The best match of nearest neighbor problem

ABSTRACT

In large databases, especially when the dimensionality of the space is large, the complexity of the space increases. The number of records grows exponentially. Although specific search techniques have been developed to handle this problem, the k-nearest neighbor problem is still a challenging one. The k-nearest neighbor query is the best match of nearest neighbors. Efficient indexing techniques are needed to perform the best match of nearest neighbors.

Structures used for associative searching:

- Finding the volume score of a given point and computing the closest neighbors.
- Searching for the nearest neighbors can be performed by a dynamic and static method. When the number of records grows, the query to be processed can involve a large number of records. Each query to be processed in the file would be processed easier to be answered.

The solution to this problem is to use many applications. In this application, the user can select the best office or the closest to it from the user's location. If a query is addressed to a given office, the user's location with the office. Associated with each office is a list of neighbors, where each neighbor is a neighbor file. For each neighbor, each neighbor is presented for searching.

The best match of nearest neighbor problem

ACKNOWLEDGMENTS

Stemmer University, Stanford, CA. 94305
Department of Computer Science

Deborah Miller, N.C. 28734
Department of North Carolina at Chapel Hill

Don L. Smith

Stemmer University, Stanford, CA. 94305
Department of Computer Science

In Association for Computing Machinery

Received July 1976
Received December 1976
Pamex'sc75
SAN-DC-77-085
(REV.)
The search algorithm

The search algorithm is used to find a record in a B-tree. The search process involves the following steps:

1. Start at the root of the tree.
2. Compare the search key with the key in the root node.
3. If the search key is equal to the root key, the search is successful and the record is returned.
4. If the search key is less than the root key, repeat the search on the left subtree.
5. If the search key is greater than the root key, repeat the search on the right subtree.
6. If the search key is not found in the tree, the search is unsuccessful.

The B-tree is a balanced tree, meaning that all leaf nodes are at the same level, which ensures efficient search times. Each node in the tree contains a maximum of a specific number of key values, and each node points to zero or more children. This structure allows for fast insertion and deletion operations as well.

The B-tree is particularly useful for managing large datasets, as it can efficiently handle additions, deletions, and searches, even as the dataset grows.
Consider that 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.

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 search algorithm can exclude examining the subtree on the opposite side of the partition 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 the opposite side of the partition to the query record. If the bounds-overlap-ball test fails, then none of the records on the opposite side of the partition to the query record can be defined recursively, avoiding a general binary tree optimization. Such an optimization is known to be NP-complete.

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 have no knowledge of the geometric domain of the k-d tree. The search algorithm will, in general, depend upon this geometric domain. 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.

The search algorithm can exclude searching the subtree on the opposite side of the partition if the partition does not intersect the query record. If so, the current list of m best matches is correct for the 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 have no knowledge of the geometric domain of the k-d tree. The search algorithm will, in general, depend upon this geometric domain.

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.

The search algorithm can exclude searching the subtree on the opposite side of the partition if the partition does not intersect the query record. If so, the current list of m best matches is correct for the 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 have no knowledge of the geometric domain of the k-d tree. The search algorithm will, in general, depend upon this geometric domain.

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.

The search algorithm can exclude searching the subtree on the opposite side of the partition if the partition does not intersect the query record. If so, the current list of m best matches is correct for the 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.
Find the closest points to the query point in the space by the given set of records. The exact algorithm is:

1. Compute the distance between the query point and each record in the space.
2. Sort the distances in ascending order.
3. Return the closest records.

The performance of the algorithm depends on the total number of records and the distribution of the data. In practice, a good partitioning of records can significantly improve the performance by reducing the number of comparisons needed.

A geometric framework is used to partition the data into smaller regions for efficient searching. The framework is based on a geometric algorithm that partition a space of a given dimension into a collection of hyperrectangles. Each hyperrectangle is defined by a set of hyperplanes that split the space into smaller regions.

The expected time performance of the search is not easily derived, but it is well-known to have a solution of O(N log N).

where N is the total number of records and k is the number of nearest neighbors to find.
in order to calculate the average number of buckets examined by

\[
\frac{b}{\lambda} = \frac{(b)^{\lambda}}{(\lambda)^{\lambda}} N^{\lambda - 1} \left( \frac{b}{\lambda} \right)^{\lambda}
\]

and

\[
\frac{b}{\lambda} = \frac{b}{\lambda} \times \left( \frac{b}{\lambda} \right)^{-1}
\]

The probability of collision depends on the distribution of records. In the case of a uniform distribution, the probability of collision is proportional to the number of records in the bucket. Consider now the expected value of a random variable with a given distribution.

where \( b \) is a point that locates the bucket in the coordinate space.

\[
\left( \frac{\lambda}{\lambda} \right)^{\lambda} N^{\lambda - 1} \left( \frac{b}{\lambda} \right)^{\lambda}
\]

and

\[
\left( \frac{\lambda}{\lambda} \right)^{\lambda} N^{\lambda - 1} \left( \frac{b}{\lambda} \right)^{\lambda}
\]

The probability density of the collision is proportional to the number of records in the bucket. In this case, the probability density is proportional to the number of records in the bucket. The expected number of collisions is therefore

\[
\left( \frac{\lambda}{\lambda} \right)^{\lambda} N^{\lambda - 1} \left( \frac{b}{\lambda} \right)^{\lambda}
\]

and from this, we can approximate the expected value of the distribution.

\[
\left( \frac{\lambda}{\lambda} \right)^{\lambda} N^{\lambda - 1} \left( \frac{b}{\lambda} \right)^{\lambda}
\]
Two important results follow from this expression. First, minimizing it with respect to \( b \) yields the result that minimizes the number of records examined, the terminal buckets should each contain one record. With this provision, eqn (12) can be interpreted as implying that the number of records examined as file size increases implies that the time required to search for best matches is logarithmic 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 records stored in the record key space. Although derived here in a somewhat obtuse fashion, these results are quite intuitive. If the goal is to minimize the accumulated coverage of all the buckets overlapped by any region, then the number of overlapped buckets and the volume containing the m best matches are directly proportional to the size of the key value. The independence of the number of overlapped buckets to the size of the key value is a direct consequence of the prescription for partitioning the k-dimensional record space so that each terminal bucket has the same properties as the region 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 volume on total file size and distribution of key values is constant.

Although derived here in a somewhat obtuse fashion, these results are quite intuitive. If the goal is to minimize the accumulated coverage of all the buckets overlapped by any region, the terminal buckets should each contain one record, and the probability density function of the expected number of records examined as the file size increases.

The second important result is that the expected number of records examined is independent of the file size, \( N \), and the probability distribution of the key values, \( p(3, \ldots) \).

Although derived here in a somewhat obtuse fashion, these results are quite intuitive. If the goal is to minimize the accumulated coverage of all the buckets overlapped by any region, the terminal buckets should each contain one record, and the probability density function of the expected number of records examined as the file size increases.

The second important result is that the expected number of records examined is independent of the file size, \( N \), and the probability distribution of the key values, \( p(3, \ldots) \).
The most common vector space distance is the vector space-prune.

The formula for the Euclidean distance between two vectors \( x \) and \( y \) in a vector space is:

\[
\| x - y \| = \left( \sum_{i=1}^{d} (x_i - y_i)^2 \right)^{1/2}
\]

where \( d \) is the dimension of the vector space.
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 cannot be corrected by reordering the search algorithm. However, for those distance metrics (e.g., Euclidean) for which the opposite is true, the k-d tree search algorithm is not far from optimal. These simulation results illustrate the 2-dimensional behavior of the algorithm.

Simulation Results

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 of 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 for the p=2 (Euclidean) and the p=\infty (maximum) norm. The Euclidean result, which predicts the expected number for the p=2 metric (\( \mathbb{E} = \frac{1}{2} \)), is also shown. The number of best matches sought increases with increasing dimensionality, as does the corresponding number of records examined.

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 for 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 slightly more slowly than linearly. This is intuitively expected, as the expected number of overlapped cells, which should increase similarly, also increases more slowly than linearly. The volume of the m-nearest neighbor ball grows exponentially with m, the average number of overlapped cells, which would therefore increase faster than linearly.

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=2, then Figure 2a shows that the inefficiency is 18% for m=1 and 52% for m=25.
This situation is most likely to occur near the bottom of the tree where the actual dissimilarity calculation.

This calculation must be performed on only a few keys. It is very likely that if one of the test points is close to the nearest point, the subtree can be excluded quickly on the basis of only a few tests. Therefore, it may be beneficial to search the tree in the order of decreasing dissimilarity.

The overhead required to search the tree is dominated by the bounds-overlap-ball calculation.

The overhead overhead to search the tree is dominated by the bounds-overlap-ball calculation. This calculation must be performed on a few keys. If one of the test points is close to the nearest point, the subtree can be excluded quickly on the basis of only a few tests. Therefore, it may be beneficial to search the tree in the order of decreasing dissimilarity.

The overhead required to search the tree is dominated by the bounds-overlap-ball calculation. This calculation must be performed on a few keys. If one of the test points is close to the nearest point, the subtree can be excluded quickly on the basis of only a few tests. Therefore, it may be beneficial to search the tree in the order of decreasing dissimilarity.

The overhead required to search the tree is dominated by the bounds-overlap-ball calculation. This calculation must be performed on a few keys. If one of the test points is close to the nearest point, the subtree can be excluded quickly on the basis of only a few tests. Therefore, it may be beneficial to search the tree in the order of decreasing dissimilarity.

The overhead required to search the tree is dominated by the bounds-overlap-ball calculation. This calculation must be performed on a few keys. If one of the test points is close to the nearest point, the subtree can be excluded quickly on the basis of only a few tests. Therefore, it may be beneficial to search the tree in the order of decreasing dissimilarity.

The overhead required to search the tree is dominated by the bounds-overlap-ball calculation. This calculation must be performed on a few keys. If one of the test points is close to the nearest point, the subtree can be excluded quickly on the basis of only a few tests. Therefore, it may be beneficial to search the tree in the order of decreasing dissimilarity.

The overhead required to search the tree is dominated by the bounds-overlap-ball calculation. This calculation must be performed on a few keys. If one of the test points is close to the nearest point, the subtree can be excluded quickly on the basis of only a few tests. Therefore, it may be beneficial to search the tree in the order of decreasing dissimilarity.
milliseconds per query) required by this sorting algorithm and the k-d tree algorithm (using buckets of sixteen records) for increasing file size. 

A~SC shown is the average number of records examined under the k-d tree algorithm. 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 size 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.

Comparison of Figure 2 to Figure 5 shows that the preprocessing computation involved in building the tree is not excessive. The fraction of the total computation represented by the k-d tree algorithm in Figure 5, except that for eight dimensions the increase is slightly faster.

Implementation on Secondary Storage

As mentioned earlier, the k-d tree 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.

In secondary storage, the k-d tree algorithm requires 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 all of the terminal buckets, any decrease in the number of accesses to the external storage device is significant. In secondary storage, the average number of records examined with the k-d tree algorithm is only slightly greater than 2000. In eight dimensions, however, the k-d tree algorithm appears inferior. The results in Figure 5 show that the asymptotic limit is not near for files as small as 16000 records. Even for this case, however, the increase in average number of records examined with file size is only slightly faster than logarithmic.

Implementation on Secondary Storage

The k-d tree algorithm is seen to have a clear computational advantage, especially for higher dimensions.
This appendix describes algorithms for the bounds-overlap-ball and ball-within-bounds tests discussed in the text.

APPENDIX 1

The purpose of the bounds-overlap-ball test is to determine if the geometric boundaries delimiting a subfile of records overlap a ball centered at the query record and 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 \( f_j \) 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 \( F^{-1}(r) \), there is no overlap. The test can terminate with failure as soon as one of these 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 in turn is compared to the radius, r. The test fails as soon as one of these coordinate distances exceeds the radius. The test succeeds if all coordinate distances are greater than the radius.

APPENDIX 2

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

```
procedure SEARCH(node);
begin
  if node is terminal then
    (examine records in bucket(node), updating PQD, R@;
    if BALL WITHIN BOUND then done else return
  d := discriminator[node];
  p := partition[node];
  if d > p then
    SEARCH(node.left);
    if BALL WITHIN BOUND then done else return
  else
    SEARCH(node.right);
    if BALL WITHIN BOUND then done else return
end.
```

The procedure \( \text{SEARCH}(\text{node}) \) is a recursive algorithm used to search the k-d tree for the m closest records. The algorithm works by traversing the tree, examining the records in each bucket, and updating the priority queue of the m nearest neighbors encountered so far. The search begins at the root of the tree and recursively examines each subtree, checking if the distance to the boundary along each coordinate is less than the radius. If so, the search continues in that subtree; otherwise, the search moves to the other subtree. The algorithm terminates if all coordinate distances exceed the radius or if the m nearest neighbors have been found.

The k-d tree is a binary search tree where each node contains a partition value and a discriminator for each coordinate. The partition value is used to determine which subtree to search next, and the discriminator is used to determine the boundary along each coordinate. The algorithm uses a priority queue \( \text{PQD} \) to store the m nearest neighbors encountered so far, along with their priority numbers \( \text{R@} \).

The purpose of the bounds-overlap-ball test is to determine if the geometric boundaries delimiting a subfile of records overlap a ball centered at the query record.

APPENDIX 1

This appendix describes algorithms for the bounds-overlap-ball and ball-within-bounds tests discussed in the text.
recursive call on closer son
if Xq[d] < B+C[d] then
  temp + B-C[d];
  if OVERLAP BALL then SEARCH(rightson(node));
  B-C[d] + temp;
else
  temp - B-C[d];
  if OVERLAP BALL then SEARCH(rightson(node));
  B-C[d] - temp;
end

recursive call on farther son, if necessary
if Xq[d] > B+C[d] then
  if OVERLAP BALL then SEARCH(rightson(node));
  B-C[d] - temp;
else
  if OVERLAP BALL then SEARCH(rightson(node));
  B-C[d] + temp;
end

we should return or terminate
if BALL WITHIN BOUNDS then done else return
the two algorithms are comparable with increasing
best matches, the considered time slice at which the performance of
three, it is shown that the parameters for a, b, and c are above
important because of its nature, and that for c, the mean above
the most common application: The increase in the 4th
the comparison in the 3rd for the best match (e.g. show this to
more for large numbers of time.

the breakpoint for one dimension with, of course, becoming larger
read the output with optimization level zero.

program were coded in FORTRAN and compiled with the IBM
All statements were performed on the IBM 7090/02 computer. All
	with reasonable time slice. The is illustrated in Figure 5a.
the large number of the squares, rectangular or nearly
applicable to other can be determined empirically by observation.
where the output is correlated in normal operational conditions.
involves the estimation of the largest the

4.1 To store 4;...4; the result of 
5.1 To store n + 1;...4; the result of 
6.1 To store n + 1;...4; the result of
7.1 To store n + 1;...4; the result of
8.1 To store n + 1;...4; the result of
9.1 To store n + 1;...4; the result of
10.1 To store n + 1;...4; the result of
11.1 To store n + 1;...4; the result of

This appendix presents a description in an illustrative notation of

functions P(x) and N(x,y) that appear in the definition of the distance
the processes (x) and coordinate distance (x) are

PROOFS
AVERAGE NUMBER OF RECORDS EXAMINED

NUMBER OF KEYS (dimensionality)

10^1 10^2 10^3

8192 Records

p = ∞ Metric

Eqn 19

AVERAGE NUMBER OF RECORDS EXAMINED

NUMBER OF BEST MATCHES

20 30 40 50 60

8192 Records

p = ∞ Metric

Eqn 19

Two Keys per Record
Figure 2a

Figure 2b

NUMBER OF BEST MATCHES

10 15 20 25

600 400 200 0

AVERAGE NUMBER OF RECORDS EXAMINED

Six Keys per Record

Eight Keys per Record

Euclidean Metric

P = oo Metric

Cornell University

August 19 8192 Records

NUMBER OF BEST MATCHES

10 15 20 25

600 400 200 0

AVERAGE NUMBER OF RECORDS EXAMINED

Four Keys per Record

Eight Keys per Record

Euclidean Metric

P = oo Metric

Cornell University

August 19 8192 Records
Figure 3a

Figure 3b