of squares is being accumulated to compute the distance from the query record to each tree record, the accumulation loop exits when the sum exceeds the square of $d_R$, the current nearest distance, because the record being examined is clearly farther from the query record than the current best record. While this *excessive sum* optimization is important on conventional sequential computers, it interferes with vectorizing techniques that can be applied to the accumulation loop on vector computers.

The effect of this optimization has been measured on the *signal* data set. Let us define $r_m$ to be the ratio of the number of multiplications actually performed by the optimized search to the number that would be performed without this optimization. The reason this optimization is interesting is that $r_m$ can be quite small, e.g., 0.15, resulting in substantial speedups. The value of $r_m$ decreases as the number of records in a terminal node increases, and also as $k$ increases. Figure 2 shows values of $r_m$ for the *signal* data for various values of $b$, the maximum number of records in each terminal node of the $k$-dimensional tree.

The excessive sum optimization interacts with the structure of the $k$-dimensional tree in an interesting way. The improvement offered by the optimization blunts the effect of further partitioning terminal nodes by additional $k$-dimensional tree structuring. As more tree structure is introduced to reduce the number of records held in each terminal node, $r_m$ increases, thus partly offsetting the benefits of the additional structure. By the same token, improvements in the partitioning effected by the $k$-dimensional tree, for example by using arbitrary partition planes, blunt the improvement offered by the excessive sum optimization by reducing the number of far distant points encountered in a terminal node. Table 1 shows that, for $k = 16$, arbitrary partition planes result in examining fewer than half as many records as axis partition planes, which suggests that the use of arbitrary planes might lead to a speedup of a factor of two. However, Figure 3, which shows actual search times,

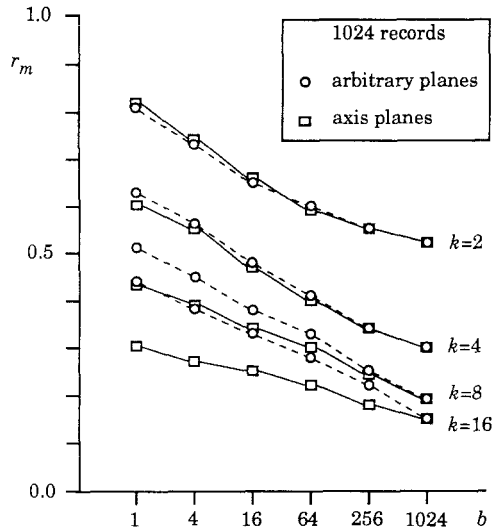


**Fig. 2.** The effect of the number of records in each terminal node, $b$, on the fraction of multiplies used, $r_m$, for *signal* data. The lines serve only to associate points, not to indicate a continuous function.

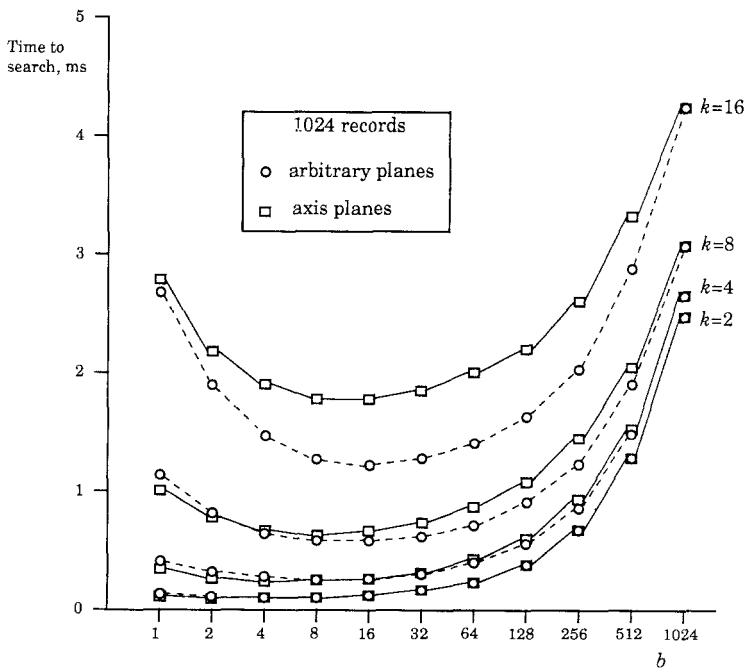


**Fig. 3.** Time to perform a nearest-neighbor query given different numbers of records in terminal nodes, for *signal* data. The lines serve only to associate points, not to indicate a continuous function.

confirms that the speedup using arbitrary partition planes for $k = 16$ is a factor of only 1.5.

**5. The Search Algorithm.** The nearest-neighbor search algorithm incorporating the refinements described above is shown in Program 1 (compare with Appendix II of [3]). This algorithm requires less bookkeeping effort than its forerunner, a consequence of restricting the distance measure to $L_2$. It allows each internal node to partition the records using an arbitrary hyperplane or a hyperplane perpendicular to a coordinate axis. The algorithm is given in Pascal, augmented with a *leave* statement to exit a loop. To avoid taking square roots as specified in the $L_2$ norm, variables often contain the square of a distance; the names of these variables end in the character "2" as a reminder.

**6. Variations on a Theme.** The original $k$-dimensional tree search algorithm, together with techniques presented in this and other papers, comprise a parts kit from which tree structures and search procedures can be tailored. Although some combinations of techniques cannot be used, the remainder provide the programmer with considerable flexibility in designing search algorithms. The following taxonomy helps to describe the possibilities, listed roughly in order of increasing computation:

1. Partition plane orientation:
   - C Plane orthogonal to a coordinate axis, in which the axes are chosen in a *cyclic* pattern with increasing tree depth [4].
   - E Plane orthogonal to the coordinate axis with greatest range, i.e., distance between *extreme* values.
   - V Plane orthogonal to the coordinate axis with greatest *variance* [1].
   - P Plane orthogonal to the *principal* axis, as described in this paper.
2. Partition plane position:
   - R Bisector of *range* of distances from partition plane to data.
   - $\mu$ *Mean* of distances.
   - M *Median*, or median estimate, of distances [1].
   - N *Nearest*-neighbor cut planes, effective for partitioning certain pathological distributions of records [2].
3. Distance representations:
   - B *Bounds* in each coordinate dimension [3].
   - S *Scalar* distance, using Euclidean norm and avoiding square root extraction, as shown in this paper.

The original $k$-dimensional tree search algorithm is characterized as VMB. A good algorithm for general use is VMS, which uses scalar distance techniques to reduce the overhead of bounds tests of the original algorithm, but does not use arbitrary partition planes that require expensive tree-building procedures. Certain combinations of techniques do not make sense, e.g., PNx, because nearest-neighbor cut planes must be orthogonal to coordinate axes [2]. Arbitrary partition plane

```
TYPE

pKDRecord = ^ KDRecord;
KDRecord = RECORD                    { Record in the tree or query }
   vec:  ARRAY [1..k] OF real;       { coordinates of the point }
   next: pKDRecord;                  { link for list of records in terminal node }
END;

NodeType = (terminal, nonterminal);

pKDNode = ^ KDNode;
KDNode = RECORD                      { Record for terminal or non-terminal node }
   CASE typ: NodeType OF
terminal:
   (recordList:   pKDRecord);        { list of records in terminal node }
nonterminal:
   (leftson:   pKDNode;              { pointers to descendant nodes }
   rightson:   pKDNode;
   discriminator: INTEGER;           { 0 if arbitrary plane, else index of
                                     { coordinate along which to partition }
   partition:  real;                 { partition value }
   plane:      ARRAY [1..k] OF real); { normal vector along principal axis }
END;

pKDTree = ^ KDTree;
KDTree = RECORD                      { Record that describes an entire kd tree }
   root:    pKDNode;                 { root of the tree }
   query:   pKDRecord;               { query record, used during search }
   R:       pKDRecord;               { current best record, used during search }
   DR2:     real;                    { square of current nearest distance }
END;

{ Search a k-d tree for the record nearest to 'queryRec' }

FUNCTION Search(tree: pKDTree; queryRec: pKDRecord): pKDRecord;
BEGIN WITH tree^ DO BEGIN
   query := queryRec;
   DR2 := MAXREAL;                   { MAXREAL is the largest real. }
   Search_Aux(tree, root);
   Search := R;
END; END;

{ Recursive procedure to search 'node' of the 'tree' }

PROCEDURE Search_Aux(tree: pKDTree; node: pKDNode);
VAR
   dist2, DP, DP2: real;
   i: integer;
   rec: pKDRecord;
BEGIN WITH tree^, node^ DO BEGIN
   IF typ = terminal THEN BEGIN { terminal node }
      { Exhaustive search of all records attached to this terminal node. }
      rec := recordList;
      WHILE rec <> NIL DO BEGIN
         dist2 := 0;
         FOR i := 1 TO k DO BEGIN
            dist2 := dist2 + (rec^.vec[i] - query^.vec[i])**2;
            IF dist2 > DR2 THEN leave; { Quick exit optimization }
         END;
         IF dist2 < DR2 THEN BEGIN { Closer match }
            DR2 := dist2;
            R := rec;
         END;
         rec := rec^.next;
      END;
   END { terminal node } ELSE BEGIN { nonterminal node }
      { Compute DP, distance to the partition plane }
      IF discriminator <> 0 THEN { Plane orthogonal to axis }
         DP := query^.vec[discriminator] - partition
      ELSE BEGIN { Arbitrary plane }
         DP := -partition;
         { plane[i] is normal vector along principal axis }
         FOR i := 1 TO k DO DP := DP + query^.vec[i] * plane[i];
      END;
      DP2 := DP*DP;  { Form the square }

      { Determine where the query point lies with respect to the partition
      { plane and make a resursive call. }
      IF DP < 0 THEN BEGIN { left son }
         Search_Aux(tree, leftson);
         { May need to search other side of partition. }
         IF DP2 < DR2 THEN Search_Aux(tree, rightson);
      END ELSE BEGIN { right son }
         Search_Aux(tree, rightson);
         { May need to search other side of partition. }
         IF DP2 < DR2 THEN Search_Aux(tree, leftson);
      END;
   END;
END; END;
```

**Program 1**

orientations are also not suitable for Bentley's variant that does bottom-up searching [2].

The time spent building a $k$-dimensional tree and the time spent searching it have different relative importance in different applications. ERS trees are quickly built, but may take longer to search. Combination PMS requires substantial computation to build the tree, but can return savings in search time, as shown in Table 1. However, in some cases, the computation required by choice P is prohibitive: in an application with $k = 192$, the computation required by a $k^3$ algorithm to compute the principal eigenvector was too large, so method V was used instead. Because of the performance variations that result from tree-building choices, our package of routines for building and manipulating $k$-dimensional trees supports all combinations of partition-plane choice, as well as statistics-taking that can lead to more efficient choices on subsequent runs.

A tradeoff is required to determine whether to use arbitrary hyperplane partitions on a given set of records. The search requires more computation at internal nodes with arbitrary partition planes than with planes orthogonal to coordinate axes: $k$ multiplies and $k$ additions are required to compute the distance to an arbitrary hyperplane, contrasted with a single addition for a plane orthogonal to an axis. The difference will be pronounced if the tree is quite deep, since more internal nodes will be visited. On the other hand, if the data are skewed with respect to all the coordinate axes, arbitrary partition planes will do a much better job of partitioning the data, which will be reflected in fewer nodes being visited because the circle surrounding the query point (Figure 1) will intersect the partition plane less often. Sometimes these tradeoffs can be dramatic. A tree of 167,555 records of *signal* data with $k = 64$ was searched 2700 times: combination VRS used 10 min to build the tree and 120 min to search it, while combination PMS used 75 min to build the tree and 30 min to search it.

One way to estimate the value of using an arbitrary plane is to compute the ratio, $r_v$, of the variance of the records along the principal axis, which is just the principal eigenvalue of the covariance matrix, to the largest variance along a coordinate axis, which is the maximum diagonal element of the covariance matrix. If this ratio is large, we know that the data are highly skewed and that the principal axis is not close to a coordinate axis. In this situation, an arbitrary partition plane is likely to be more valuable than an axis partition plane. For example, the *signal* data set with $k = 16$ has $r_v = 14$, suggesting that arbitrary partition axes will perform much better than partition axes aligned with the coordinate axes.

A simple argument can be used to obtain bounds on this ratio: $1 \leq r_v \leq k$. The lower bound is obvious: the principal eigenvalue of the covariance matrix is never less than the variance along any of the coordinate axes. To obtain the upper bound, consider the case where all the data records lie along a line whose projection along each axis is the same. In two dimensions, this is the 45° line. The variances along each coordinate axis will be identical, and the variance along the principal axis, which lies on the line, will be $k$ times the variance along an axis. Thus $r_v = k$.

We have experimented with building trees by setting a threshold, $r_{vt}$, on this ratio: when $r_v \geq r_{vt}$ at a node, an arbitrary partition plane is used, otherwise an axis partition plane is used. Choosing the best threshold $r_{vt}$ will depend on $k$ and the

characteristics of the computer used, especially the relative cost of arithmetic that computes distances to planes and the bookkeeping associated with recursive calls of the search procedure.

**7. Performance.** $K$-dimensional records are organized in a $k$-dimensional tree to speed searches of various kinds. Friedman *et al.* [3] argue that, under a certain set of assumptions, a nearest-neighbor query requires logarithmic expected time and examines approximately $R$ records, where

$$R(k, b) \approx b\{[G(k)/b]^{1/k} + 1\}^k. \tag{1}$$

$G(k) \geq 1$ is a constant that captures geometric properties of the norm used for distance measurement and that adjusts for the fact that the search examines a region of space somewhat greater than the hypersphere centered at the query point and just touching the nearest neighbor; we assume $G(k) = 1$ in subsequent calculations. To obtain logarithmic behavior, $N$, the number of records in the tree, must be much greater than $R$. For small $k$, this assumption is easy to satisfy, and $k$-dimensional trees perform very well.

Figure 4 shows the results of experiments for different values of $k$ and $N$. Each experiment generates $N$ records uniformly distributed in $k$-dimensional space, builds a VMS tree in which the maximum number of records in a terminal node is $b = 5$, and performs 1000 nearest neighbor searches with random queries. Figure 4(a) shows that, for $k = 2$, the $k$-dimensional tree is very effective. A cyclic behavior is evident as terminal nodes fill up, then split when the number of terminal nodes reaches a power of two [2]. Note that the number of records examined is approximately a constant near ten, which compares favorably with $R(2, 5) = 10.5$. The number of nodes visited appears to grow logarithmically.

As $k$ increases, the performance of $k$-dimensional trees deteriorates. To achieve logarithmic search times, it is necessary that $N$ grows exponentially with $k$, a requirement that may be hard to meet. When $N < R$, a large portion of the $N$ records may be searched. For example, Figure 4(d) reveals that, for $k = 16$, where $R(16, 5) = 150{,}000$, a search of $N = 75{,}857$ records examines 72,774 records —nearly every one! Moreover, it appears that (1) can substantially underestimate the number of records that must be searched. Figure 4(c) shows that, for $k = 8$, over 1400 records are examined when $N = 100{,}000$, although $R(8, 5) = 600$.[2]

When points are not homogeneously distributed, $k$-dimensional tree performance for large $k$ improves. The $k$-dimensional tree structure is good at excluding large portions of empty space from consideration and concentrating search on clusters of points. Moreover, points may cluster so as to exhibit a lower effective dimensionality. For example, imagine a set of points with $k = 16$, with 14 of the 16 coordinates always zero, and the values of the remaining two coordinates distributed uniformly. These points are uniformly distributed along two coordinate axes,

[2] Although it may be possible to adjust $G(k)$ to achieve a better fit, it is also possible that (1) does not properly represent the behavior of $k$-dimensional trees.

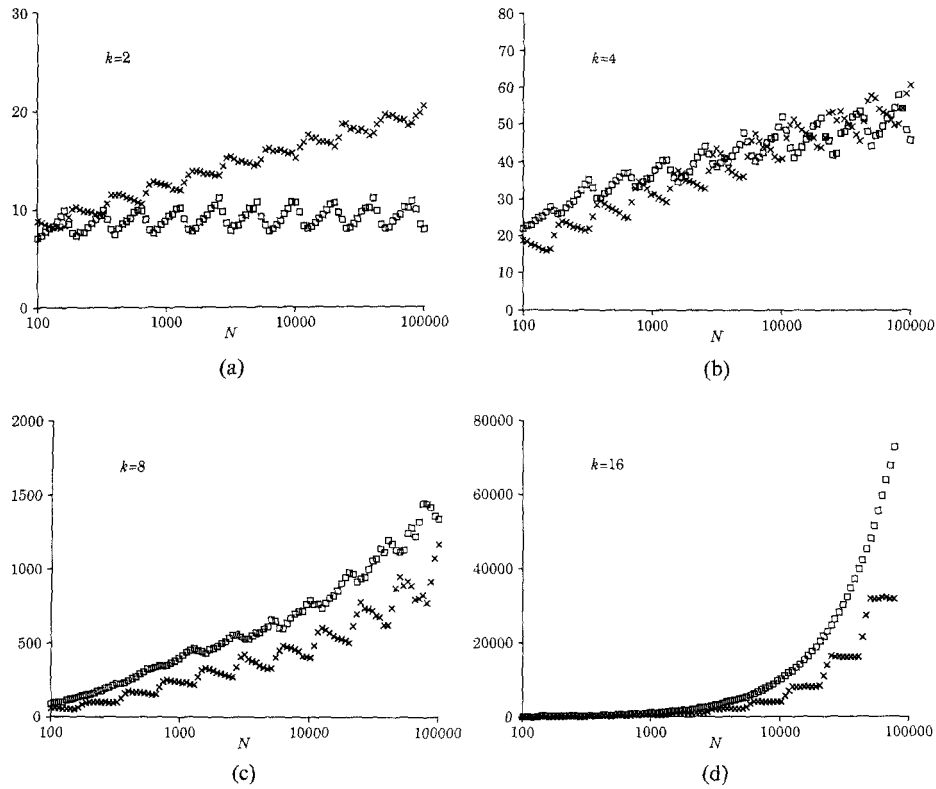


**Fig. 4.** Experiments using $k$-dimensional trees built using VMS techniques to store and retrieve $N$ *random* records. Each terminal node holds $b = 5$ or fewer records. The boxes show the number of records examined and the crosses the number of tree nodes visited on each search. (a) $k = 2$, (b) $k = 4$, (c) $k = 8$, and (d) $k = 16$.

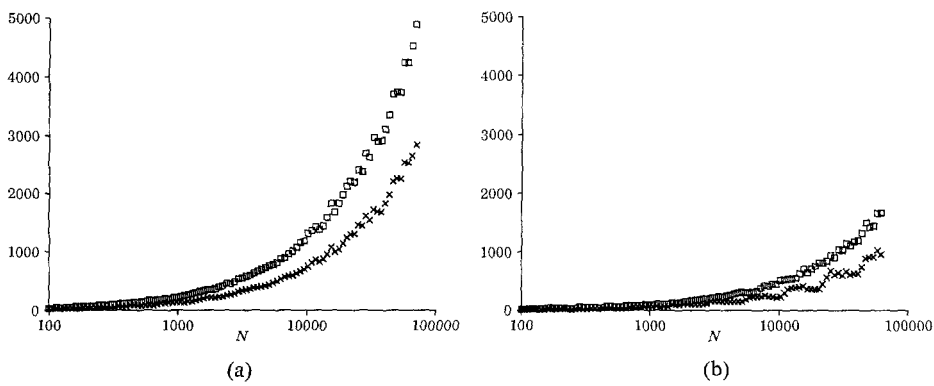


**Fig. 5.** Experiments using $k$-dimensional trees to store and retrieve $N$ *signal* records, $k = 16$, $b = 5$. (a) VMS and (b) PMS.

and the performance of $k$-dimensional tree searches of these points will match the $k = 2$ case; the $k$-dimensional tree structure ignores all axes along which there is no variation. Although this is an extreme example, $k$-dimensional trees adapt well to the distribution of records and can show impressive improvements compared with the case of uniformly distributed records. Figure 5 shows results from experiments analogous to those of Figure 4, except that *signal* data is used for points and queries. Figure 5(a) shows that a VMS tree searches many fewer records for *signal* data than for *random*. Figure 5(b) shows that the PMS performance is still better, presumably because of its superior ability to partition clusters locally.

The best algorithm for a particular application will depend on properties of the data being searched and, to a lesser extent, on the properties of the compiler and computer used for translating and executing the algorithm. Although it would be nice to make such determinations analytically, $k$-dimensional tree search performance has yet to be modeled analytically, 12 years after the algorithm was developed (see [4]). Experiments and tuning such as illustrated in this paper are required, especially when the data are inhomogeneously distributed.

**Acknowledgments.** Discussions with Ivan Sutherland led to the use of arbitrary partition planes. Tom Stockham suggested the excessive sum optimization. Comments of Sutherland and Jon Bentley greatly improved the paper. I am grateful to Apple Computer for support of this work.

## References

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

[2] J. L. Bentley, *K*-dimensional Trees for Semidynamic Point Sets, AT&T Bell Laboratories, Murray Hill, NJ, 1989.

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

[4] J. E. Zolnowsky, Topics in Computational Geometry, STANC-CS-78-659, Ph.D. Thesis, Stanford University, 1978.