Query Processing for Distance Metrics*

Tsong-Li Wang, wang@cs.nyu.edu
Dennis Shasha, shasha@cs.nyu.edu
Courant Institute of Mathematical Sciences, New York University

Abstract

In applications such as vision and molecular biology, a common problem is to find the similar objects to a given target (according to some distance measure) in a large database. This paper presents a scheme for query processing in such situations. The basic strategy is to (partially) precompute inter-object distances, and by using the distance information and the triangle inequality, we eliminate the need to calculate certain object distances while evaluating queries. We propose several heuristics that may speed up query evaluation. A series of experiments are then performed to evaluate the effectiveness of our scheme and the relative performance of the heuristics for different queries. Finally we investigate the possibility of parallelizing our scheme through simulation. Our results show that parallelism is best applied in the later stages in evaluating a query.

1 Introduction

Recently, a significant body of research has been performed for query optimization in object-oriented database systems. Most of the research has been concentrating on retrieving multidimensional [15, 22, 23, 27] or complex objects [4, 10, 14, 25, 31, 32, 34] often arising in spatial and VLSI/CAD applications. Here, we study a different class of queries, namely, to find the similar or dissimilar objects to a given target from a database. The similarity of two objects is defined in terms of a distance measure. Vision and molecular biology have many such applications where the objects are patterns [17], strings [18], trees [26], graphs [5, 8] and so forth.

Many distance measures are used in these applications. For our purposes, we assume that we have a true distance metric, that is, a function d that takes pairs of objects into nonnegative numbers, satisfying the following three properties: for any objects O_1, O_2, O_3, $d(O_1, O_2) \geq 0$, and $d(O_1, O_2) = 0$ iff $O_1 = O_2$ (non-negative definiteness); $d(O_1, O_2) = d(O_2, O_1)$ (symmetry); $d(O_1, O_2) \leq d(O_1, O_3) + d(O_3, O_2)$ (triangle inequality).

The queries we are concerned with are categorized as follows: Given a target T and a database $\mathcal{D}$ of objects,

- (Type 1 query) find the k objects, for some k, in $\mathcal{D}$ that are closest to T;
- (Type 2 query) find the closest (i.e. most similar) object of T in $\mathcal{D}$;[1]
- (Type 3 query) find the objects in $\mathcal{D}$ that are sufficiently similar to T, i.e. those that are within some distance, say ϵ, of T;
- (Type 4 query) find the k objects in $\mathcal{D}$ that are farthest from T;
- (Type 5 query) find the farthest (i.e. most dissimilar) object of T in $\mathcal{D}$;
- (Type 6 query) find the objects in $\mathcal{D}$ that are sufficiently dissimilar to T, i.e. those that are beyond distance ϵ of T.

To answer these queries, a query system could compute the distance between each object of the database and the target, and then search for the desired objects. The major problem with this approach is its computational expense, particularly when there are many targets to be identified and the distance computation is costly.[2] It is our goal to minimize such computation in response to queries of the above categories. (Our objective in this aspect is similar to [15, 22, 27], where efforts were devoted to saving the overlapping computation of spatial objects.)

*Work supported in part by NSF grant IRI-8901699 and by ONR contract N00014-85-K-0046.

Proceedings of the 16th VLDB Conference
Brisbane, Australia 1990

[1]This query is a special case for the type 1 query where $k = 1$. The latter retrieves not only the closest object, but the ith, $i = 2, \ldots, k$, closest object of T in $\mathcal{D}$.

[2]Throughout, we shall assume that distance computation is the dominant cost for query processing. As evidence to support this assumption, we note that it can take several seconds, on the average, to compare even one string against another on a VAX system [18].

Numerous techniques have been proposed for handling such queries in the past. Investigators in artificial intelligence have formulated strategies for the type 1 query [13, 17], though they are mainly interested in specific distance metrics, usually Euclidean, and make stronger assumptions such as that similar objects can be grouped into a cluster. The type 2 query, also known as the best-match retrieval, was first discussed by Minsky and Papert [21], and has been studied extensively in many areas including information retrieval [6, 11, 29], molecular biology [18, 20] and computational geometry [19]. See [30] for a survey on the methods appeared previously. The type 3 query, also known as the good-match retrieval, was discussed by Ito and Kizawa [16], where they organized a database into a hierarchical structure and used both best-match and good-match processes to retrieve similar strings for spelling correction applications. This type of query is particularly important when objects are likely to be corrupted (e.g. images in a noisy environment) [9, 28]. The queries falling in the last three categories, in particular the type 5 query (worst-match retrieval) and the type 6 query (bad-match retrieval), appear to be less common. We suspect their use will appear with the advent of new technology and new applications.

We present in this paper a scheme to answering all six types of queries. The work is an extension of [30], where we focused on the best-match retrieval. Our overall strategy is to (partially) precompute inter-object distances, and by using the preprocessed information and the triangle inequality, we eliminate certain irrelevant distance calculations while evaluating the queries. Our approach differs from previously published techniques, many of which rely on organizing files into some structure, in that we allow the optimum use of *any* given set of precomputed information. The motivation for considering starting with arbitrarily preprocessed information is that at times one may be given a set of distances which have been calculated, but one was not able to choose which distances. In Section 2, we use a Floyd-Warshall [12, 33] style algorithm to approximate those absent distances and develop our scheme, using the type 1 query as the running example. In Section 3 all the other queries are studied. Section 4 shows extensions to various important join operations. Experimental results are reported in Section 5. We make concluding remarks in Section 6.

2 The Scheme

Let $\mathcal{D}$ be a database composed of n objects.[3] For convenience, we assume these objects are numbered from 1 to n and refer to the object numbered i simply as O_i. Consider the situation where we are given a set of distances that were calculated beforehand. We proceed in two phases when evaluating a query:

- First, estimate those absent distances and store the distance values (including estimated ones and exact ones) in an *approximate distance map* (ADM).
- Then process the query based on the ADM, filtering out as many objects as possible that could not possibly satisfy the query.

[3]The objects we consider are rather general and could be any ones on which distance functions can be defined.

If all the inter-object distances are precomputed, phase 1 is omitted. In Section 2.1, we discuss how to obtain an ADM. Section 2.2 shows how it can be used to process queries, and outlines a number of heuristics for expediting the process. The type 1 query is used as the running example. Extensions to the other queries will be discussed in Section 3.

2.1 Approximate Distance Map

To compute the ADM, we start by constructing a weighted undirected graph on $\mathcal{D}$, such that there is an edge between O_i and O_j iff $d(O_i,\ O_j)$ has been computed. If there is such an edge e, its weight, denoted $w(e)$, is the computed distance. We define a path from $O_i = O_{i_1}$ to $O_j = O_{i_n}$ as a sequence of distinct objects $O_{i_1}, O_{i_2}, \ldots, O_{i_n}$ such that $\{O_{i_1}, O_{i_2}\}, \{O_{i_2}, O_{i_3}\}, \ldots, \{O_{i_{n-1}}, O_{i_n}\}$ are edges in the graph and the weight of the path is the sum of weights of its constituent edges.

Lemma 1. *(Generalized Triangle Inequality) Suppose there is a path P from O_i to O_j. Let $\hat{e}$ be the edge of maximum weight in P. Then*

$$d(O_i, O_j) \geq w(\hat{e}) - \sum_{e \in P - \{\hat{e}\}} w(e).$$

Proof: By induction on the number of objects in P and repeated application of the triangle inequality. □

Lemma 1 states that one can obtain a lower bound for $d(O_i,\ O_j)$ by applying the triangle inequality to a path from O_i to O_j. Of course, such a bound is useless if the term on the right hand side of the inequality is less than or equal to 0. Generally, we want this bound to be as high as possible. Let $P(i,j)$ be the set containing all paths from O_i to O_j. We define $ADM[i,\ j]$ to be the maximum bound obtained from all paths in $P(i,j)$. By the triangle inequality, $ADM[i,j] = d(O_i,\ O_j)$ if edge $\{O_i,\ O_j\} \in P(i,j)$.

It is impractical, in general, to enumerate all paths in $P(i,j)$ to get $ADM[i,j]$, because there may be an exponential number of them. Instead, we use a dynamic programming technique similar to the transitive closure algorithm [33] to compute the ADM. To facilitate the computation, we also maintain an additional matrix MIN, where $MIN[i,j]$ is the minimum weight of any path from O_i to O_j. Thus, $MIN[i,j]$ gives the least upper bound of the distance between O_i and O_j. Clearly, $MIN[i,\ j] = d(O_i,\ O_j)$ when the value is computed.

Following [3], let $ADM_k[i,j]$ (resp. $MIN_k[i,j]$), $0 \leq k \leq n$, be the greatest lower bound (resp. least upper bound) of any path from O_i to O_j that does not pass through an object numbered higher than k.

Lemma 2. *Let $S_k(i,j)$, $1 \le k \le n$, denote the set of paths going from O_i to O_k and then from O_k to O_j, without passing through an object numbered higher than k. Suppose $S_k(i,j) \neq \emptyset$. Let $B_k(i,j)$ be the greatest lower bound obtained by applying the generalized triangle inequality to all the paths in $S_k(i,j)$. Then*

$$B_k(i,j) = \max \begin{cases} ADM_{k-1}[i,k] - MIN_{k-1}[k,j] \\ ADM_{k-1}[j,k] - MIN_{k-1}[k,i] \end{cases}$$

$$\textit{for } 1 \le k \le n$$

Proof: Let $P \in S_k(i,j)$ be a path yielding $B_k(i,j)$. Let P_1 be the segment of P between O_i and O_k, and P_2 be the segment of P between O_k and O_j. Suppose first that the edge $\hat{e}$ of maximum weight is in P_1.
By Lemma 1, we get

$$B_k(i,j) = w(\hat{e}) - \sum_{e \in P_1 - \{\hat{e}\}} w(e) - \sum_{e \in P_2} w(e).$$

Claim that

$$ADM_{k-1}[i,k] = w(\hat{e}) - \sum_{e \in P_1 - \{\hat{e}\}} w(e).$$

By induction,

$$ADM_{k-1}[i,k] \ge w(\hat{e}) - \sum_{e \in P_1 - \{\hat{e}\}} w(e),$$

and if inequality held, we could construct a path P' in $S_k(i,j)$ by concatenating a path P_1', which yields $ADM_{k-1}[i,k]$, and P_2. The bound achieved by P' would be greater than $B_k(i,j)$, contradicting the definition of $B_k(i,j)$. By an analogous argument,

$$MIN_{k-1}[k,j] = \sum_{e \in P_2} w(e).$$

Thus, $B_k(i,j) = ADM_{k-1}[i,k]$ - $MIN_{k-1}[k,j]$.
If $\hat{e}$ is in P_2, symmetric arguments yield $B_k(i,j) = ADM_{k-1}[j,k]$ - $MIN_{k-1}[k,i]$. □

From the above lemma, we have, for each k,

$$ADM_k[i,j] = \max \begin{cases} ADM_{k-1}[i,j] \\ ADM_{k-1}[i,k] - MIN_{k-1}[k,j] \\ ADM_{k-1}[j,k] - MIN_{k-1}[k,i] \end{cases}$$

Moreover [3],

$$MIN_k[i,j] = \min \begin{cases} MIN_{k-1}[i,j] \\ MIN_{k-1}[i,k] + MIN_{k-1}[k,j] \end{cases}$$

These formulae give rise to a Floyd-Warshall style algorithm for computing the approximate distance map. The procedure is given in Figure 1.

```
for $i$ := 1 to $n$ do
  for $j$ := 1 to $i$ - 1 do
    if $d(O_i, O_j)$ is known then begin
      $ADM[i,j] := d(O_i,O_j)$; $MIN[i,j] := d(O_i,O_j)$
    end
    else begin
      $ADM[i,j] := 0$; $MIN[i,j] := \infty$
    end;
for $k$ := 1 to $n$ do
  for $i$ := 2 to $n$ do
    for $j$ := 1 to $i - 1$ do begin
      $ADM[i,j]$ := max ($ADM[i,j]$,
                        $ADM[i,k] - MIN[k,j]$,
                        $ADM[k,j]$ - $MIN[i,k]$);
      $MIN[i,j]$ := min ($MIN[i,j]$,
                        $MIN[i,k] + MIN[k,j]$);
    end;
```

Figure 1: Algorithm *APPROXIMATE*.

Notice that due to the symmetry, we only compute the lower triangular part of the matrices. Also, in the algorithm,

$$ADM[i,k] = \begin{cases} ADM[i,k] & \text{if } i \ge k \\ ADM[k,i] & \text{otherwise} \end{cases}$$

This holds for MIN as well.
Using induction on k, we obtain

Theorem 1. *Algorithm APPROXIMATE correctly computes matrices ADM and MIN; that is, it achieves the optimal distance approximation in the sense that the lower (resp. upper) bound of any path going from O_i to O_j is less (resp. greater) than or equal to ADM[i, j] (resp. MIN[i, j]), given the distances that have been computed.*

2.2 Processing Queries Using An ADM

While evaluating a query, we augment the ADM with an additional row, row $n+1$, for the target T (i.e. treating T as O_{n+1}), with entry $ADM[n+1, i]$ being the current greatest lower bound for $d(T, O_i)$.[4] We proceed in stages to seek the answer for the query. At each stage an object O is chosen and its distance from T is calculated. Each stage eliminates objects whose lower bounds are greater than the distance from T to its current kth closest object.

Following [29, 30], define ξ = the distance from T to its current kth closest object, B = the set of the current k closest objects, I = the set of candidates (i.e. objects that haven't been eliminated, nor been computed), and the function update (B, O), which tests whether $d(T, O)$

[4] We shall discuss how to update such an augmented ADM in Section 2.3. For now let us assume that this map can somehow be maintained.

$< \xi$ and, if so, updates B and ξ, discarding the object from B that becomes now the $k+1$th closest object to T. Figure 2 gives the algorithm.[5]

```
1.  $\xi := \infty$; $B := \emptyset$; $I := \mathcal{D}$;
    initialize $ADM$ and $MIN$ as done in Figure 1;
    augment $ADM$ and $MIN$ with an additional row
    for object $T$;

    /* Choose the first $k$ closest objects */
2.  Arbitrarily select $k$ objects from $I$, calculating their
    distances from $T$; delete them from $I$; put them in
    $B$; update the augmented $ADM$ and $MIN$; set $\xi$ to
    the maximum of these distances;
    while $I \neq \emptyset$
3.    pick an object $O$ in $I$;
4.    update $(B, O)$;
5.    update the augmented $ADM$ and $MIN$;
6.    $I := \{O_i \mid (d(T, O_i)$ is not computed)
                $\wedge (ADM[n+1, i] < \xi)\}$;
    end;
```

Figure 2: Algorithm 1.

We have developed four heuristics for picking candidates (i.e. objects that are still in I) at each stage (step 3, Figure 2). They use different criterion in picking objects.

Pick least lower bound
Choose an object O_i such that the lower bound of the distance between O_i and the given target T is minimized based on all previous candidates (i.e. $ADM[n+1,i] \leq ADM[n+1,j]$, $\forall O_j \in I$). Intuitively this heuristic uses the lower bound to estimate the exact distance. Thus the object having the least lower bound is expected to be (potentially) the closest object to T. If several candidates have the same lower bound, the heuristic selects one that has the least upper bound (i.e. the one with the smallest MIN value). The reason for doing so is that we expect the smaller the difference between the lower and upper bounds, the more precise the estimated distance is. Ties on the difference are resolved arbitrarily.

Pick greatest lower bound
This heuristic is similar to *pick least lower bound* except that the candidate with the greatest lower bound is selected.

Pick least upper bound
Choose a candidate with the smallest MIN value. If ties occur, choose the one with the greatest lower bound.

Pick greatest upper bound
This heuristic is similar to *pick least upper bound* except that the candidate with the greatest upper bound is selected. We expect the heuristic performs well when searching for dissimilar objects, but poorly when searching for similar objects to T.

In certain applications one may consider incorporating other factors related to objects into the heuristics presented above. For example, a hybrid heuristic for strings that takes into account the length of strings may yield estimators that help achieve better performance. But since not all distance metrics have such information associated with them, we don't pursue the topic of estimators.

In Section 5 we perform experiments to evaluate the relative performance of these heuristics for different queries. Our results confirm the expectation that *pick greatest upper bound*is generally better for finding dissimilar objects, but *pick least lower bound* is best for finding similar objects. It should be noted that starting with an optimal approximate distance map (Theorem 1), the algorithm developed here is the best possible for evaluating the concerned queries in the sense that given an object at stage i, it throws out *all* the objects that can be inferred to be irrelevant to the answers at that stage. What may influence the performance of the algorithm is the heuristic utilized in selecting objects at each stage – the better the heuristic (or the better our luck), the better performance the algorithm achieves.

2.3 Updating Augmented ADM and MIN

Each computation of the distance between T and some object O_k may lead to modifications of the augmented ADM and MIN. Observe that the value of $d(T, O_k)$ affects only the paths going through $\{T, O_k\}$. Let L (resp. U) be the new lower (resp. upper) bound of the paths from O_i to O_j *via* $\{T, O_k\}$; similarly to Lemma 2, we obtain

$$L = \max \left\{ \begin{array}{l} d(T,O_k) - MIN[i,n+1] - MIN[k,j] \\ d(T,O_k) - MIN[i,k] - MIN[j,n+1] \\ ADM[i,n+1] - d(T,O_k) - MIN[k,j] \\ ADM[i,k] - d(T,O_k) - MIN[n+1,j] \\ ADM[j,k] - d(T,O_k) - MIN[i,n+1] \\ ADM[n+1,j] - d(T,O_k) - MIN[i,k] \end{array} \right.$$

and

$$U = \min \left\{ \begin{array}{l} MIN[i,n+1] + d(T,O_k) + MIN[k,j] \\ MIN[i,k] + d(T,O_k) + MIN[n+1,j] \end{array} \right.$$

Thus, after computing $d(T, O_k)$, to find the new (tighter) bounds for the distances between objects O_i, $O_j \in \{T\} \cup \mathcal{D}$, it suffices to compare $ADM[i,j]$ ($MIN[i,j]$) with L (U) (recall that $ADM[n+1, i]$ always gives the current greatest lower bound for $d(T, O_i)$).

[5]To keep the presentation concise, we assume that objects have distinct distances from T. Relaxing this assumption, nevertheless, only requires a slight modification of the presented algorithm.

Note that we update only the pairs whose distances are still unknown. For those pairs of objects whose distances have been calculated, the distance values already represent both the best lower bounds and upper bounds, and hence they need not be modified. Calculating L and U takes only constant time. Thus the overhead incurred by updating a map is negligible when most inter-object distances are present.

If, however, there exist a large portion of object pairs whose distances are absent, the recomputation would be quite expensive. In such a situation, we could update the bounds for pairs (T, O_i), $O_i \in \mathcal{D}$, while keeping the initial bounds for (O_i, O_j), $O_i, O_j \in \mathcal{D}$ (this strategy is similar to the one suggested in [1] for maintaining shortest paths in a sizable graph), or could only update the bounds for pairs (T, O), where O is still a candidate. In [30], both the updating policies have been shown empirically to be very competitive to the one that globally updates the bounds for all object pairs (including the target), yet saving a significant amount of computation time.

3 Extensions to Other Queries

We can apply Algorithm 1 to all the other queries by slightly modifying some definitions and steps in it. The modifications for each type of query are listed below. (Note: In the following, step i refers to that in Figure 2.)

Type 2 query (best-match)

ξ = the current minimum distance to T;
B = the set of the current closest object;
update(B, O): tests whether $d(T, O) < \xi$ and, if so, updates B and ξ;

Step 2: Arbitrarily choose an object and compute its distance from T; delete it from I; put it in B; set ξ to the distance;

Type 3 query (good-match)

B = the set of objects that are within distance ϵ of T;
update(B, O): tests whether $d(T,O) < \epsilon$ and, if so, inserts O to B;

Step 6: $I := \{O_i \mid (d(T,O_i)$ is not computed$) \wedge (ADM[n+1,i] < \epsilon)\}$

Note that for this query, it becomes pointless to maintain ξ, the distance from T to its current kth closest object; step 2 is deleted.

Type 4 query (k-farthest match)

ξ = the distance from T to its current kth farthest object;
B = the set of the current k farthest objects;
update(B, O): tests whether $d(T, O) > \xi$ and, if so, updates B and ξ, discarding the object from B that becomes now the $k+1$th farthest object from T;

Step 2: set ξ to the minimum distance of the first k computed objects;
Step 6: $I := \{O_i \mid (d(T, O_i)$ is not computed$) \wedge (MIN[n+1, i] > \xi)\}$;

For this query, rather than consider lower bounds, we eliminate objects whose upper bounds are less than or equal to ξ at each stage.

Type 5 query (worst-match)

ξ = the current maximum distance to T;
B = the set of the current farthest object;
update(B, O): tests whether $d(T, O) > \xi$ and, if so, updates B and ξ;

Step 2: Arbitrarily select an object from I, calculating its distance from T; delete it from I; put it in B; set ξ to the distance;
Step 6: $I := \{O_i \mid (d(T, O_i)$ is not computed$) \wedge (MIN[n+1, i] > \xi)\}$;

As in the previous case, objects whose upper bounds are less than or equal to ξ are eliminated at each stage.

Type 6 query (bad-match)

B = the set of objects that are beyond distance ϵ of T;
update(B, O): tests whether $d(T,O) > \epsilon$ and, if so, inserts O into B;

Step 6: $I := \{O_i \mid (d(T, O_i)$ is not computed$) \wedge (MIN[n+1, i] > \epsilon)\}$;

As in the good-match retrieval, step 2 is deleted and we do not maintain the current maximum distance to T.

4 Join Operations

We may be given two sets of objects and want to find the most similar (or most dissimilar) objects between the two sets. This is a generalization of the relational join operations. In this section, we introduce two such kinds of operations. Some easy variants are also listed, with their algorithms being omitted.

4.1 Clustering Join

Given two sets of objects, R and S, the operator clustering join finds the pair (r, s) such that r is a member of R, s is a member of S, and r and s are closest. (The best-match retrieval is a special case in which one of the sets contains a single object.)

It is not difficult to extend our approach to support clustering join. To begin with, let us denote the ADM for R (S) as ADM_R (ADM_S), and MIN for R (S) as MIN_R (MIN_S). We combine ADM_R and ADM_S to form a matrix ADM, initializing all entries that are neither in ADM_R nor in ADM_S to 0. Construct and initialize the matrix MIN similarly. We then proceed in a way analogous to Algorithm 1; the code is given below. (The set B now contains the closest pair (r, s), $r \in R$, $s \in S$.)

```
1.  $\xi := \infty$; $B := \emptyset$;
    for each object $T$ in the smaller set, say $R$, do
    begin
2.    $I := \{O \mid (O \in S) \wedge (ADM[T,O] < \xi)\}$;
      while $I \neq \emptyset$
3.      pick an object $O$ in $I$;
4.      update $(B, O)$;
5.      update $ADM$ and $MIN$ globally (or partially,
        as suggested in Section 2.3);
6.      $I := \{O \mid (O \in S) \wedge (d(T, O)$ is not computed)
              $\wedge (ADM[T,O] < \xi)\}$
      end;
    end;
```

Figure 3: Algorithm Clustering Join. .

Step 3 picks an object according to the heuristics developed in Section 2.2, namely picks those with the extreme ADM (or MIN) value.

Variants:

- For each object in R, find its k-closest (farthest) objects in S.
- For each object in R, find its best (worst) matching object in S.

4.2 Radius Join

This operator is much like clustering join except that pairs of objects from R and S which are within distance ϵ are chosen. One can implement the operator by slightly modifying the algorithm for clustering join, replacing ξ by ϵ.

Variants:

- Find pairs of objects from R and S that are beyond distance ϵ.
- For each object in R, find its good (bad) matching objects in S.

5 Performance Analysis

A series of experiments were performed to evaluate the effectiveness of our scheme, its behavior when executed in a multiprocessor environment, and the relative performance of the proposed heuristics for different queries. Table 1 shows the basic parameters used in the experiments.

Parameter	Meaning	Value
$NumPE$	# of processors employed	1
$Size$	# of objects in the file	150
$Density$	Portion of known distances in the map	1
$MinDistance$	Minimum distance between objects	0
$MaxDistance$	Maximum distance between objects	10,000

Table 1: Experimental Parameters.

[$MinDistance$, $MaxDistance$] specifies the range over which distances between objects are distributed. The $Density$ parameter represents the portion of known distances in a map, and is computed by dividing the number of object pairs with known distances by the total number of object pairs in the corresponding database. The metric that we used in comparing different heuristics was

$$PERFO = \frac{NumComputed}{Size} \times 100\%$$

where $NumComputed$ is the number of objects actually computed. $PERFO$ stands for PERcentage of brute FOrce cost (i.e. the cost of computing all objects in the database). One would like this percentage to be as low as possible.

The sample maps used in the experiments were synthesized as follows. We used a random-number generator to produce inter-object distance values for each pair of objects, where the values were distributed uniformly over some positive interval. Each such value was inserted into a $(Size + 1) \times (Size + 1)$ auxiliary map provided that it didn't violate the triangle inequality. After generating the map, we randomly selected $Density \times Size \times (Size - 1)$ entries and used them as precomputed inter-object distances. Entries on the outermost row (column) of the auxiliary map represented distances between the target and objects in the database. In each experiment 30 maps were tested and the average was computed.

For many of the experiments the base values for parameters were as shown in Table 1. The distance range was arbitrarily chosen, since empirically it was found to have little effect on the performance of our scheme. To keep the analysis tractable, $Size$ was set to 150. Nevertheless our experimental results showed that the larger a database, the more effective our scheme became.

Table 2 summarizes the four heuristics proposed in Section 2, and provides their abbreviations which we will use when referring to them. For comparison purposes, the heuristic which picks an object randomly was also included. We present the results for the queries only, omitting those for join operations, since they essentially lead to similar conclusions.

Abbreviation	Heuristic
LLB	Pick least lower bound
GLB	Pick greatest lower bound
LUB	Pick least upper bound
GUB	Pick greatest upper bound
PR	Pick random

Table 2: Summary of Heuristics.

5.1 Results for Generated Data

In this experiment we examined the relative performance of the five heuristics for different queries. For types 1 and 4 queries, k was set to 3; for types 3 and 6 queries, ϵ was set to 1,000. The results are shown in Figures 4(a) - 4(f).

Examining these graphs, we see that LLB performs best for types 1 and 2 and GUB behaves poorly for these queries (Figures 4(a) and 4(b)); however the opposite is true for types 4 and 5 queries (Figures 4(d) and 4(e)). These results are consistent with our intuition: the heuristic using the least lower bound (greatest upper bound) to estimate exact distances catches closest (farthest) objects sooner; as a result the process can be completed sooner. Interestingly, PR seems to be the second best for all these queries.

From Figures 4(c) and 4(f), it can be seen that, for types 3 and 6 queries, all the heuristics have almost identical performance. This happens because for these queries, the number of calculations eliminated depends mainly on ϵ, not on how one picks an object. Notice also that $PERFO$ is very high for the bad-match retrieval. This is due to the fact that the chosen ϵ is rather small, as compared to the entire distance range, and hence few objects are beyond this distance from T.

Figure 5 graphs the number of remaining candidates against stages elapsed for various queries, showing how objects are eliminated when our scheme proceeds. LLB was used for retrieving similar objects and GUB used for retrieving dissimilar objects. The parameters had the values shown in Table 1.

We see from the figure that except for the bad-match retrieval, the curves drop sharply in the first ten stages, becoming smoother afterwards. One expects this because as time goes on, most objects whose lower (upper) bounds are greater (less) than ξ have been eliminated earlier; consequently fewer objects are actually discarded in the later stages. The curve for the bad-match retrieval is nearly a straight line. This is understandable, given the previous analysis that almost no objects, on the average, can be eliminated for this query.

5.2 Results for Proteins

In real applications we would not expect distances between objects to distribute as uniformly as those studied in the previous section. To understand how effective our scheme is when applied to actual database systems, we have run it on a set of proteins. 151 proteins were randomly selected from the sequence database of Thinking Machines. Each protein has between 4 and 20 amino acids. (An amino acid is represented by a numerical or alphabetical character.) The inter-protein distances were computed based on the *dayhoff* score metric [18].[6]

With different heuristics, we ran our scheme on these proteins thirty times, each time using a randomly selected (distinct) protein as the target. The results for k-closest match and best-match queries are shown in Figures 6(a) and 6(b), where k was set to 3.

Comparing Figures 4 and 6, the values of $PERFO$ obtained from the proteins are in general higher (i.e. worse) than those from artificial data. Second, it can be seen that LUB becomes superior to PR and is very competitive to LLB. A close look at the data reveals why this happens. In this sample database, there are lots of small clusters, proteins in which are close to one another. All the other proteins are (roughly) equally distinct from each other (and from those in the clusters). Thus, when targets are members of clusters, LLB (LUB) can quickly locate the desired proteins, yielding a fairly low $PERFO$, whereas a non-member target results in many proteins being computed.[7] This result indicates that in cases where there exist objects extremely close to a given target, picking a candidate with the least bound, whether based on ADM or MIN, is always better than arbitrarily picking one.

One interesting finding is that for the type 1 query, unlike uniformly distributed data, the performance of the heuristics is rather sensitive to the value of k. For example, when k is 6, few proteins can be eliminated, regardless of which heuristic is employed. This happens because all clusters contain less than 6 proteins; thus in order to get the 6th closest protein, one needs to compute almost all proteins in the database.

The values of $PERFOR$ for the k-farthest match and the worst-match are also much higher than those from generated data, being over 97% for all heuristics even when the entire map is precomputed. This is understandable, given the observation that most proteins are equally distant from the target, forcing our scheme to compute all of them in order to get the farthest ones.

The results for the good-match and bad-match queries

[6]The dayhoff score metric differs from, albeit is isomorphic to, a distance metric in the sense that the higher the score between two proteins, the closer they are. We used the following formula to compute the distance between two proteins based on their scores: $d(p_1, p_2) = c - s(p_1, p_2)$, where c is an empirical constant assuring that the difference satisfies the conditions of distance metrics, and $s(p_1, p_2)$ is the score between proteins p_1 and p_2.

[7]For the clustered proteins, our scheme for processing the type 1 query might be improved by first picking centers of k clusters and then using LLB (LUB), where the *center* of a cluster C is the protein p of minimum *eccentricity*, and the eccentricity of p is the distance between p and a protein in C that is farthest from p.

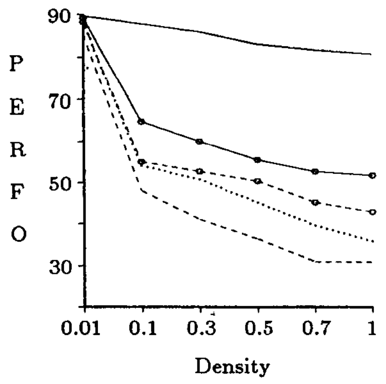

Figure 4(a): k-closest match.

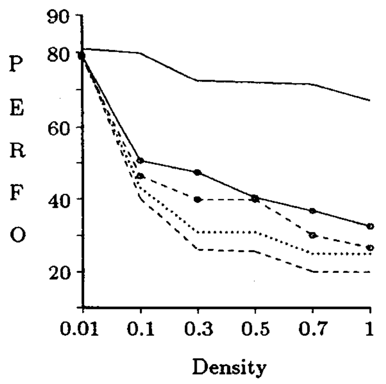

Figure 4(b): best-match.

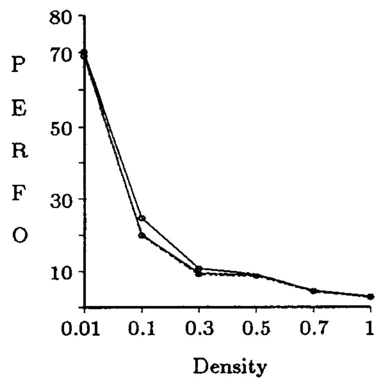

Figure 4(c): good-match.

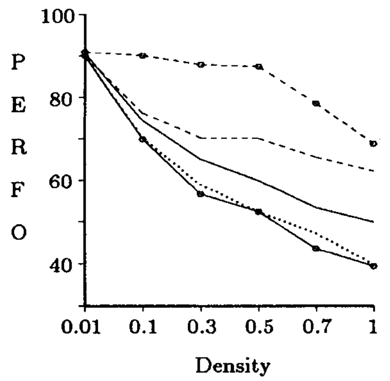

Figure 4(d): k-farthest match.

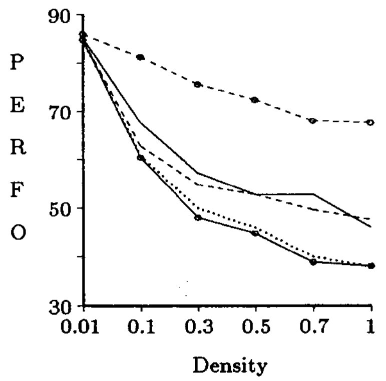

Figure 4(e): worst-match.

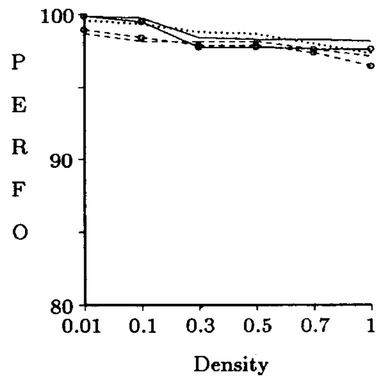

Figure 4(f): bad-match.

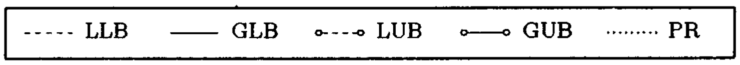

Figure 4: Comparison of Heuristics for Various Queries (Artificial Data).

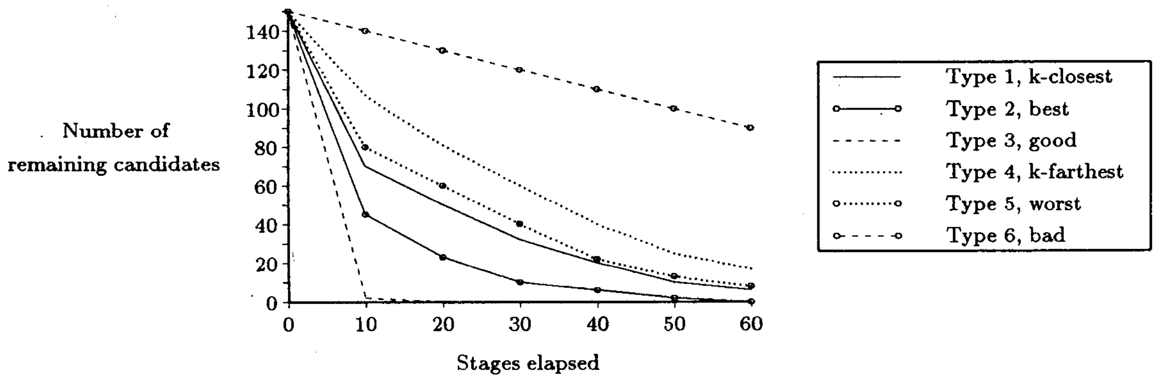

Figure 5: Candidate Set Size vs. Stages Elapsed.

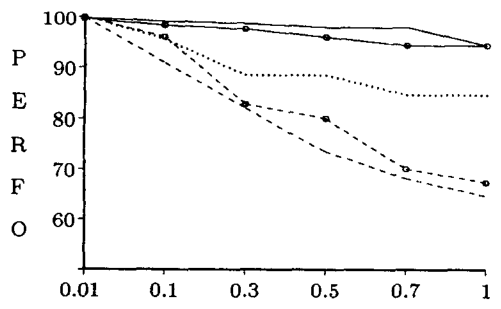

Figure 6(a): k-closest match.

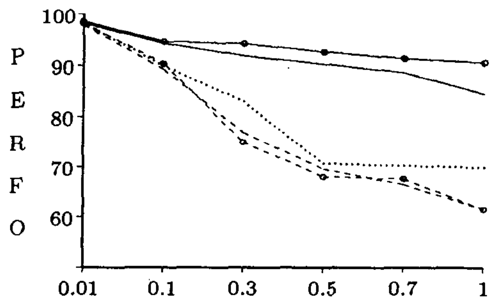

Figure 6(b): best-match.

Figure 6: Comparison of Heuristics for Types 1 and 2 Queries (Proteins).

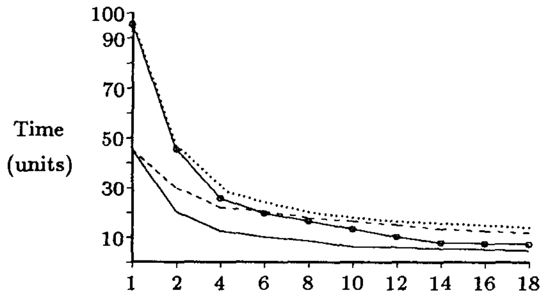

Figure 7(a): Time vs. Number of Processors.

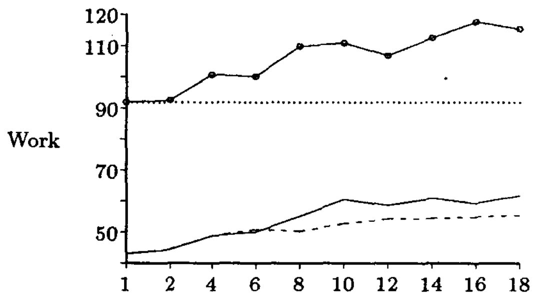

Figure 7(b): Work vs. Number of Processors.

Figure 7: Parallel Evaluation of Type 1 Query; k = 3.

are quite similar to those obtained from uniformly distributed data, namely the performance of all heuristics is close and dependent heavily on the value of ϵ.

5.3 Issues in Parallelizing Our Scheme

A set of programs were written to simulate the execution of the scheme using multiprocessors for both uniformly distributed data and proteins. The simulation study used the following strategy: When p processors are used in evaluating a query, at each stage these processors take the top p choices according to its best heuristic, compute all of them, update the augmented ADM and MIN, and then eliminate objects (if any) simultaneously. The other parameters had the values shown in Table 1.

Due to space limitations we cannot present all our results but have selected two graphs for the type 1 query to illustrate our findings. Figure 7(a) shows the time our scheme requires for various number of processors, where each distance computation is assumed to take one time unit. Figure 7(b) plots the total work, defined as $NumPE$ $\times$ the number of objects each processor computed, against the number of processors. Recalling that the size of candidate sets decreases fast in the first ten stages and then slowly as time goes on (cf. Figure 5), one may be interested in knowing whether parallel work would not be much greater than sequential work if one introduces more processors only after these stages. Both figures also answer this question.

First, we observe that using more processors is not as efficient as one might hope. For example, when using 10 processors for uniformly distributed data, one might hope the query evaluation can be completed in $44/10 = 4.4$ time units; yet empirically nearly 6 time units are needed (Figure 7(a)). The reason is that processors do useless work: one may compute an object which could be discarded if the scheme were executed serially. Second, it can be seen from Figure 7(b) that for uniformly distributed data, introducing more processors after 10 stages requires much less work than starting all of the processors from the beginning (the difference is less impressive when the number of processors is small). Taking an extreme example, if there are 150 processors available, using them altogether yields work = 150; in contrast, since 80 objects on the average can be eliminated in the first 10 stages (cf. Figure 5), using all the processors after these stages yields the total work = 10 + (150 - 80) = 80 only. Thus to effectively use processors, it is advisable to delay multiprocessing to later stages.

The results for proteins support this contention, where the parallel work for delaying 10 stages is exactly the same as sequential work. The explanation is somewhat different, however. Here, when a target protein belongs to a cluster, the process can be completed quickly, requiring less than 10 stages; on the other hand, a non-member target causes all the proteins to be computed. Thus neither case yields extra work if our scheme is parallelized after 10 stages. We feel that one would need to test larger files for various densities before drawing more precise conclusions about the exact number of stages one should wait and the work difference between whether to wait. One thing is clear, however, if there are a large number of processors available, more benefit can be obtained if one starts using them in the later stages in evaluating a query.

6 Conclusions

In this paper we have presented a scheme to answering a class of queries for retrieving similar or dissimilar objects to a given target from a database. Unlike previously published algorithms, the scheme allows the optimum use of any given precomputed information. We proposed four heuristics for expediting query processing. Our simulation results showed that of these heuristics, ***pick least lower bound*** (LLB) is best when searching for similar objects, while ***pick greatest upper bound*** (GUB) is best when searching for dissimilar objects. Our results also showed that when the scheme is executed in a multiprocessor environment, parallelism is best applied in the later stages in evaluating a query.

We have not discussed how to maintain an approximate distance map (ADM) when objects are inserted or deleted from a database, or when objects are updated in such a way that distances change. To maintain an ADM, one could periodically calculate those absent inter-object distances, or could incrementally update the distance information using the techniques described in [2, 7, 24].

Our algorithm for approximating a distance map requires $O(n^3)$ time, where n is the size of the database. This may make it infeasible to perform the complete computation in a single run when the database is very large. For this case, a "divide and conquer" approach can be useful and we suggest two such strategies to cope with a sizable file. One is simply to divide the whole database into several subdatabases, apply our scheme to each to find the desired objects, and then compare them to get the final result. The other is to preprocess and maintain an ADM (MIN) by decomposing the entire map into m blocks, each being an $n/m^{1/2} \times n/m^{1/2}$ matrix, and get the best approximation for each block. Then incrementally add the distances between objects that straddle different blocks and update the bounds for all object pairs using the algorithm presented in Section 2.3. When evaluating queries, the ADM map is obtained by coupling these blocks and available interblock distance bounds. We plan to study these alternatives and evaluate their performance in the future.

7 Acknowledgements

We are indebted to Jill Mesirov of Thinking Machines for providing proteins and their pairwise distances used in Sections 5.2 and 5.3. Kaizhong Zhang made many useful comments on the preliminary drafts of this paper and is gratefully acknowledged. Thanks are also to Richard

Cole, Rakesh Agrawal, Mitra Basu, and Deepak Sherlekar for their help in doing this work, and to the anonymous referees for their constructive suggestions.

References

[1] R. Agrawal and H. V. Jagadish, "Efficient Search in Very Large Databases", *Proc. 14th Int'l Conf. on Very Large Data Bases*, 1988, 407-418.

[2] R. Agrawal and H. V. Jagadish, "Materialization and Incremental Update of Path Information", *Proc. IEEE 5th Int'l Conf. Data Engineering*, Los Angeles, Ca., Feb. 1989, 374-383.

[3] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley Publishing Company, Reading, Mass., 1983.

[4] E. Bertino and W. Kim, "Indexing Techniques for Queries on Nested Objects", MCC TR. ACT-OODS-132-89, March 1989.

[5] J. M. Brayer and K. S. Fu, "The Derivation Diagram of a Web Grammar and Its Application to Scene Analysis", *1976 Joint Workshop on Pattern Recognition and Artificial Intelligence* (Hyannis, Mass., June 1-3, 1976), IEEE Publ. 76CH1169-2C, 1976.

[6] W. A. Burkhard and R. M. Keller, "Some Approaches to Best-Match File Searching", *Comm. ACM 16*, 4 (Apr. 1973), 230-236.

[7] G. Cheston, "Incremental Algorithms in Graph Theory", Tech. Rep. TR 91, Dept. of Comp. Sci., Univ. of Toronto, Canada, 1976.

[8] V. Claus, M. Ehrig and G. Rozenberg, Eds., *Graph-Grammars and Their Application to Computer Science and Biology*, Springer-Verlag, 1979.

[9] L. S. Davis and N. Roussopoulos, "Approximate Pattern Matching in a Pattern Database System", *Information Systems 5*, (1980), 107-119.

[10] U. Dayal, et al., "Simplifying Complex Objects: The PROBE Approach to Modeling and Querying Them", *Proc. German Database Conf.*, Darmstadt, Apr. 1987.

[11] C. M. Eastman and S. F. Weiss, "Tree Structures for High Dimensionality Nearest Neighbor Searching", *Information Systems 7*, 2 (1982), 115-122.

[12] R. W. Floyd, "Algorithm 97: Shortest Path", *Comm. ACM 5*, 6 (1962), 345.

[13] K. Fukunaga and P. M. Narendra, "A Branch and Bound Algorithm for Computing k-Nearest Neighbors", *IEEE Trans. on Computers 24*, 7 (July 1975), 750-753.

[14] T. Haerder, H. Schoning and A. Sikeler, "Parallelism in Processing Queries on Complex Objects", *Proc. Int'l Symp. on Databases in Parallel and Distributed Systems*, Austin, TX, Dec. 1988.

[15] A. Henrich, H.-W. Six and P. Widmayer, "The LSD Tree: Spatial Access to Multidimensional Point and Non-Point Objects", *Proc. 15th Int'l Conf. on Very Large Data Bases*, 1989, 45-53.

[16] T. Ito and M. Kizawa, "Hierarchical File Organization and Its Application to Similar-String Matching", *ACM Trans. on Database Systems 8*, 3 (Sep. 1983), 410-433.

[17] B. Kamgar-Parsi and L. N. Kanal, "An Improved Branch and Bound Algorithm for Computing k-Nearest Neighbors", *Pattern Recognition Letters 3*, 1 (1985), 7-12.

[18] E. Lander, J. P. Mesirov and T. Washington, "Protein Sequence Comparison on a Data Parallel Computer", *Proc. Int'l Conf. on Parallel Processing*, 1988.

[19] D. T. Lee and F. P. Preparata, "Computational Geometry – A Survey", *IEEE Trans. on Computers 33*, 12 (Dec. 1984), 1072-1101.

[20] D. J. Lipman and W. R. Pearson, "Rapid and Sensitive Protein Similarity Searches", *Science 227*, (1985), 1435-1441.

[21] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, M.I.T. Press, Cambridge, Mass., 1969.

[22] J. Orenstein, "Spatial Query Processing in an Object-Oriented Database System", *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, Washington D.C., May 1986, 326-336.

[23] J. Orenstein, "Redundancy in Spatial Databases", *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, 1989, 294-305.

[24] S. R. Pawagi, *Incremental Graph Algorithms for Parallel Random Access Machines*, Ph.D thesis, Dept. Comput. Sci., Univ. of Maryland, 1986.

[25] G. Saake, V. Linnemann, P. Pistor and L. Wegner, "Sorting, Grouping and Duplicate Elimination in the Advanced Information Management Prototype", *Proc. 15th Int'l Conf. on Very Large Data Bases*, 1989.

[26] H. Samet, "Distance Transform for Images Represented by Quadtrees", *IEEE Trans. on Pattern Analysis and Machine Intelligence 4*, 3 (May 1982), 298-303.

[27] T. Sellis, N. Roussopoulos and C. Faloutsos, "The R^+-Tree: A Dynamic Index for Multi-Dimensional Objects", *Proc. 13th Int'l Conf. on Very Large Data Bases*, 1987.

[28] L. G. Shapiro and R. M. Haralick, "Structural Descriptions and Inexact Matching", *IEEE Trans. Pattern Anal. Mach. Intell. 3* (Sept. 1981), 504-519.

[29] M. Shapiro, "The Choice of Reference Points in Best-Match File Searching", *Comm. ACM 20*, 5 (May 1977), 339-343.

[30] D. Shasha and T. L. Wang, "Optimal Best-Match Retrieval", submitted for publication, also available as NYU Computer Science Tech. Report, TR. 480, Dec. 1989.

[31] M. Stonebraker, B. Rubenstein and A. Guttman, "Application of Abstract Data Types and Abstract Indices to CAD Databases", *Proc. Database Week: Engineering Design Applications*, IEEE Computer Society, 1983.

[32] P. Valduriez, S. Khoshafian and G. Copeland, "Implementation Techniques of Complex Objects", *Proc. 12th Int'l Conf. on Very Large Data Bases*, 1986.

[33] S. Warshall, "A Theorem on Boolean Matrices", *J. ACM 9*, 1 (1962), 11-12.

[34] C. Zaniolo, "The Representation and Deductive Retrieval of Complex Objects", *Proc. 11th Int'l Conf. on Very Large Data Bases*, 1985.