SLAC-PUB-1549 (Rev.)
STAN-CS-75-482
February 1975
Revised December 1975
Revised July 1976

AN ALGORITHM FOR FINDING BEST MATCHES IN LOGARITHMIC EXPECTED TIME

Jerome H. Friedman
Stanford Linear Accelerator Center
Stanford University, Stanford, Ca. 94305

Jon Louis Bentley
Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, N.C. 27514

Raphael Ari Finkel
Department of Computer Science
Stanford University, Stanford, Ca. 94305

ABSTRACT

An algorithm and data structure are presented for searching a file containing N records, each described by k real valued keys, for the m closest matches or nearest neighbors to a given query record. The computation required to organize the file is proportional to kNlogN. The expected number of records examined in each search is independent of the file size. The expected computation to perform each search is proportional to logN. Empirical evidence suggests that except for very small files, this algorithm is considerably faster than other methods.

(Submitted to ACM Transactions on Mathematical Software)

Work supported in part by U.S. Energy Research and Development Administration under contract E(043)515

The Best Match or Nearest Neighbor Problem

The best match or nearest neighbor problem applies to data files that store records with several real valued keys or attributes. The problem is to find those records in the file most similar to a query record according to some dissimilarity or distance measure. Formally, given a file of N records (each of which is described by k real valued attributes) and a dissimilarity measure D, find the m closest records to a query record (possibly not in the file) with specified attribute values.

A data file, for example, might contain information on all cities with post offices. Associated with each city is its longitude and latitude. If a letter is addressed to a town without a post office, the closest town that has a post office might be chosen as the destination.

The solution to this problem is of use in many applications. Information retrieval might involve searching a catalog for those items most similar to a given query item; each item in the file would be cataloged by numerical attributes that describe its characteristics. Classification decisions can be made by selecting prototype features from each category and finding which of these prototypes is closest to the record to be classified. Multivariate density estimation can be performed by calculating the volume about a given point containing the closest m neighbors.

Structures Used for Associative Searching

One straightforward technique for solving the best match or nearest neighbor problem is the cell method. The k-dimensional key space is divided into small, identically sized cells. A spiral search of the cells from any query record will find the best matches of that record. Although this procedure minimizes the number of records examined, it is extremely costly in space and time, especially when the dimensionality of the space is large.

Burkhard and Keller [1] and later Fukunaga and Narendra [2] describe heuristic strategies based on clustering techniques. These strategies use the triangle inequality to eliminate some of the records from consideration while searching the file. Although no calculations of expected performance are presented, simulation experiments indicate that these techniques permit a substantial fraction of the records to be eliminated from consideration.

Friedman, Baskett, and Shustek [3] describe another strategy for solving the nearest neighbor problem. It involves forming a projection of the records onto one or more keys, keeping a linear list on those keys, and searching only those records that match closely enough on one of the keys. The method is applicable to a wide variety of dissimilarity measures and does not require that they satisfy the triangle inequality. They were able to show that the expected computation required to search the file with this method is proportional to $km^{\frac{1}{k}} N^{1-\frac{1}{k}}$.

Rivest [4] shows the optimality of an algorithm due to Elias which deals with binary keys. That is, each key takes on only two values; the distance function applied is the Hamming distance.

Shamos [5] employs the Voroni diagram (a general structure for searching the plane) to the best match problem for the special case of two keys per record (two dimensions) and Euclidean distance measure. He presents two algorithms. One can search for best matches in worst case $O[(\log N)^2]$ time, after a file organization that requires storage proportional to N and computation proportional to NlogN The other algorithm can perform the search in worst case O[logN] time, after a file organization that requires both storage and computation proportional to N^2. Unfortunately, these methods have not yet been generalized to higher

dimensionalities or more general dissimilarity measures.

Finkel and Bentley [6] describe a tree structure, called the quad tree, for the storage of composite keys. It is a generalization of the binary tree for storing data on single keys. Bentley [7] develops a different generalization of the same one-dimensional structure; it is termed the k-d tree. In his article, Bentley suggests that k-d trees could be applied to the best match problem.

This paper introduces an optimized k-d tree algorithm for the problem of finding best matches. This data structure is very effective in partitioning the records in the file so that the average number of record examinations(1) involved in searching the file for best matches is quite small. This method can be applied with a wide variety of dissimilarity measures and does not require that they obey the triangle inequality. The storage required for file organization is proportional to N, while computation is proportional to kNlogN. For large files, the expected number of record examinations required for the search is shown to be independent of the file size, N. The time spent in descending the tree during the search is proportional to logN, so that the expected time required to search for best matches with this method is proportional to logN.

Definition of the k-d Tree

The k-d tree is a generalization of the simple binary tree used for sorting and searching. The k-d tree is a binary tree in which each node represents a subfile of the records in the file and a partitioning of that subfile. The root of the tree represents the entire file. Each nonterminal node has two sons or ~~successor nodes~~. These successor nodes represent the two subfiles defined by the partitioning. The terminal nodes represent mutually exclusive small subsets of the data records, which collectively form a partition of the record space. These terminal subsets of records are called buckets.

In the case of one-dimensional searching, a record is represented by a single key and a partition is defined by some value of that key. All records in a subfile with key values less than or equal to the partition value belong to the left son, while those with a larger value belong to the right son. The key variable thus becomes a discriminator for assigning records to the two subfiles.

In k dimensions, a record is represented by k keys. Any one of these can serve as the discriminator for partitioning the subfile represented by a particular node in the tree; that is, the discriminating key number can range from 1 to k. The original k-d tree proposed by Bentley [7] chooses the discriminator for each node on the basis of its level in the tree; the discriminator for each level is obtained by cycling through the keys in order. That is,

$$D = L \bmod k + 1$$

where D is the discriminating key number for level L and the root node is defined to be at level zero. The partition values are chosen to be random key values in each particular subfile.

This paper deals with choosing both the discriminator and partition value for each subfile, as wel as the bucket size, t minimize the expected cost of searching for nearest neighbors. This process yields what is termed an optimized k-d tree.

The Search Algorithm

The k-d tree data structure provides an efficient mechanism for examining only those records closest to the query record, thereby greatly reducing the computation required to find the best matches.

The search algorithm is most easily described as a recursive procedure. The argument to the procedure is the node under investigation. The first invocation passes the root of the tree as this argument. Available as a global array is the domain of that node; that is, the geometric boundaries delimiting the subfile represented by the node. The domain of the root node is defined to be plus and minus infinity on all keys. These geometric boundaries are determined by the partitions defined at the nodes above it in the tree. At each node, the partition not only divides the current subfile, but it also defines a lower or upper limit on the value of the discriminator key for each record in the two new subfiles. The accrual of these limits in the ancestors of any node defines a cell in the multidimensional record-key space containing its subfile. The volume of this cell is smaller for subfiles defined by nodes deeper in the tree. If the node under investigation is terminal, then all the records in the bucket are examined. A list of the m closest records so far encountered and their dissimilarity to the query record is always maintained as a priority queue during the search. Whenever a record is examined and found to be closer than the most distant member of this list, the list is updated. If the node under investigation is not terminal, the recursive procedure is called for the node representing the subfile on the same side of the partition as the query record. When control returns, a test is made to determine if it is necessary to consider the records on the side of the partition opposite the query record. It is necessary to consider that subfile only if the geometric boundaries delimiting those records overlap the ball centered at the query record with radius equal to the dissimilarity to the mth closest record so far encountered. This is referred to as the "bounds-overlap-ball" test. If the bounds-overlap-ball test fails, then none of the records on the opposite side of the partition can be among the m closest records to the query record. If the bounds do overlap the ball, then the records of that subtree must be considered and the procedure is called recursively for the node representing that subfile. A "ball-within-bounds" test is made before returning to determine if it is necessary to continue the search. This test determines whether the ball is entirely within the geometric domain of the node. If so, the current list of m best matches is correct for the entire file and no more records need be examined. The bounds-overlap-ball and ball-within-bounds tests are described in Appendix 1. Appendix 2 contains a detailed description of the complete search algorithm using an algorithmic notation.

The Optimized k-d Tree

The goal of the optimization is to minimize the expected number of records examined with the search algorithm. The parameters to be adjusted are the discriminating key number and partition value at each non-terminal node, and the number of records contained in each terminal bucket.

The solution to the optimization will, in general, depend upon the distribution of query records in the record key space. Usually, one has no knowledge of this distribution in advance of the queries. Thus, we seek a procedure that is independent of the distribution of queries and only uses information contained in the file records. Such a procedure will be seen to be good for all possible query distributions but will not be optimal for any particular one.

A second restriction is that the solution values for discriminating key number and partition value at any particular node depend only on the subfile represented by that node. This restriction is necessary so that the k-d tree can be defined recursively, avoiding a general binary tree optimization. Such an optimization is known to be NP-complete [8] and thus, very likely of non-polynomial time complexity.

Under these two restrictions, we can provide a prescription for choosing the discriminating key and partition value at each nonterminal node. The information provided to the search algorithm by the partitioning is the location of the partition and the identities of those records that lie on either side. It is well known that information provided to a binary choice is maximal when the two alternatives were equally likely. Thus, each record should have had equal probability of being on either side of the partition. This criterion dictates that we locate the partition at the median of the marginal distribution of key values, irrespective of which key is chosen for the discriminator.

The search algorithm can exclude searching the subfile on the opposite side of the partition to the query record if the partition does not intersect the current m-nearest neighbor ball. That is, if the distance to the partition is greater than the radius of the ball. By definition, the radius is the same along all key coordinates. Thus, the probability of the partition intersecting the ball is least (averaged over all possible query locations) for that key which exhibited the greatest spread or range in values before the partitioning.

The prescription for optimizing the k-d tree, then, is to choose at every nonterminal node the key with the largest spread in values as the discriminator, and to choose the median of the discriminator key values as the partition. The optimum number of records for each terminal bucket is developed in the next section on analysis of performance. Appendix 3 presents an algorithm that builds an optimized k-d tree according to this prescription.

Analysis of the Performance

The storage required for file organization is proportional to the file size, N. The discriminating key number and partition value must be stored for each nonterminal node of the k-d tree.[2] The number of nonterminal nodes is $\left\lceil \frac{N}{b} \right\rceil - 1$ where b is the number of records in each terminal bucket.

The computation required to build the k-d tree is easily derived. At each level of the tree, the entire set of key values must be scanned. This requires computation proportional to kN. The depth of the tree is logN, so the total computation to build the tree is proportional to kNlogN. [Here we are solving the recurrence relation $T_N = 2T_{N/2} + kN$, which is well-known to have the solution $T_N = O(kN\log N)$.]

The expected time performance of the search is not so easily derived. It is most easily discussed in a geometric framework. Let $\vec{X}_i = [X_i(1), X_i(2), \ldots X_i(k)]$ represent the set of key values for the ith record in the file. If the value of each key is plotted along a coordinate axis, then the set of key values for a record represents a point in a coordinate space of k dimensions. The entire file is a collection of such points in k-dimensional coordinate space. The query record can similarly be represented as a point, $\vec{X}_q$, in this space. The best match problem is then to find the m closest points to the query point in this space by the given dissimilarity measure.

The performance of the algorithm may depend upon the total number of records in the file, N, the dimensionality (number of keys), k, the number of nearest neighbors sought, m, the number of records in the terminal buckets, b, the dissimilarity measure $D(\vec{X},\vec{Y})$, employed, and the distribution $p(\vec{X})$ of the file records in the record key space.

Let $S_m(\vec{X}_q)$ be the smallest ball in the coordinate space centered at $\vec{X}_q$ that exactly contains the m closest points to $\vec{X}_q$. That is,

$$S_m(\vec{X}_q) \equiv \{\vec{X} \mid D(\vec{X},\vec{X}_q) \leq D(\vec{X}_q,\vec{X}_m)\} \tag{1}$$

where $\vec{X}_m$ is the mth nearest neighbor to $\vec{X}_q$. The volume $v_m(\vec{X}_q)$ of this region is

$$v_m(\vec{X}_q) = \int_{S_m(\vec{X}_q)} d\vec{X}\,, \tag{2}$$

and the probability content of this region, $u_m(\vec{X}_q)$, is defined as

$$u_m(\vec{X}_q) \equiv \int_{S_m(\vec{X}_q)} p(\vec{X})\, d\vec{X}\,, \qquad \text{with } 0 \leq u_m(X_q) \leq 1 \tag{3}$$

It can be shown [9] that the probability distribution of $u_m(\vec{X}_q)$ follows a beta distribution, B(m,N); that is,

$$p(u_m) = \frac{N}{(m-1)!(N-m)!} [u_m]^{m-1} [1-u_m]^{N-m} \tag{4}$$

independently of the probability density function of the points, $p(\vec{X})$, or the dissimilarity measure, $D(\vec{X},\vec{Y})$. The expected value of this distribution is

$$E[u_m] \equiv \int_0^1 u_m\, p[u_m]\, du_m = \frac{m}{N+1} \tag{5}$$

These results state that any compact volume enclosing exactly m points has probability content m/(N+1) on the average.

To proceed further, we assume that the file size, N, is large enough so that $S_m(\vec{X}_q)$ is small and thus the probability distribution $p(\vec{X})$ is approximately constant within the region $S_m(\vec{X}_q)$. In this case, we can approximate eqn 3 by

$$u_m(\vec{X}_q) \quad \bar{p}(\vec{X}_q)\, v_m(\vec{X}_q),$$

and from eqn 5

$$E[v_m(\vec{X}_q)] \quad \frac{m}{N+1} \; \frac{1}{\bar{p}\,(\vec{X}_q)}$$

Here $\bar{p}\,(\vec{X}_q)$ is the probability density averaged over the small region $S_m(\vec{X}_q)$. Note that it can never be zero.

Consider now the effect of the optimized k-d tree partitioning algorithm described in the previous section. Choosing the median insures that each bucket will contain very nearly b records, where b is the maximum bucket size. Choosing the key with the largest spread in values at each node insures that the geometric shape of these buckets will be reasonably compact In fact, the expected shape of these buckets is hypercubical with edge length equal to the kth root of the volume of the space occupied by the bucket. The edges are parallel to the coordinate axes The effect of the optimized k-d tree partitioning, then, is to divide the coordinate space into approximately hypercubical subregions, each containing very nearly the same number of records. From eqn 7, we have that the expected volume of such a bucket is

$$E[v_b(\vec{X}_b)] \simeq \frac{b}{N+1} \; \frac{1}{\bar{p}\,(\vec{X}_b)} \tag{8}$$

where $\vec{X}_b$ is a point that locates the bucket in the coordinate space.

Consider now the smallest k dimensional hypercube with edges parallel to the coordinate axes that completely contains the region $S_m(\vec{X}_q)$. The volume of this hypercube $V_m(\vec{X}_q)$ is proportional to $v_m(\vec{X}_q)$, with proportionality constant, G(k), depending on the dissimilarity measure and the dimensionality [10]. That is,

$$V_m(\vec{X}_q) \quad G(k)\; v_m(\vec{X}_q) \tag{9a}$$

and

$$E[V_m(\vec{X}_q)] \quad \frac{m}{N+1} \; \frac{G(k)}{\bar{p}\,(\vec{X}_q)} \tag{9b}$$

In order to calculate the average number of buckets examined by the k-d tree searching algorithm described above, it is necessary to calculate the average number of buckets, $\bar{\ell}$, overlapped by the region $S_m(\vec{X}_q)$. This number will be bounded from above by the average number of buckets, $\bar{L}$, overlapped by the smallest hypercube that contains $S_m(\vec{X}_q)$. This average number is

$$\bar{L} = [\frac{e_m(\vec{X}_q)}{e_b(\vec{X}_q)} + 1]^k . \tag{10}$$

Here, $e_m(\vec{X}_q)$ is the edge length of the hypercube containing $S_m(\vec{X}_q)$ and $e_b(\vec{X}_q)$ is the edge length of the hypercubical buckets in the neighborhood. The edge length of a hypercube is the kth root of its volume; from eqns 10, 9, and 8, we have

$$\bar{\ell} \le \bar{L} = \{[\frac{m}{b} G(k)]^{\frac{1}{k}} + 1\}^k \tag{11}$$

as an upper bound on the average number of buckets overlapped by the constant radius ball $S_m(\vec{X}_q)$. The number of records in each bucket is b, so that an upper bound on the average number of records examined, $\bar{R}$, is

$$\bar{R} \le b\bar{L} = b\{[\frac{m}{b} G(k)]^{\frac{1}{k}} + 1\}^k . \qquad (12)$$

Two important results follow from this expression. First, minimizing it with respect to b yields the result b=1; to minimize the (upper bound on the) number of records examined, the terminal buckets should each contain one record. With this provision, eqn 12 becomes

$$\bar{R} \le \{[mG(k)]^{\frac{1}{k}} + 1\}^k . \qquad (13)$$

The second important result is that the expected number of records examined is <u>independent</u> of the file size, N, and the probability distribution of the key values, $p(\vec{X})$, in the record key space.

Although derived here in a somewhat obtuse fashion, these results can be easily understood intuitively. If the goal is to minimize the accumulated coverage of all the buckets overlapped by any region, then the partitioning should be as fine as possible. This is accomplished by making each bucket as small as possible.

The independence of the number of overlapped buckets to file size and distribution of key values is a direct consequence of the prescription for optimizing k-d trees. This prescription partitions the k-dimensional record space so that each terminal bucket has the same properties as the region, $S_m(\vec{X}_q)$, containing the m best matches. Namely, each contains a fixed number of records (b and m, respectively) and their geometrical shapes are reasonably compact. As a result, the dependence of the bucket volumes on total file size and distribution of key values is identical to that for the region $S_m(\vec{X}_q)$ containing the m best matches. An the file size or the local key density increases, the bucket volumes and the volume containing the m best matches shrink at exactly the same rate, leaving the number of overlapped buckets, ℓ, constant.

The constancy of the number of records examined as file size increases implies that the time required to search for best matches is <u>logarithmic</u> in file size. The k-d tree is a balanced binary tree. Thus, the time required to descend from the root to the terminal buckets is logarithmic in the number of nodes, which is directly proportional to the file size, N. The amount of backtracking in the tree is proportional to $\bar{\ell}$, which we have demonstrated to be independent of N. Thus, the expected search time for the m best matches to a prespecified query record is proportional to logN.

<u>Dissimilarity Measures</u>

The derivations of the preceding section make no explicit assumptions concerning the particular dissimilarity measure, $D(\vec{X},\vec{Y})$, employed. There are, however, some implicit assumptions that are now discussed.

A <u>dissimilarity measure</u> is defined as

$$D(\vec{X},\vec{Y}) \equiv F\left(\sum_{i=1}^{k} f_i[X(i), Y(i)]\right), \qquad (14)$$

where the k + 1 arbitrary functions F and $\{f_i\}_{i=1}^k$, are required to satisfy the basic properties of <u>symmetry</u>

$$f_i(x,y) = f_i(y,x) \qquad 1.5 \quad i \le k \qquad (15a)$$

and <u>monotonicity</u>

$$F(x) \ge F(y) \quad \text{if } x > y \qquad (15b)$$

$$f_i(x,z) \ge f_i(x,y) \quad \text{if} \begin{cases} z \ge y \ge x \\ \text{or} \\ x \ge y \ge z \end{cases} \quad 1 \le i \le k . \qquad (15c)$$

The k functions, $\{f_i(x,y)\}_{i=1}^k$, are called the <u>coordinate distance functions</u>; they define the one-dimensional distance along each coordinate. Sins-e the spread in coordinate values is defined to be the average distance from the center, the ith coordinate distance function should be used to estimate the spread in the ith key values during the construction of the optimized k-d tree. (These coordinate distance functions also appear in the bounds-overlap-ball and ball-within-bounds tests described in Appendix 1.) To this extent, the construction of the k-d tree depends upon the particular dissimilarity measure employed. It is not necessary that exactly these functions be used in building the k-d tree. The purpose of the spread estimation is to order the key numbers. Any set of functions that yields the same ordering as the coordinate distance functions will serve just as well. For example, if the coordinate distance functions are all identical, that is, $f_i(x,y) = f(x,y)$ for $1 \le i \le k$, then the linear function $\hat{f}(x,y) = |x-y|$ can be used to estimate the spread in key values.

The properties of the dissimilarity measure enter into this algorithm directly through the bounds-overlap-ball and ball-within-bounds tests (see Appendix 1). These tests require only two properties of a dissimilarity measure. First, the dissimilarity between two points, $D(\vec{X},\vec{Y})$, must be nondecreasing with increasing linear distance, $|X(i)-Y(i)|$, along any coordinate. Second, a partial dissimilarity based on any subset of the coordinates must be less than or equal to the actual dissimilarity based on the full coordinate set. The form required for a dissimilarity measure by eqn 14, together with the restrictions of eqn 15, are sufficient to guarantee both of these properties.

A dissimilarity measure is said to be a <u>metric distance</u> if, in addition to symmetry and monotonicity (eqns 14-15c), it obeys the triangle inequality

$$D(\vec{X},\vec{Y}) + D(\vec{Y},\vec{Z}) \ge D(\vec{X},\vec{Z}). \qquad (16)$$

The most common metric distances are the vector space p-norms

$$D_p(\vec{X},\vec{Y}) = \left[\sum_{i=1}^{k} |X(i) - Y(i)|^p\right]^{\frac{1}{p}} . \qquad (17)$$

Of these, the most commonly used are:

p = 1: taxicab or city block distance

p = 2: Euclidean distance

$p = \infty$: maximum coordinate distance .

That is,

$$D_{\infty}(\vec{X},\vec{Y}) = \max_{1 \le i \le k} |X(i) - Y(i)| . \qquad (18)$$

Since the separate coordinate distance functions are identical for these distances, the linear distance function, $\hat{f}(x,y) = |x-y|$, can be used to estimate the key spreads for building the k-d tree.(3) For the special case of the $p = \infty$ distance (eqn 18), the geometric constant G(k) (eqns 9a and 13) is unity, and the inequality of eqn 13 becomes an equality. For this particular distance, we can therefore calculate the expected number of records examined (instead of an upper bound on the expected number) as a function of the number of best matches, m, and number of keys, k :

$$\bar{R}_{\infty}(m,k) = (m^{\frac{1}{k}} + 1)^k . \qquad (19)$$

Note that for m=1, $R_{\infty}(1,k) = 2^k$. The number of buckets overlapped by a ball of constant volume decreases with increasing p, so the $p = \infty$ result serves as a <u>lower bound</u> for all vector space p norms.

There is an assumption that is implicit in the results of the previous section. It is that the search algorithm examines the buckets in optimal order; that is, in order of increasing dissimilarity from the query record. It is not clear how close the k-d tree search algorithm comes to this ideal. Since this inefficiency is purely geometrical, it can be absorbed into the geometric constant, $G(k)$, in eqns 12 and 13, leaving the general conclusions unchanged. However, to the extent that this inefficiency does exist, eqn 19 is overly optimistic (as it assumes $G(k) = 1$) and thus, eqn 19 represents a lower bound even for the $p = \infty$ distance.

<u>Simulation Results</u>

Several simulations were performed to gain insight into the performance of the algorithm and to compare it to the performance predicted by eqn 19. The results are presented in Figures 1 and 2. For each simulation, a file of 8192 sets of record keys was generated from a normal distribution with unit dispersion matrix. A similar set of 2000 query record keys was generated and the number cf record examinations required to find the m best matches was averaged over these 2000 queries. The statistical uncertainty of these averages is quite small, being around two percent in the worst cases.

Figure 1 shows how the average number of record examinations required to find the best match ($m=1$) varies with dimensionality (number of keys per record). Results are shown both fcr the $p=2$ (Euclidean) and the $p=\infty$ vector space norms. The solid line represents eqn 19 which predicts the expected number for the $p = \infty$ metric ($\bar{R} = 2^k$).

The behavior of the algorithm corresponds closely to that discussed in the previous section. For low dimensionality ($k \leq 6$), the $p=\infty$ results

strongly exhibit the 2^k dependence. These simulation results indicate that, at least for $m=1$, the k-d tree search algorithm is not far from optimal. For those dimensionalities ($k \leq 6$) where $N = 8192$ appears to be big enough for the validity of the large file assumption,(4) the simulation results for $p = \infty$ lie no more than 20% above that predicted by eqn 10.

The Euclidean distance results shown in Figure 1 confirm that the performance of the algorithm for lower p-norms is not as good as for $p=\infty$. The increase in expected number of records examined is not severe, but becomes more pronounced for the higher dimensionalities. If a distance is to be chosen mainly for rapid calculation, the $p=\infty$ distance is a good choice.

Figure 2 shows how the number of records examined depends on the number of best matches sought. The average number of record examinations required to find the corresponding number of best matches fcr both the Euclidean and $p=\infty$ norms is displayed along with the prediction of eqn 19 (solid line). The average number of records examined rises with increasing number of best matches slightly more slowly than linearly. One would intuitively expect the increase to be linear since the expected volume of the m-nearest neighbor ball grows linearly with m. The average number of overlapped cells, therefore, should increase similarly. This is approximately borne out by the results shown. Figure 2 also shows that the effect of the non-optimality of the search algorithm becomes more pronounced for a larger number of best matches. If it is assumed that 8192 records is large enough so that the large file assumption is valid even for $m=25$ in four dimensions, then Figure 2a shows that the inefficiency is 18% for $m=1$ and 50% for $m=25$.

Implementation

The above discussion has centered on the expected number of records examined as the sole criterion for performance evaluation of the algorithm. This has the advantage that evaluation is independent of the details of implementation and the computer upon which the algorithm is executed. Although the computational requirements of the algorithm are strongly related to the number of records examined, there are other considerations as well. These considerations include the computation required to build the k-d tree and the overhead computation required to search the tree.

The computation required to build the k-d tree is proportional to kNlogN, as previously stated. This is illustrated empirically in Figure 3 where the actual computation (5) per record needed to build the tree is shown as a function of the total number of records for several values of k.

The overhead required to search the tree is dominated by the bounds-overlap-ball calculation. This calculation must be performed at each non-terminal node visited in the search. As described in Appendix 1, it involves calculating the dissimilarity from the query record to the closest boundary of the subfile under consideration. The coordinate distances are compared one key at a time; if the boundary is far from the test point, the subfile can be excluded quickly on the basis of only a few keys. If, on the other hand, the boundary is close to the test point, then it may be necessary to examine most or all of the keys. If the bounds do in fact overlap the ball, then all keys are included and the test becomes as expensive as a full dissimilarity calculation. This suggests that if a subfile is very likely to overlap the ball, it should simply be investigated and the bounds-overlap-ball calculation omitted. This situation is most likely to occur near the bottom of the tree where

the file records are closest to the query record. Therefore, it may be profitable to increase the bucket sizes even at the expense of increasing the number of record comparisons.

With one record per bucket, a bounds-overlap-ball calculation must be made for each file record close to the query record near the bottom of the tree. With several records per bucket, a bounds-overlap-ball calculation need only be performed once for each bucket. Since the records in a bucket are relatively close together, it is very likely that if one of them passes the test, most or all will pass. It is then more computationally efficient to have larger bucket sizes even though this increases the number of records examined.

This speculation is confirmed in Figure 4. Here the computation required for finding best matches is shown for various bucket sizes. Increasing the bucket size from one record per bucket considerably improves the performance of the search. This improvement is approximately constant for bucket sizes from 4 to 32.

Although Figure 4 shows results for only a few situations, other simulations (not shown) verify that this behavior is completely independent of dimensionality, k, number of best matches, m, and number of file records, N.

Comparison to Other Methods

The only previous method with verified expected performance for various dimensionalities, number of best matches, and number of file records is the sorting algorithm of Friedman, Baskett and Shustek [3]. This algorithm has been shown to yield a considerable improvment over the brute force method (linear search over all the records in the file) for a wide variety of situations. Figure 5 shows the computation (CPU milliseconds per query) required by this sorting algorithm and the k-d tree algorithm (using buckets of sixteen records) for increasing file size. Also shown is the average number of records examined under the k-d tree method. The rate of increase of this average with increasing file size indicates how near it is to the asymptotic limit where the large file assumption is valid. The results in Figure 5 show that in two dimensions near-asymptotic behavior occurs even for files as small as 128 records. In four dimensions, the asymptotic limit appears reasonably close for file sizes greater than 2000. In eight dimensions, the limit is not near for files of 16000 records. Even for this case, however, the increase in average number of records examined with file size is only slightly faster than logarithmic.

The logarithmic behavior of the overall computation as the file size increases is illustrated for the k-d tree algorithm in Figure 5, except that for eight dimensions the increase is slightly faster.(6) Comparison of Figure 3 to Figure 5 shows that the preprocessing computation involved in building the tree is not excessive. The fraction of computation spent on preprocessing decreases with increasing dimensionality. When the number of query records is the same as the number of file records, preprocessing represents about 25% of the total computation for two dimensions, while for eight dimensions that fraction is between three and five percent.

The computation required by the sorting algorithm has been shown [3] to be proportional to $km^{\frac{1}{k}}N^{1-\frac{1}{k}}$. Although this is much worse than logN, the sorting algorithm introduces very little overhead so that for very small files, it is faster than the k-d tree algorithm. For larger

files, however, the k-d tree algorithm is seen to have a clear computational advantage, especially for higher dimensions.(7)

Implementation on Secondary Storage

Efficient operation of the k-d tree algorithm doesnot require that all of the terminal buckets reside in fast memory. During the preprocessing, these data can be arranged on an external storage device so that records in the same bucket are stored together. Buckets close together in the tree can be stored similarly. Since the search algorithm examines a small number of buckets on the average, there will be few accesses to the external storage for each query.(8) For extremely large files, it is not even necessary that the entire k-d tree reside in fast memory. Only the top levels of the tree need to be in fast memory; the lower levels can be stored on an external device under an arrangement that keeps non-terminal nodes close to their sons.

ACKNOWLEDGMENT

Helpful discussions with F. Baskett, M.G.N. Hine, C.T. Zahn, and J.E. Zolnowsky are gratefully acknowledged.

APPENDIX 1

This appendix describes algorithms for the bounds-overlap-ball and ball-within-bounds tests discussed in the text.

The purpose of the <u>bounds-overlap-ball test</u> is to determine if the geometric boundaries delimiting a subfile of records overlap a ball centered at the query record with radius r equal to the dissimilarity to the mth closest record so far encountered. That is, $r = D(\vec{X}_m, \vec{X}_q)$ where $\vec{X}_q$ is the query record and $\vec{X}_m$ is the mth best match so far encountered in the search. The technique employed is to find the smallest dissimilarity between the bounded region and the query record. If this dissimilarity is greater than r, then the subfile can be eliminated from consideration. This minimal dissimilarity is determined as follows: if the query record's jth key is within the bounds for the jth coordinate of the geometric domain, then the jth partial distance is set to zero; otherwise it is set to the coordinate distance f_j (eqns 14, 15) by which the key falls outside the domain in that coordinate. If any of these coordinate distances is greater than the radius of the neighborhood, then there is no overlap between the domain and the neighborhood. If the sum of coordinate distances exceeds $F^{-1}(r)$ (eqn 14), there is no overlap. The test can terminate with failure as soon as the partial sum of coordinate distances exceeds $F^{-1}(r)$. In the special case of the $p=\infty$ vector space norm, this technique reduces to testing whether any of the distances is greater than the radius and, if so, failing.

The ball-within-bounds test is simpler. Here the coordinate distance from the query record to the closer boundary along each key is in turn compared to the radius, r. The test fails as soon as one of these coordinate distances is less than the radius. The test succeeds if all

of these coordinate distances are greater than the radius.

Descriptions of these tests in an algorithmic notation are presented in the next appendix.

APPENDIX 2

This appendix presents the k-d tree search algorithm in an algorithmic notation.

```
global
    Xq[1:k],          "key values of the query record"
    PQD[1:m],         "priority queue of the m closest- distances en
                       countered at. any phase of the search.  PQD[1]
                       is the distance to the mth nearest neighbor so
                       far encountered."
    PQR[1:m],         "priority queue of the record numbers of the
                       corresponding m best matches encountered at
                       any phase of the search"
    B+[1:k],          "coordinate upper bounds"
    B-[1:k],          "coordinate lower bounds"
    discriminator [1:I], "discriminator at each k-d tree node"
    partition [1:I];     "partition value at each k-d tree node"
                         "I is the number of internal nodes"
"initialize"  PQD[1:m] ← ∞; B+[1:k] ← ∞; B-[1:k] ← -∞;
"search"      SEARCH(root);
procedure SEARCH(node);
    begin
    local p,d, temp;
    if node is terminal
        begin
        (examine records in bucket(node), updating PQD, PQR);
        if BALL WITHIN BOUNDS then done else return
        end;
    d ← discriminator[node]; p ← partition[node];
```

```
"recursive call on closer son"
if X_q[d] ≤ p
then begin
    temp ← B_+[d]; B_+[d] ← p;
    SEARCH(leftson(node)); B_+[d] ← temp;
    end
else begin
    temp ← B_-[d]; B_-[d] ← p;
    SEARCH(rightson(node)); B_-[d] ← temp;
    end;
"recursive call on farther son, if necessary"
if X_q[d] ≤ p
then begin
    temp ← B_-[d]; B_-[d] ← p;
    if BOUNDS OVERLAP BALL then SEARCH(rightson(node));
    B_-[d] ← temp;
    end
else begin
    temp ← B_+[d]; B_+[d] ← p;
    if BOUNDS OVERLAP BALL then SEARCH(leftson(node));
    B_+[d] ← temp;
    end;
"see if we should return or terminate"
if BALL WITHIN BOUNDS then done else return;
end;
```



```
logical procedure BALL WITHIN BOUNDS;
    begin
    local d;
    for d ← 1 step 1 until k do
        if COORDINATE DISTANCE (d, X_q[d], B_-[d]) ≤ PQD[1]
        or COORDINATE DISTANCE (d, X_q[d], B_+[d]) ≤ PQD[1]
        then return(false);
    return(true);
    end;
logical procedure BOUNDS OVERLAP BALL;
    begin
    local sum, d;
    sum ← 0;
    for d ← 1 step 1 until k do
        if X_q[d] < B_-[d]
        then begin "lower than low boundary"
            sum ← sum + COORDINATE DISTANCE (d,X_q[d], B_-[d]);
            if DISSIM(sum) > PQD[1] then return (true);
            end
        else if X_q[d] > B_+[d]
        then begin "higher than high boundary"
            sum ← sum + COORDINATE DISTANCE (d,X_q[d], B_+[d]);
            if DISSIM(sum) > PQD[1] then return (true);
            end;
    return (false);
    end;
```

The procedures DISSIM (x) and COORDINATE DISTANCE (j,x,y) are the functions F(x) and $f_j(x,y)$ that appear in the definition of the dissimilarity measure (eqn 14).

APPENDIX 3

This appendix presents a description in an algorithmic notation of the procedure for constructing an optimized k-d tree for best match file searching.

```
root ← BUILD TREE (entire file);
node procedure BUILD TREE (file);
    begin
    local j,d, maxspread, p;
    if SIZE(subfile) ≤ b then return(MAKE TERMINAL(file));
    maxspread ← 0;
    for j ← 1 step 1 until k do "find coordinate with greatest spread"
        if SPREADEST(j,file) > maxspread
        then begin
            maxspread ← SPREADEST(j,file);
            d ← j;
            end;
        end;
    p ← MEDIAN(d,file);
    return MAKE NONTERMINAL(d,p,BUILDTREE(LEFT SUBFILE(d,p,file)),BUILDTREE
        (RIGHTSUBFILE(d,p,file));
    end;
```

The procedure SPREADEST (j,subfile) returns the estimated jth key value spread for the records in the subfile represented by the node, using the ωth coordinate distance function The procedure MEDIAN (j,subfile) returns the median of the jth key values. MAKE TERMINAL and MAKE NONTERMINAL are procedures that store their parameters as values of a node in the k-d tree and return a pointer to that node.

FOOTNOTES

(1) A record involves fetching the record keys from memory, calculating the dissimilarity to the query record, comparing it to the dissimilarity to the mth closest record so far encountered, and if necessary, updating the list of m closest records.

(2) Since the k-d tree is a complete binary tree, it is not necessary to store pointers to the sons of each nonterminal node [11].

(3) The spread of values along each key can be estimated by calculating the trimmed variance of the key values. The trimming insures that the estimate is robust against extreme outliers.

(4) Asymptotic behavior can be determined empirically by observing the rate of increase of the average number of records examined with increasing file size. This is illustrated in Figure 5.

(5) All simulations were performed on an IBM 370/168 computer. All programs were coded in FORTRAN IV and compiled with the IBM FORTRAN x (extended) compiler with optimization level two.

(6) The behavior for eight dimensions will, of course, become logarithmic for large enough file sizes.

(7) The comparison in Figure 5 is for the best match (m=1) since this is the most common application. The increase in computation for larger m grows as $m^{\frac{1}{k}}$ for the sorting algorithm, while for k-d tree algorithm, it grows nearly linearly with m. Thus, for large numbers of best matches, the crossover file size at which the performance of the two algorithms is comparable will increase.

(8) Inspection of Figure 5 shows that for bucket size of 16 records, the average number of buckets accessed is 1.56, 6.25 and 75.0 for two, four, and eight dimensions, respectively, for total file size of 16000 records. Increasing the bucket size to 32 records (not shown) reduces the average number of accesses for eight dimensions to 44.0 while increasing the total computation required for the search by only 8%.

REFERENCES

[1] Burkhard, W.A. and Keller, R.M. Some approaches to best match file searching. Com. of ACM, Vol. 16 (April 1973), 230-236.

[2] Fukunaga, K., and Narendra, P.M. A Branch and bound algorithm for computing k-nearest neighbors. IEEE Trans. Comput., C24 (1975), 750-753.

[3] Friedman, J.H., Baskett, F., and Shustek, L.J. An algorithm for finding nearest neighbors. IEEE Trans. Comput., C-24(1975) 1000-1006.

[4] Rivest, R. On the optimality of Elias' algorithm for performing best match searches. Proceedings IFIP Congress 74, Stockholm, Sweden (August 1974), 678-681.

[5] Shamos, M.I. Computational Geometry. Conference record of Seventh Annual ACM Symposium of Theory of Computing, Albuquerque, N.M., (May 7, 1975).

[6] Finkel, R.A. and Bentley, J.L. Quad trees - a data structure for retrieval on composite keys. Acta Informatica 4(1)(1974),1-9.

[7] Bentley, J.L. Multidimensional binary search trees used for associative searching. Com. of ACM, Vol.18 (Sept.1975), 509-517.

[8] Hyafil, L., and Rivest, R.L., Constructing optimal binary decision trees is NP-complete. Information Processing Letters, Vol. 5, (May 1976), 15-17.

[9] Fukunaga, K., and Hostetler, L.D., Optimization of k-nearest neighbor density estimates. IEEE Trans. Info. Theory, IT-19 (1973), 320-326.

[10] Pizer, S.M., Numerical Computing and Mathematical Analysis, Science Research Associates, Palo Alto, Ca., 1975, pp 88, eqn 87.

[11] Knuth, D.E., The Art of Computer Programming, Vol. 1, Addison-Wesley, Menlo Park, Ca., 1969, p 401.

FIGURE CAPTIONS

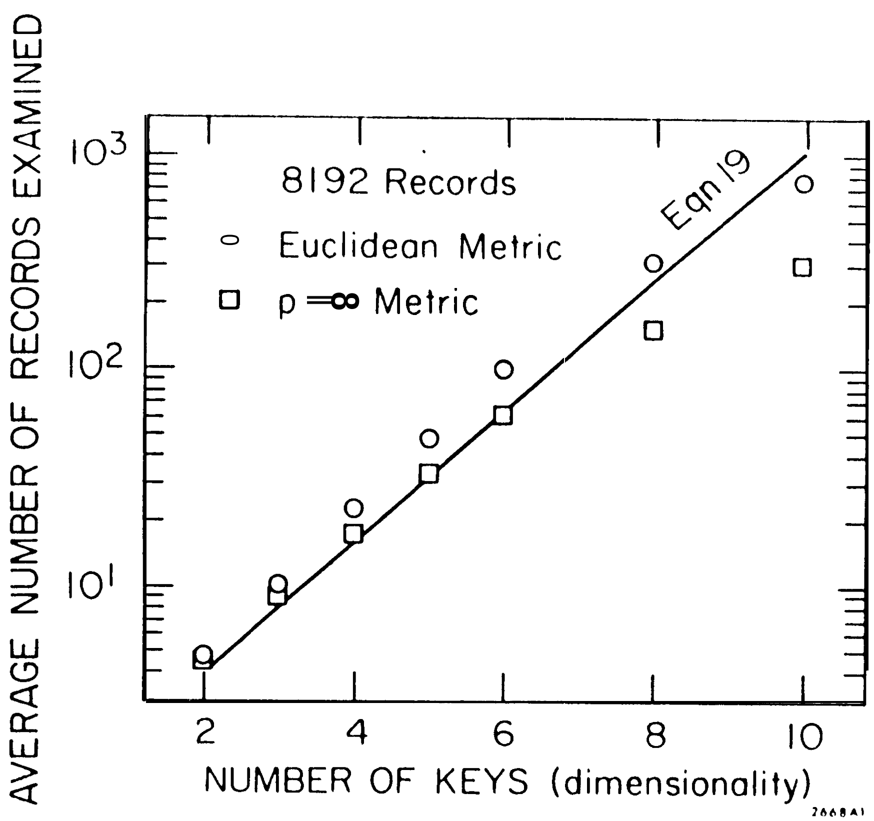
FIGURE Variation of the average number of records examined with dimensionality (number of keys per record) for constant file size. Results are shown for the Euclidean ($p=2$) and $p=\infty$ metrics. The solid line is the prediction of eqn 19 for the $p=\infty$ metric.

FIGURE 2 Variation of the average number of records examined with number of best matches sought for several dimensionalities. Results are shown for the Euclidean ($p=2$) and $p=\infty$ metrics. The solid lines are the predictions of eqn 19 for the $p=\infty$ metric.

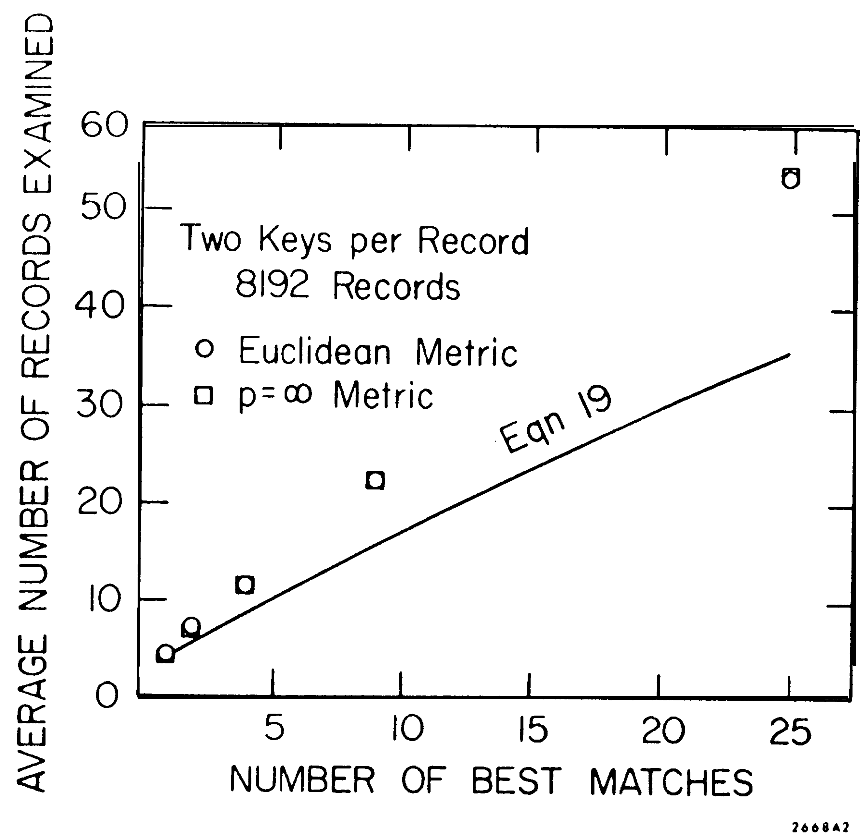
FIGURE 3 Computation per file record required to build the k-d tree as a function of total file size for several dimensionalities.

FIGURE 4 Computation required for the best match search as a function of terminal bucket size.

FIGURE Computation required for best match searching as a function of total file size for both the sorting and k-d tree algorithms at several dimensionalities. Also shown is the variation of the average number of records examined with total file size. Terminal buckets of 16 records were used with the k-d tree algorithm.


AVERAGE NUMBER OF RECORDS EXAMINED
10^3
10^2
10^1
8192 Records
Euclidean Metric
p =∞ Metric
Eqn 19
2
4
6
8
10
NUMBER OF KEYS (dimensionality)
2668A1

Figure


AVERAGE NUMBER OF RECORDS EXAMINED
60
50
40
30
20
10
0
Two Keys per Record
8192 Records
Euclidean Metric
p=∞ Metric
Eqn 19
5
10
15
20
25
NUMBER OF BEST MATCHES
2668A2

2a

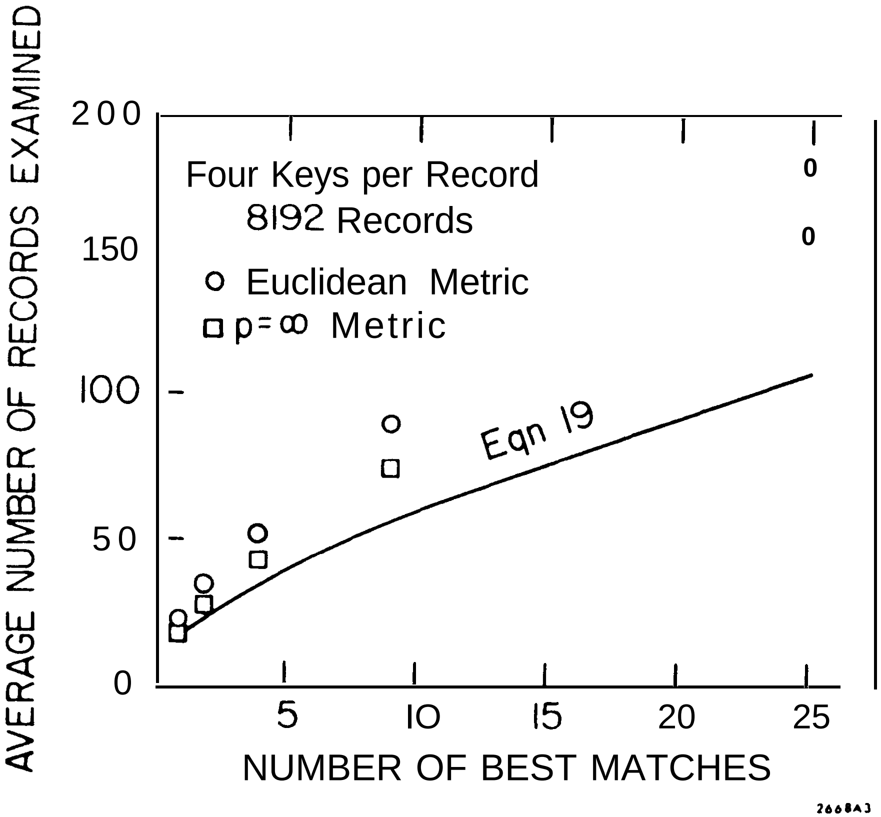


Figure 2b

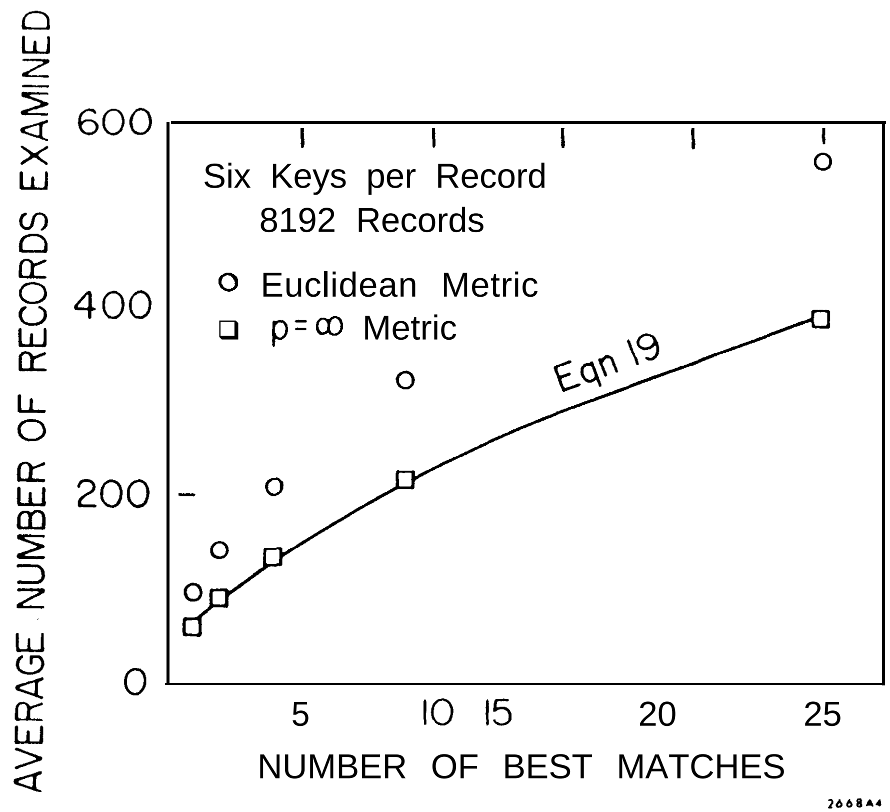


Figure 2c

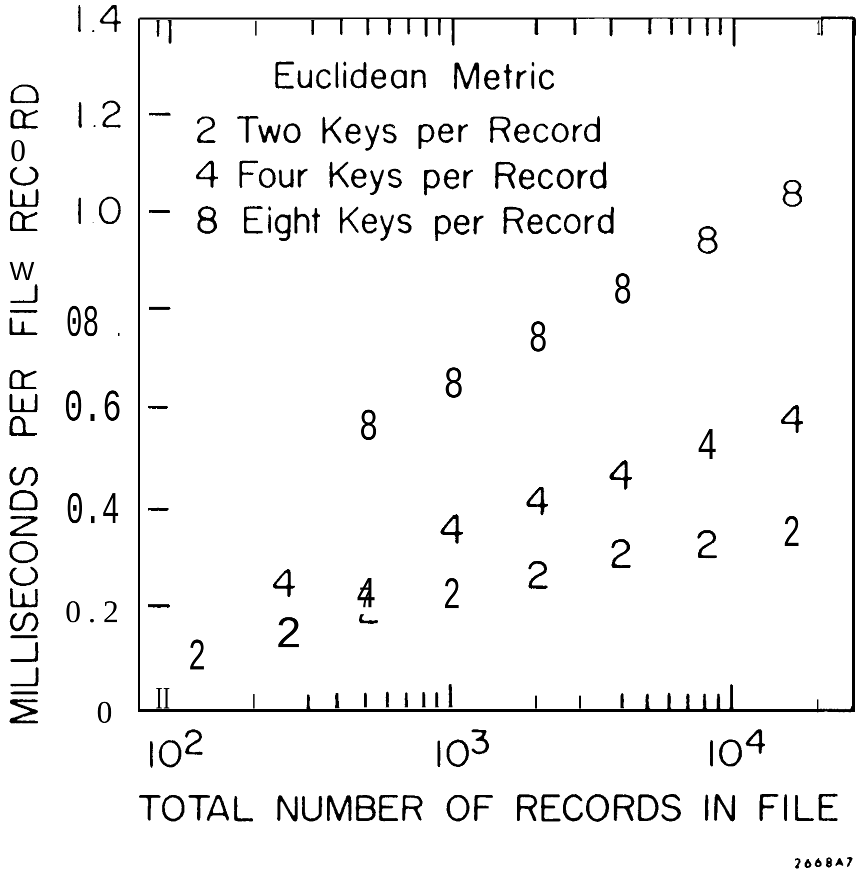


Figure 3

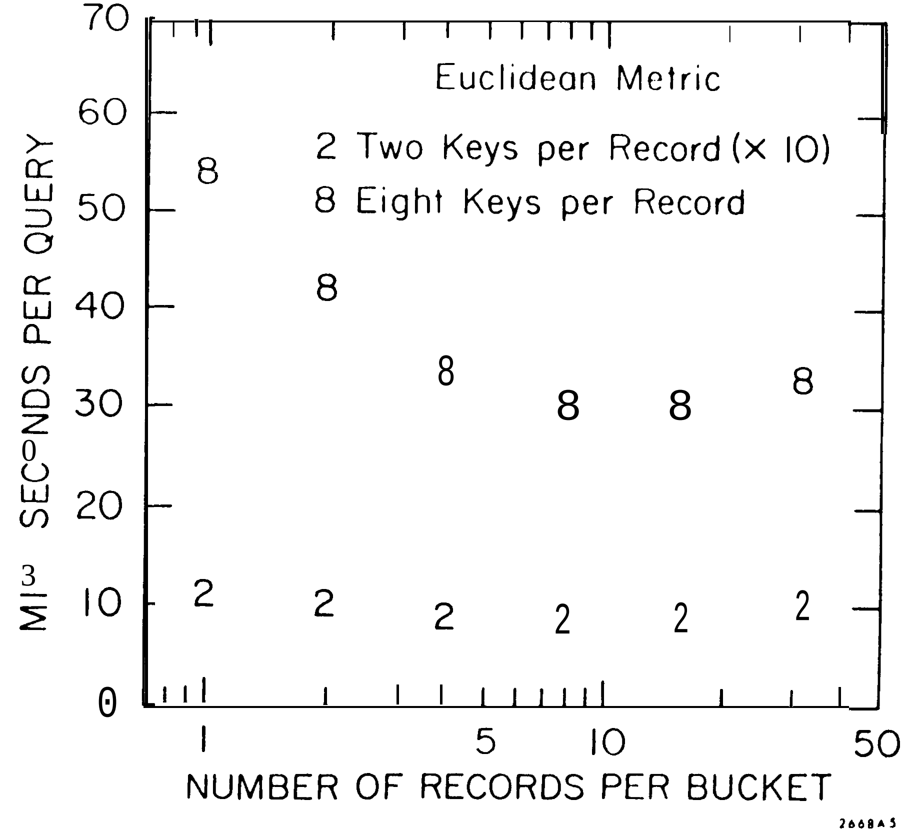


Figure 4

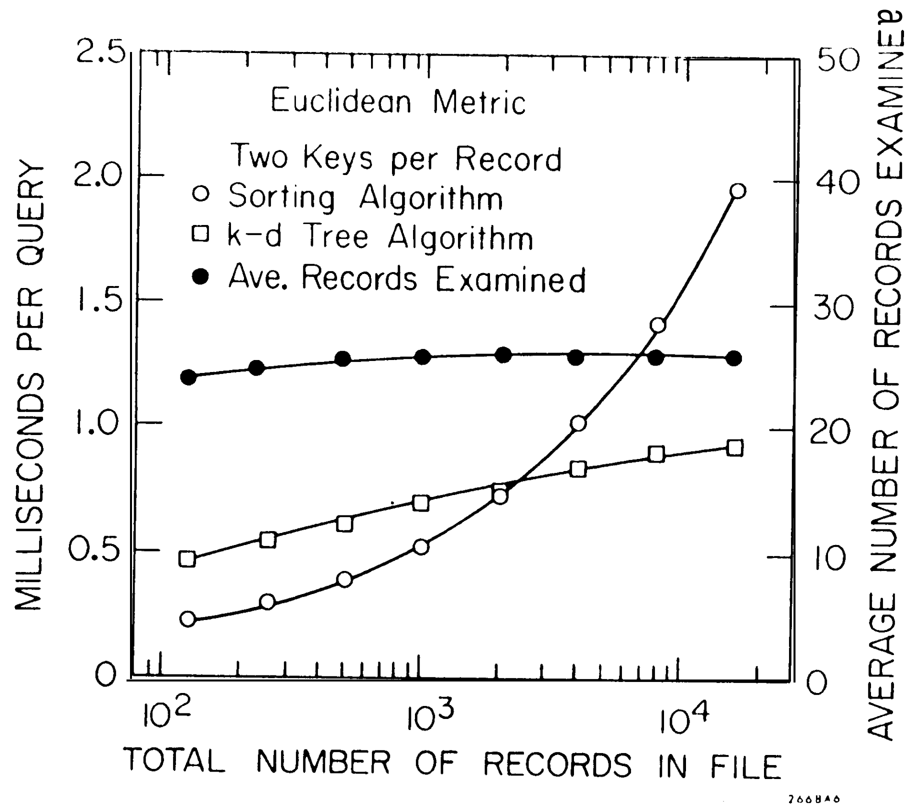


Figure 5a

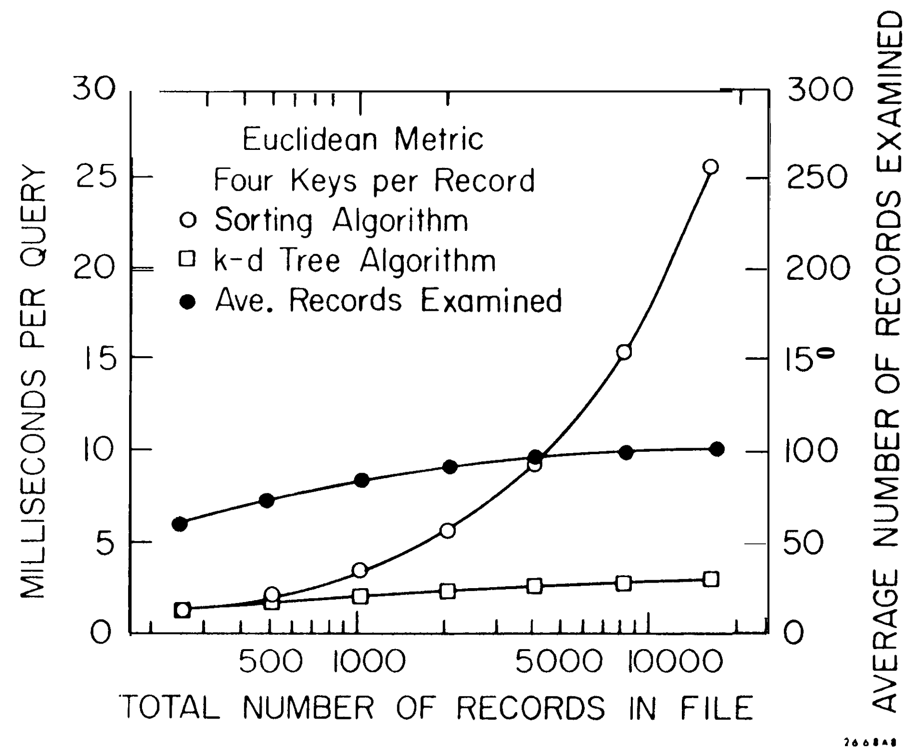
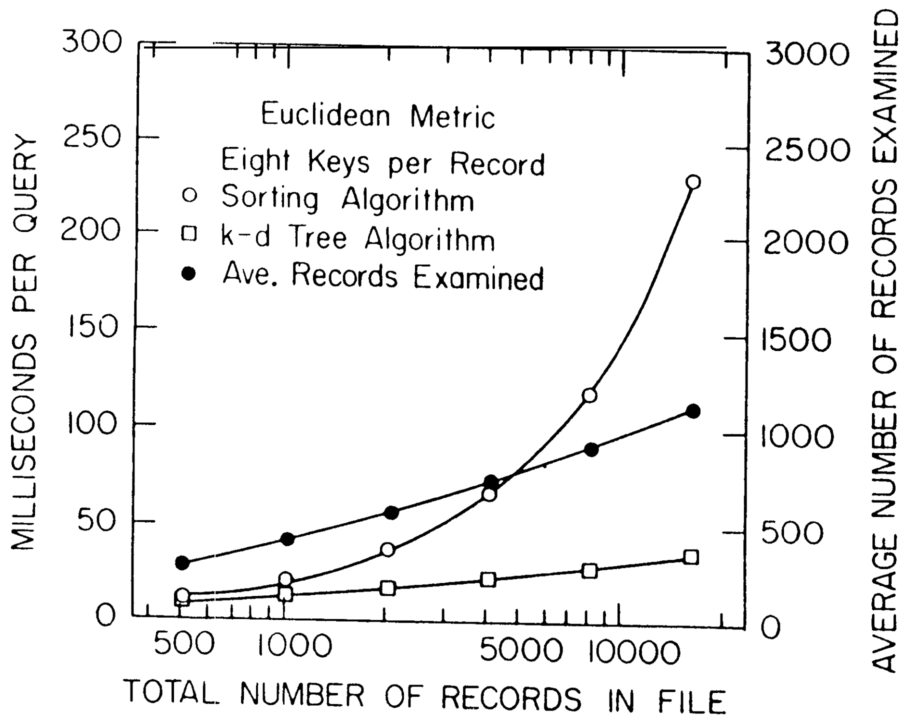


Figure 5b



MILLISECONDS PER QUERY
300
250
200
150
100
50
0
Euclidean Metric
Eight Keys per Record
Sorting Algorithm
k-d Tree Algorithm
Ave. Records Examined
3000
2500
2000
1500
1000
500
0
AVERAGE NUMBER OF RECORDS EXAMINED
500
1000
5000
10000
TOTAL NUMBER OF RECORDS IN FILE

Figure 5c