K-Nearest Neighbor Finding Using MaxNearestDist

Hanan Samet, *Fellow, IEEE*
Computer Science Department
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
hjs@cs.umd.edu
www.cs.umd.edu/~hjs

***Abstract*—**

Similarity searching often reduces to finding the k nearest neighbors to a query object. Finding the k nearest neighbors is achieved by applying either a depth-first or a best-first algorithm to the search hierarchy containing the data. These algorithms are generally applicable to any index based on hierarchical clustering. The idea is that the data is partitioned into clusters which are aggregated to form other clusters, with the total aggregation being represented as a tree. These algorithms have traditionally used a lower bound corresponding to the minimum distance at which a nearest neighbor can be found (termed MINDIST) to prune the search process by avoiding the processing of some of the clusters as well as individual objects when they can be shown to be farther from the query object q than all of the current k nearest neighbors of q. An alternative pruning technique that uses an upper bound corresponding to the maximum possible distance at which a nearest neighbor is guaranteed to be found (termed MAXNEARESTDIST) is described. The MAXNEARESTDIST upper bound is adapted to enable its use for finding the k nearest neighbors instead of just the nearest neighbor (i.e., $k = 1$) as in its previous uses. Both the depth-first and best-first k-nearest neighbor algorithms are modified to use MAXNEARESTDIST, which is shown to enhance both algorithms by overcoming their shortcomings. In particular, for the depth-first algorithm, the number of clusters in the search hierarchy that must be examined is not increased thereby potentially lowering its execution time, while for the best-first algorithm, the number of clusters in the search hierarchy that must be retained in the priority queue used to control the ordering of processing of the clusters is also not increased, thereby potentially lowering its storage requirements.

***Index Terms*— k-nearest neighbors; similarity searching; metric spaces; depth-first nearest neighbor finding; best-first nearest neighbor finding**

I. INTRODUCTION

SIMILARITY searching is an important task when trying to find patterns in applications involving mining different types of data such as images, video, time series, text documents, DNA sequences, etc. Similarity searching often reduces to finding the k nearest neighbors to a query object. This process is facilitated by building an index on the data which is usually based on a hierarchical clustering. The idea is that the data objects are partitioned into clusters (termed *nonobjects*) which are aggregated to form other clusters, with the total aggregation being represented as a tree known as a search hierarchy. Numerous search hierarchies have been used for both vector data and non-vector data such as data lying in a metric space many of which are surveyed in [7], [10], [15], [20], [26], [30]. The methods that we describe in this paper are independent of the nature of the data.

The most common strategy for finding the k nearest neighbors is the depth-first method which explores the elements of the search hierarchy in a depth-first manner (e.g., [14]). The k nearest neighbors found so far are kept track of in a set L with the aid of a variable D_k that indicates the distance, using a suitably defined distance function d, of the current kth-nearest object from the query object q. The depth-first method visits every element of the search hierarchy. The *branch and bound* variant of the depth-first method yields better performance by not visiting every nonobject and its objects when it can be determined that it is impossible for the nonobject to contain any of the k nearest neighbors of q [14], [21]). For example, this is true if we know that for every nonobject element e of the search hierarchy, $d(q, e) \leq d(q, e_0)$ for every object e_0 in e and that the relation $d(q, e) > D_k$ is satisfied[1]. This can indeed be achieved if we define $d(q, e)$ as the minimum possible distance from q to any object e_0 in nonobject e (referred to as MINDIST in contrast to MAXDIST, the maximum possible distance, which unlike MINDIST cannot be used for pruning).

Letting $A(e)$ denote the set of nonobject immediate descendants e_i of nonobject element e of the search hierarchy, using the above definition of distance for nonobject elements (i.e., MINDIST) makes it possible to obtain even better performance as a result of speeding up the convergence of D_k to its final value by processing elements of $A(e)$ in increasing order of $d(q, e_i)$ (i.e., a MINDIST ordering). In this way, once an element e_i in $A(e)$ is found such that $d(q, e_i) > D_k$, then $d(q, e_j) > D_k$ for all remaining elements e_j of $A(e)$. This means that none of these remaining nonobject descendants of e need to be processed further, and the algorithm backtracks

This work was supported in part by the National Science Foundation under grants EIA-99-00268, EIA-99-01636, IIS-00-86162, EIA-00-91474, and CCF-0515241, Microsoft Research, and the University of Maryland General Research Board.

[1]This stopping condition ensures that all objects at the distance of the kth-nearest neighbor are examined. Note that if the size of L is limited to k and if there are two or more objects at distance D_k, then some of them may not be reported in the set of q's k nearest neighbors.

to the parent of e, or terminates if e is the root of the search hierarchy.

An alternative strategy is the best-first method (e.g., [1], [8], [11], [16]–[18], [23]) which explores the nonobject elements of the search hierarchy in increasing order of their distance from q (hence the characterization as "best-first") rather than in a predetermined order, as in the depth-first method. In other words, at each step of the algorithm, the next nonobject element to be visited is the closest one to q which has yet to be visited. This is achieved by storing the nonobject elements of the search hierarchy in a priority queue *Queue* according to this order. *Queue* is initialized to contain the root of the search hierarchy at a distance of 0 from q, and as nonobject elements are dequeued, their immediate descendants e that are nonobject elements are enqueued with their corresponding distances from q if $d(q, e) < D_k$, while immediate descendants o that are objects are inserted into L if $d(q, o) < D_k$, where D_k, the distance of the current kth-nearest neighbor of q, is initialized to ∞. The algorithm repeatedly removes nonobject elements from *Queue* until it is empty or until encountering a nonobject element that is farther from q than D_k, at which time it halts as it has found the k nearest neighbors, which are now in L. In order for the algorithm to be correct, the distance $d(q, e)$ of any nonobject element e from the query object q must be less than or equal to the distance from q to any object in e's descendants [19]. Again, as in the depth-first method, this property is satisfied by letting $d(q, e)$ be MINDIST.

The drawback of the best-first method is that the priority queue may be rather large. In particular, the necessary amount of storage may be as large as the total number of nonobjects (and hence on the order of the number of objects) if the distance of each of the nonobjects from the query object q is approximately the same. In low dimensions, such an event is relatively rare as its occurrence requires two seemingly independent events — that is, that all objects lie in an approximate hypersphere centered at some point p and that the query object q be coincident with p. However, in high dimensions, where most of the data lies on the surface (e.g., [5]) and the curse of dimensionality [3], [6] comes into play, and in metric spaces with concentrated distance histograms, this situation is less rare. In contrast, the amount of storage required by the depth-first method is bounded. In particular, it is proportional to the sum of k and the maximum depth of the search hierarchy, where, in the worst case, all of the sibling nonobject elements must be retained for each partially explored nonobject element in the search hierarchy while executing the depth-first search.

Nevertheless, the advantage of the best-first algorithm over the depth-first algorithm is that it has been shown to be I/O optimal for $k = 1$ [4]. This means that the algorithm does not visit more than the minimum number of nonobject elements—that is, it avoids visiting nonobject elements that will eventually be determined to be too far from q due to poor initial estimates of D_k. This is equivalent to stipulating that the algorithm is range-optimal, which means that the cost of finding the k nearest neighbors is the same as that of a range search with the search radius set to the distance from q to its kth-nearest neighbor.

As we point out above, the implementations of both the depth-first and best-first algorithms make heavy use of a lower bound MINDIST corresponding to the minimum distance at which a nearest object can be found vis a vis the distance D_k to the current kth-nearest object from q. In this paper, we show how to also use an upper bound MAXNEARESTDIST corresponding to the maximum possible distance at which a nearest neighbor is guaranteed to be found in both the depth-first and best-first algorithms for finding the k nearest neighbors for arbitrary values of k. This upper bound was first introduced in [22] for the case of $k = 1$ for the purpose of improving the initial estimate of D_k in the depth-first method. It was also proposed in [25] as an alternative to the MINDIST ordering for the processing of the nonobject immediate descendants of a nonobject element in the depth-first method but again limited to $k = 1$. However, for the purpose of ordering, MINDIST has been shown to be more useful [18], [25] and hence MAXNEARESTDIST is not discussed further here in this context. Therefore, in this paper we focus on the first purpose and show how to use the MAXNEARESTDIST upper bound for arbitrary values of k (i.e., $k \geq 1$) to overcome the shortcomings that we pointed out earlier of both the depth-first (i.e., pruning) and best-first (i.e., size of the priority queue) k-nearest neighbor algorithms. Note that other upper bounds can be used in the k-nearest neighbor algorithms to yield what are termed probabilistically approximate nearest neighbors (e.g., [9], [11]), although they are beyond the scope of this paper.

The rest of this paper is organized as follows. Section II defines the MAXNEARESTDIST upper bound, while Section III describes how to use it in the process of finding the k nearest neighbors by incorporating it into the set L of the k nearest neighbors found so far, while Section IV discusses the management of L in greater detail. Section V demonstrates how to incorporate it in a depth-first k-nearest neighbor algorithm to eliminate some elements from further consideration, while Section VI demonstrates how to incorporate it in a best-first k-nearest neighbor algorithm to reduce the size of the priority queue of nonobject elements remaining to be processed. Section VII presents the results of an experiment where use of MAXNEARESTDIST does indeed improve the performance of both the depth-first and best-first k-nearest neighbor algorithms, while concluding remarks are made in Section VIII.

II. THE MAXNEARESTDIST UPPER BOUND

The key motivation for the introduction of the MAXNEARESTDIST upper bound in k-nearest neighbor finding have been the observations that until the first set of k candidate nearest neighbors has been found (which enables setting D_k to a value other than the initial value of ∞),

1) regardless of the algorithm that is used, no visits of nonobject elements of the search hierarchy can be prevented, and
2) in the best-first method, no insertion of nonobject elements into the priority queue can be avoided.

In fact, Larsen and Kanal [22] first introduced MAXNEARESTDIST as an alternative to the MAXDIST upper bound that

was proposed by Fukunaga and Narendra [14] as a means of obtaining an upper bound on D_1 when finding the nearest neighbor using the depth-first method (see Figure 1a). In particular, Fukunaga and Narendra's proposal for using MAXDIST assumed a very simple search hierarchy where objects are clustered into groups where cluster c has a cluster center M which need not necessarily correspond to an object in the cluster and all objects in c lie within a distance r_{max} of M. Thus in this context, the minimum distance at which the nearest neighbor could be found was indeed MAXDIST=$d(q, M) + r_{max}$.

Larsen and Kanal [22] improved upon Fukunaga and Narendra [14] by noting that once the cluster center is not required correspond to an object in the cluster, the clusters can be formed even more tightly by taking advantage of the knowledge that r_{min} is the distance from cluster center M to M's closest object within cluster c. This results in a cluster having the shape of a spherical shell as shown in Figure 1b. In particular, they point out that the distance from the query object q to its nearest object, which we term MAXNEARESTDIST, regardless of which cluster it is in, cannot exceed $d(q, M) + r_{min}$ and thus D_1 could be reset if it exceeds this value.

Figure 1c illustrates the relationship between MINDIST, MAXNEARESTDIST, and MAXDIST by assuming a Euclidean distance metric and a cluster c in the form of a minimum bounding hypersphere of the objects lying within it. In this case, we see that the value of MAXNEARESTDIST is $\sqrt{d(q, M)^2 + r_{max}^2}$ and does lie between MINDIST and MAXDIST. Note that if we assume that the minimum bounding hypersphere in Figure 1c is a spherical shell with r_{min} as its inner radius and a Euclidean distance metric d, then MAXNEARESTDIST is the minimum of $\sqrt{d(q, M)^2 + r_{max}^2}$ and $d(q, M) + r_{min}$.

III. USING MAXNEARESTDIST IN k-NEAREST NEIGHBOR FINDING

Using MAXNEARESTDIST to tighten the estimate D_k when finding the k nearest neighbors instead of just the nearest neighbor (i.e., D_1 when $k = 1$) is not a simple matter, although neither Fukunaga and Narendra [14] nor Larsen and Kanal [22] give it any mention in their depth-first algorithms. In particular, note that the simple solution used for $k = 1$ of resetting D_1 cannot be generalized to k by simply resetting D_k to $\mathrm{MAXNEARESTDIST}(q, e)$ whenever $\mathrm{MAXNEARESTDIST}(q, e) < D_k$ for nonobject element e. The problem is that the distance s from q to some of the k nearest neighbors of q may lie within the range $\mathrm{MAXNEARESTDIST}(q, e) < s \leq D_k$, and thus resetting D_k to $\mathrm{MAXNEARESTDIST}(q, e)$ may cause them to be missed (e.g., object o in Figure 2, assuming a Euclidean distance metric).

The problem is that given the way in which the MAXNEARESTDIST upper bound is defined here, its primary role is to set an upper bound on the distance from the query object to its nearest neighbor in a particular nonobject element. It is important to observe that this is not the same as saying that the upper bound computed by using MAXNEARESTDIST is the minimum of the maximum possible distances to the kth-nearest neighbor of the query object, which is not true. Instead, the way in which the MAXNEARESTDIST upper bound should be, and is, used in k-nearest neighbor finding is to provide upper bounds for a number of different clusters. Only once we have obtained k distinct such upper bounds, do we have an upper bound on the distance to the kth-nearest neighbor.

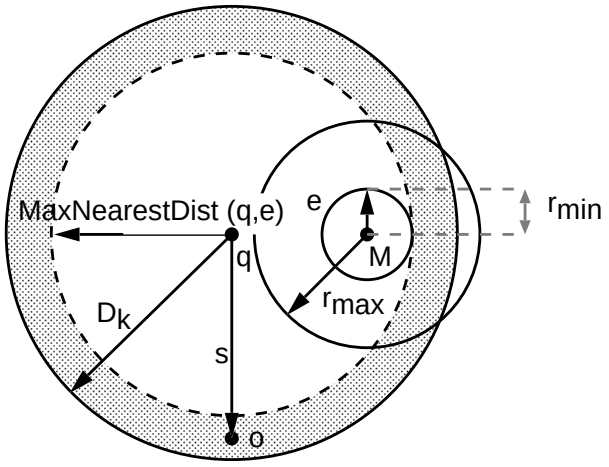

Figure 2: Example showing that we cannot simply reset D_k to MaxNearestDist(q,e) whenever MaxNearestDist(q,e)$<$ D_k for nonobject element e which is a spherical shell so that MaxNearestDist(q,e)$=$ $d(q, M) + r_{min}$, assuming a Euclidean distance metric.

We make use of the MAXNEARESTDIST upper bound by expanding the role played by the set L of the k nearest neighbors encountered so far so that it also contains nonobject elements e, such that $\mathrm{MAXNEARESTDIST}(q, e) < D_k$, that have been encountered so far (in the course of either the depth-first or best-first k-nearest neighbor algorithms) along with their corresponding MAXNEARESTDIST values, as well as continuing its role of containing objects with their corresponding distance values from q. In particular, each time we process a nonobject element e of the search hierarchy, we insert in L all of e's nonobject child elements along with their corresponding MAXNEARESTDIST values. In addition, before we attempt to insert the nonobject child elements of e into L, we remove e from L. The insertion of e into L requires some care so that we are sure to always find e when attempting to remove it.

IV. MANAGEMENT OF L

In this section we discuss the management of the set L of the k nearest neighbors encountered so far using the principles outlined in Section III. This discussion is independent of whether a depth-first or best-first algorithm is used to find the k nearest neighbors of query object q. In particular, we only assume that each time we process a nonobject element e of the search hierarchy, we insert in L all of e's nonobject child elements along with their corresponding MAXNEARESTDIST values. In addition, before we attempt to insert the nonobject child elements of e into L, we remove e from L.

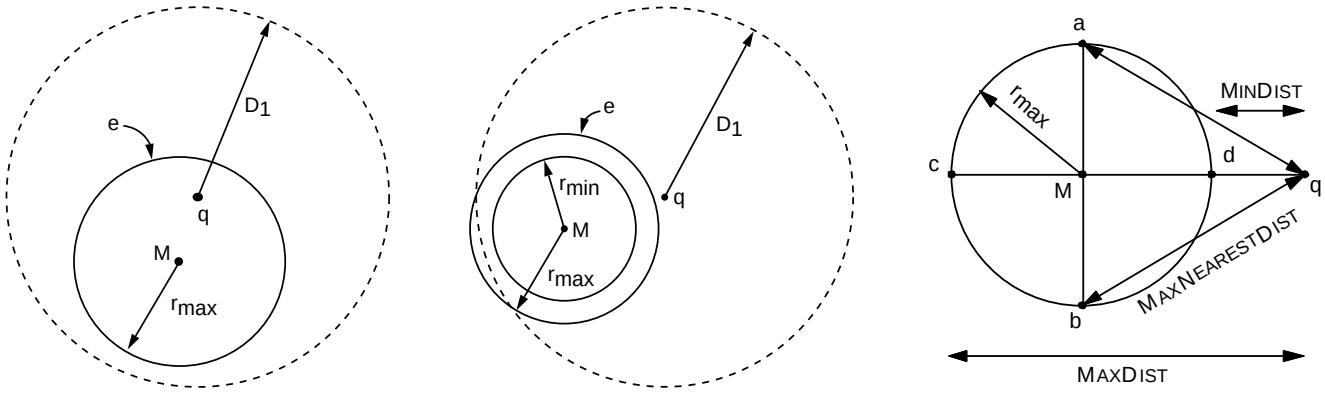

Figure 1: Examples showing the calculation of MaxDist, MaxNearestDist, and MinDist when objects are clustered into groups where the cluster centers do not necessarily correspond to objects in the clusters. (a) Cluster c has a cluster center M and all objects in c lie within a distance r_{max} of M, (b) adding the condition that r_{min} is the distance from M to M's closest object within c, and (c) the cluster is the minimum bounding hypersphere of a set of objects whose cluster center is determined to be M.

Given these assumptions about the new algorithms, the rest of this section is organized as follows. Section IV-A presents the implementation of L. Section IV-B proves a couple of key properties of L when used to store nonobject elements along with their corresponding MAXNEARESTDIST values in k-nearest neighbor finding. They are independent of which of the two variants of the original k-nearest neighbor algorithm is used. In particular, the only property of the algorithms that is used in the proofs is that they process the nonobject elements of the search hierarchy in increasing order of their MINDIST values either locally (depth-first algorithm) or globally (best-first algorithm). Section IV-C describes in greater detail the process of inserting nonobject elements in L, while Section IV-D discusses the process of the explicit removal of nonobject elements from L. The actual code for the two variants of the k-nearest neighbor finding algorithms that incorporate the MAXNEARESTDIST upper bound is given in Sections V and VI.

A. Implementation of L

The variants of the k-nearest neighbor algorithms that we describe need to be able to insert and remove specific objects and nonobjects with corresponding MAXNEARESTDIST values into and from the set L. In addition, we want to be able to access as well as delete the farthest of the k nearest neighbors. We implement L using a priority queue as it enables the latter two operations to be performed without needless exchange operations as would be the case if L were to be implemented using an array. Each element e in L has two data fields E and D corresponding to the item i (object or nonobject) that e contains and i's distance from q (i.e., $d(q, i)$ or $\text{MAXNEARESTDIST}(q, i)$, respectively), and a number of fields corresponding to control information specific to the data structure used to implement the priority queue (e.g., a binary heap). We use the function $\text{MAXL}(L)$ to access the element in L with the highest priority (i.e., at the top of the queue or, equivalently, the first and most accessible element). When the queue L is full, then $\text{MAXL}(L)$ corresponds to the kth-nearest neighbor (i.e., the farthest of the known k nearest neighbors of q).

B. Properties of L When Containing MaxNearestDist Values

In this section, we prove some important properties of the modification of L described in Section III. We assume that each object appears just once in the search hierarchy[2]. Associated with each nonobject element e in L is its corresponding $\text{MAXNEARESTDIST}(q, e)$ value. This value results from the postulation of the existence of an object o at this distance, and o is said to be *associated with* e, whether or not such an object actually exists. The key idea is that the particular positioning of o is what determines the $\text{MAXNEARESTDIST}(q, e)$ value. In particular, we prove that each of the elements of L is associated with a unique object (Theorem 4.1), and that there is no need to insert a nonobject element e such that $\text{MAXNEARESTDIST}(q, e) \geq D_k$ which means that L contains at most k elements (Theorem 4.2).

Theorem 4.1: The object o associated with each element e of L is unique.

Proof: Since the objects in the set from which the neighbors are drawn are unique, once object o appears as one of the elements in L, it cannot be a member of one of the nonobject elements in L. Moreover, the fact that nonobject element e is removed from L before inserting any of e's children into L ensures that no ancestor-descendant relationship exists between any two elements of L. Therefore, the object o associated with the nonobject element u of L at a distance of MAXNEARESTDIST is guaranteed to be unique even though it may not have been identified yet. In other words, at the time at which we are ready to insert u into L, its

[2]Search hierarchies where objects appear more than once as is the case for those based on a disjoint decomposition of the space from which the objects are drawn such as an R^+-tree [28] and a PM quadtree [27] are more complex and beyond the scope of this paper.

associated object o is not already in L nor is o associated with any other nonobject element in L. Note that the definition of the MAXNEARESTDIST upper bound ensures that for each of the entries u in L, there is at least one object o in the data set whose maximum possible distance from q is the one that is associated with u. ■

Theorem 4.2: There is no need to insert into L any nonobject element e such that $\text{MAXNEARESTDIST}(q, e) \geq D_k$, and thus the maximum size of L is k.

Proof: There is no need to insert in L any nonobject element e such that $\text{MAXNEARESTDIST}(q, e) \geq D_k$ as e by itself cannot be used to further reduce the value of D_k. Moreover, failing to insert such an e in L does not affect the k-nearest neighbor finding process as regardless of the nature of the k-nearest neighbor finding algorithm that is used (i.e., depth-first or best-first), the appropriate nonobject elements that are subsequently processed are explored in increasing order of their MINDIST values. Therefore, the fact that $\text{MINDIST}(q, e) \leq \text{MAXNEARESTDIST}(q, e)$ for all q and nonobject elements e, means that if $\text{MINDIST}(q, e) \geq D_k$, then e will never be explored further anyway, and if $\text{MINDIST}(q, e) < D_k$, then e will be explored regardless of the value of $\text{MAXNEARESTDIST}(q, e)$ and whether or not e was inserted into L. Therefore, not having inserted e in L makes no difference since the descendants of e will be explored anyway. The size of L is upper-bounded by k (actually, it can decrease) as when nonobject element e is removed from L, it could be the case that $\text{MAXNEARESTDIST}(q, e_i) \geq D_k$ for each of the immediate nonobject child elements e_i of e in which case none of them are inserted into L. Figure 3 is an example that illustrates how the size of L can decrease for an element e with three spherical shell-like children $\mathsf{e_a}$, $\mathsf{e_b}$, and $\mathsf{e_c}$, assuming a Euclidean distance metric. In particular, in this case, we assume, without loss of generality, that D_k is equal to $\text{MAXNEARESTDIST}(\mathsf{q}, \mathsf{e})$ and that $\text{MAXNEARESTDIST}(\mathsf{q}, \mathsf{e}_i) = d(\mathsf{q}, \mathsf{M}_i) + \mathsf{r}_{i,\mathsf{min}}$ for each of the children e_i ($i = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$). It is easy to see that $\text{MAXNEARESTDIST}(\mathsf{q}, \mathsf{e}_i) > D_k$ for each of e_i ($i = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$). ■

C. Insertion of Nonobject Elements into L

One of the key properties underlying any k-nearest neighbor algorithm is that the value of D_k, the distance of the kth-nearest neighbor of q, is nonincreasing. This is ensured by Theorem 4.2 and the actual k-nearest neighbor algorithms (see Sections V and VI) which make use of procedure MAXNEARESTDISTINSERTL, given below, to update L as closer objects and nonobjects are found, thereby causing existing elements in L to be removed when L already contains k elements. Elements of L that are removed in such a manner are said to be removed *implicitly*, in contrast to elements that are removed *explicitly* whenever attempts are made to insert their children into L (see Section IV-D).

```
1  procedure MAXNEARESTDISTINSERTL(e, s)
2  /* Insert element (object or nonobject) e at distance s
       from query object q into the priority queue L using
       ENQUEUE. Assume that objects have precedence over
       nonobjects when they are at the same distance D_k (i.e.,
       the kth- nearest neighbor) from the query object q. */
3  if SIZE(L) = k then
4     h←DEQUEUE(L)
5     if not ISOBJECT(E(h)) then
6        while not ISEMPTY(L)
7              and not ISOBJECT(E(MAXL(L)))
8              and D(MAXL(L)) = D(h) do
9           DEQUEUE(L)
10       enddo
11    endif
12 endif
13 ENQUEUE(e, s, L)
14 if SIZE(L) = k then
15    if D(MAXL(L)) < D_k then
16       D_k←D(MAXL(L))
17    endif
18 endif
```

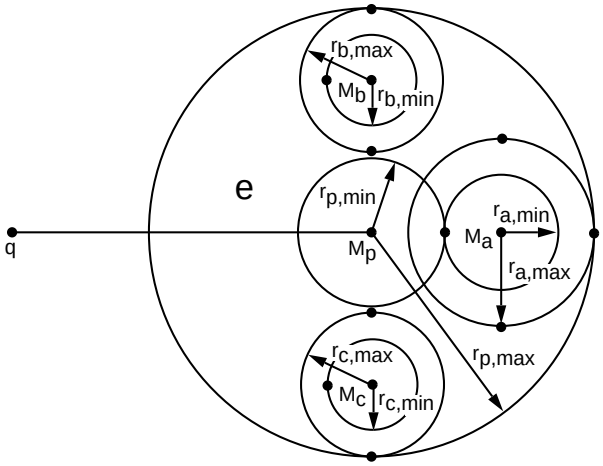

Figure 3: Example illustrating how the number of elements in L can decrease as a result of the situation that arises when the MaxNearestDist values of the three nonobject child elements $\mathsf{e_a}$, $\mathsf{e_b}$, and $\mathsf{e_c}$ of $\mathsf{e_p}$ are greater than the current value of $\mathsf{D_k}$ which is the MaxNearestDist value of $\mathsf{e_p}$, assuming a Euclidean distance metric.

Procedure MAXNEARESTDISTINSERTL proceeds as follows. If there are k candidate nearest neighbors in L (determined with the aid of the function $\text{SIZE}(L)$ which is not given here), then MAXNEARESTDISTINSERTL precedes the insertion, which is performed by ENQUEUE (not given here), by first dequeueing (i.e., implicitly removing) the current farthest member (i.e., the kth-nearest member) from L using DEQUEUE (not given here)[3]. Next, if there are k candidate nearest neighbors after the insertion, then MAXNEARESTDISTINSERTL resets D_k to the distance of the current farthest nearest neighbor (see lines 15 and 16), accessed by the

[3]Note the asymmetry between DEQUEUE which removes the item at the front of the queue while ENQUEUE inserts an item in its appropriate position in the queue. We also make use of a procedure REMOVEQUEUE in Section IV-D which is the complement of ENQUEUE in that it removes a specific element from the queue which may involve a search.

function MAXL (not given here, although in the case of a priority queue implemention of L as in this case, it is the current first element in the queue).

The situation is different when there are fewer than k candidate nearest neighbors in L as now there is no need to dequeue (i.e., implicitly remove) any elements from L. An example of such a situation was given in the proof of Theorem 4.2 and is also always the case in the initial stages of the k-nearest neighbor finding process. Moreover, in this case, the insertion of an object or nonobject e into L does not cause a change in D_k as D_k indicates the minimum of all of the distance values that have been associated with the entry in L that corresponds to the kth-nearest neighbor of q and D_k has not changed as a result of the current insertion of e at a distance $d(q, e) < D_k$ into L.

When L contains k elements and we must implicitly remove an element e in order to accommodate the insertion of the new element with distance less than D_k, then we must exercise some caution if there are several elements (objects and nonobjects) at the same distance D_k from q. The motivation for this is to try to minimize the need to search for e, and possibly not find e due to e having been implicitly removed from L earlier, when we subsequently attempt to explicitly remove e from L. This needless search is avoided by adopting some convention as to which element of L at distance D_k should be removed implicitly by MAXNEARESTDISTINSERTL when there are several nonobjects in L having D_k as their MAXNEARESTDIST value as well as objects at distance D_k.

We adopt the convention that objects have priority over nonobjects in the sense that in terms of nearness, objects have precedence over nonobjects in L. This means that when nonobjects and objects are at the same distance from q, the nonobjects appear closer to the maximum entry in the priority queue L (i.e., MAXL(L)), which corresponds to the kth-nearest candidate neighbor. In particular, we stipulate that whenever insertion into a full priority queue results in dequeuing a nonobject element b with MAXNEARESTDIST value d, we check if the new MAXL(L) entry c corresponds to a nonobject with the same MAXNEARESTDIST value d in which case c is also dequeued. This loop continues until the new MAXL(L) entry corresponds to an object at any distance including d, or corresponds to a nonobject at any other distance $d' < d$, or L is empty (see lines 5–11). Note that D_k is only reset if exactly one entry has been dequeued (from a full priority queue) and the distance of the new MAXL(L) entry is less than D_k (see lines 13–17). Otherwise, if we dequeue more than one entry, then even though the distance of the new MAXL(L) entry may now be less than D_k, it cannot be used to reset D_k as L now contains fewer than k entries. In fact, it should be clear that D_k should not be reset as D_k has not been decreased since the only reason for the removal of the multiple nonobject entries is to avoid subsequent possibly needless searches when explicitly removing nonobject elements with MAXNEARESTDIST value D_k.

D. Removal of Nonobject Elements from L

As we pointed out in the proof of Theorem 4.2, there is no need for L to ever contain more than k elements. This simplifies the k-nearest neighbor algorithms considerably. However, it does mean that when we need to explicitly remove a nonobject element e from L just before inserting in L all of e's nonobject child elements along with their corresponding MAXNEARESTDIST values that are less than D_k, it could be the case that e is no longer in L. This is because e may have been implicitly removed as a byproduct of the insertion of closer objects or nonobject elements as a result of their corresponding MAXNEARESTDIST values being smaller than that of e and thereby resulted in resetting D_k. Procedure MAXNEARESTDISTREMOVEL, given below, accomplishes this task, while also following our convention, set forth in Section IV-C, that objects have priority over nonobjects when they are in L at the same distance D_k (i.e., the kth-nearest neighbor) from the query object q.

```
1 procedure MAXNEARESTDISTREMOVEL(e)
2 /* Remove element (object or nonobject) e from the
     priority queue L using REMOVEQUEUE. Assume that
     objects have precedence over nonobjects when they
     are in L at the same distance D_k (i.e., the kth-nearest
     neighbor) from the query object q. */
3 if MAXNEARESTDIST(q, e) < D_k or
4    (MAXNEARESTDIST(q, e) = D_k and
5     D(MAXL(L)) = D_k and
6     not ISOBJECT(E(MAXL(L)))) then
7    REMOVEQUEUE(e, L)
8 endif
```

Procedure MAXNEARESTDISTREMOVEL proceeds as follows. When a nonobject e is to be removed explicitly from L and e's MAXNEARESTDIST value is $< D_k$ (see line 3), then e has to be in L as it is impossible for e to have been removed implicitly since D_k is nonincreasing in our algorithms (i.e., the depth-first and best-first given in Sections V and VI). Therefore, we remove e and decrement the size of L using procedure REMOVEQUEUE which is not given here. On the other hand, the situation is more complicated when e's MAXNEARESTDIST value is equal to D_k (see line 4). First, if the maximum value associated with an element in L (i.e., the one associated with MAXL(L)) is less than D_k (see line 5), then e cannot be in L, and we do not attempt to remove e. Such a situation arises, for example, when we have dequeued more than one nonobject due to having several nonobjects at distance D_k. Second, if the maximum value associated with an element in L (i.e., the one associated with MAXL(L)) is equal to D_k, then there are two cases depending on whether the entry c in MAXL(L) corresponds to an object or a nonobject (see line 6). If c corresponds to an object, then nonobject e cannot be in L as we have given precedence to objects, and all nonobjects at the same distance are either in L or they are all not in L. If c corresponds to a nonobject, then nonobject e has to be in L as all of the nonobjects at the same distance have been either removed implicitly together or retained, and, in this case, by virtue of the presence of c in L we know that they have been retained in L. Note that when we explicitly remove a nonobject at distance D_k from L, we do not remove all remaining nonobjects at the same distance from L as this needlessly complicates the algorithm with no additional benefit

as they will all be removed implicitly together later if at least one of them must be implicitly removed due to a subsequent insertion into a full priority queue.

V. Depth-First Algorithm Using MaxNearestDist

Incorporating the MaxNearestDist upper bound in the depth-first k-nearest neighbor algorithm, thereby yielding what we characterize as a *maxnearest depth-first* k-nearest neighbor algorithm, is straightforward and is realized below by the recursive procedure MaxNearestDistDF which is invoked with parameter e initialized to the root of the search hierarchy. In MaxNearestDistDF, if the nonobject element e being visited is at the deepest level of the search hierarchy (usually referred to as a *leaf* or *leaf element*), then every object o in e that is nearer to q than the current kth-nearest neighbor of q (i.e., $d(q,o) < D_k$) is inserted into L, with its associated distance from q (i.e., $d(q,o)$), using procedure MaxNearestDistInsertL given in Section II (lines 2–9 of MaxNearestDistDF). Otherwise (i.e., e is not a leaf element), MaxNearestDistDF generates the immediate successors of e, places them in a list $A(e)$, known as the *active list* of e, and proceeds to insert any of them whose MinDist and MaxNearestDist values are less than D_k into L (lines 12–21). It then proceeds to recursively process all of the nonobject child elements of e whose MinDist value is less than D_k (line 24) after removing them from L if possible (line 26).

```
 1 recursive procedure MaxNearestDistDF(e)
 2 if IsLeaf(e) then   /* e is a leaf with objects */
 3    foreach object child element o of e do
 4       Compute d(q,o)
 5       if d(q,o) < D_k or
 6          (d(q,o) = D_k and Size(L) < k)   then
 7          MaxNearestDistInsertL(o, d(q,o))
 8       endif
 9    enddo
10 else /* e is a nonleaf with nonobjects e_p */
11    Generate active list A with child elements e_p of e
12    /* A is sorted in increasing order with respect to q
         using MinDist and processed in this order */
13    foreach element e_p of A do
14       /* Try to apply MaxNearestDist while processing A in increasing order */
15       if MinDist(q, e_p) > D_k then
16          exit_for_loop /* No further insertions */
17       elseif MaxNearestDist(q, e_p) < D_k   then
18          MaxNearestDistInsertL(
19             e_p, MaxNearestDist(q, e_p))
20       endif
21    enddo
22    foreach element e_p of A do
23       /*Process A in increasing order */
24       if MinDist(q, e_p) > D_k then exit_for_loop
25       else
26          MaxNearestDistRemoveL(e_p)
27          MaxNearestDistDF(e_p)
28       endif
29    enddo
30 endif
```

We now prove a couple of important properties of our algorithm. First, there are no nonobject elements in L when the algorithm (i.e., MaxNearestDistDF) terminates.

Theorem 5.1: There are no nonobject elements in L when the maxnearest depth-first k-nearest neighbor algorithm (i.e., MaxNearestDistDF) terminates.

Proof: We show this by contradiction. Suppose that L contains a nonobject element e upon termination. The fact that the algorithm has terminated means that $\text{MinDist}(q,u) > D_k$ for all unprocessed nonobject elements u. However, the presence of e in L means that $\text{MaxNearestDist}(q,e) \leq D_k$ and therefore by virtue of $\text{MinDist}(q,e) \leq \text{MaxNearestDist}(q,e)$ we know that e has been processed already which means that e must have been removed explicitly from L which contradicts our initial assumption that L contains nonobject elements upon termination. ■

Second, we prove our main result which is that the number of nonobject elements that must be examined due to using the MaxNearestDist upper bound is not increased.

Theorem 5.2: The maxnearest depth-first k-nearest neighbor algorithm (i.e., MaxNearestDistDF) visits at most the same number of nonobject elements of the search hierarchy as the conventional depth-first algorithm, and may visit less.

Proof: We know that D_k is nonincreasing as it is only updated when an object or nonobject at a distance less than D_k is inserted into L. This is true for both the conventional and maxnearest versions of the depth-first k-nearest neighbor algorithm. Inserting a nonobject element e into L in the maxnearest algorithm causes D_k to decrease or at the worst to maintain the same value. Suppose that D_k has indeed decreased so that it now has the value d_e instead of the previous value of d_p. This means that a nonobject element n with minimum distance d_n such that $d_e < d_n < d_p$ will not be visited whereas n would have been visited had we not used the MaxNearestDist upper bound, and thus the number of nonobject elements that are visited has decreased. ■

VI. Best-First Algorithm Using MaxNearestDist

Incorporating the MaxNearestDist upper bound in the best-first algorithm, thereby yielding what we characterize as a *maxnearest best-first* k-nearest neighbor algorithm, is straightforward and is realized below by procedure MaxNearestDistBF which is invoked with parameter e initialized to the root of the search hierarchy. Recall that in the depth-first algorithm, incorporation of MaxNearestDist enabled the use of the nonobject elements of the search hierarchy to speed up the convergence of D_k to its final value thereby helping to prune the set of k candidate nearest neighbors instead of pruning only with the aid of the k nearest objects as in the standard implementation. It should be clear that both the fact that a best-first algorithm examines the nonobject elements in increasing MinDist order and the fact that every nonobject element with MinDist less than the final value of D_k must be examined together mean that no matter how fast the value

of D_k converges to its final value, a best-first algorithm will never examine any extra nonobject elements. However, use of MAXNEARESTDIST in the best-first algorithm still helps to speed up the convergence of D_k to its final value which means that its use results in reducing the size of the priority queue *Queue* as fewer nonobject elements are inserted into it initially while D_k is at its initial value of ∞.

```
 1 procedure MAXNEARESTDISTBF(e)
 2 ENQUEUE(e, MAXNEARESTDIST(q, e), L)
 3 ENQUEUE(e, 0, Queue)
 4 while not ISEMPTY(Queue) do
 5    t←DEQUEUE(Queue)
 6    e←E(t)
 7    if D(t) > D_k then
 8       return /* Found k nearest neighbors and exit */
 9    else MAXNEARESTDISTREMOVEL(e)
10    endif
11    if ISLEAF(e) then /* e is a leaf with objects */
12       foreach object child element o of e do
13          Compute d(q, o)
14          if d(q, o) < D_k or
15             (d(q, o) = D_k and SIZE(L) < k) then
16             MAXNEARESTDISTINSERTL(o, d(q, o))
17          endif
18       enddo
19    else /* e is a nonleaf */
20       foreach child element e_p of e do
21          if MINDIST(q, e_p) < D_k then
22             if MAXNEARESTDIST(q, e_p) < D_k then
23                MAXNEARESTDISTINSERTL(
24                   e_p, MAXNEARESTDIST(q, e_p))
25             endif
26             ENQUEUE(e_p, MINDIST(q, e_p), Queue)
27          endif
28       enddo
29    endif
30 enddo
31 return
```

MAXNEARESTDISTBF processes all nonobject elements in *Queue* in the order in which they appear in *Queue* (i.e., the element e at the front is processed first). We first remove e from *Queue* (lines 5–6), and also check if e should be explicitly removed from L using the same method as in MAXNEARESTDISTDF (line 9). Recall that this step ensures that the objects that are associated with the different entries in L are unique. This removal step is missing in a variant of a best-first k-nearest neighbor algorithm that uses MAXNEARESTDIST proposed by Ciaccia, Patella, and Zezula for the M-tree [12] thereby possibly leading to erroneous results. Next, we check if e is a leaf element in which case we examine its constituent objects using the same method as in MAXNEARESTDISTDF (lines 12–18). Otherwise, we examine each of e's child elements e_p, and insert e_p and its associated $\text{MINDIST}(q, e_p)$ value into *Queue* (line 26) if $\text{MINDIST}(q, e_p)$ is less than D_k (line 21). When $\text{MINDIST}(q, e_p) \leq D_k$, we also check if e_p's associated $\text{MAXNEARESTDIST}(q, e_p)$ value is less than D_k (line 22), in which case we use MAXNEARESTDISTINSERTL to insert e_p and $\text{MAXNEARESTDIST}(q, e_p)$ into L, and possibly reset D_k (line 23). As in MAXNEARESTDISTDF, this action may cause some elements (both objects and nonobjects) to be implicitly removed from L. Thus the MAXNEARESTDIST upper bound is used here to tighten the convergence of D_k to its final value.

Notice that in contrast to the depth-first algorithm (MAXNEARESTDISTDF), the nonobject child elements e_p of nonobject element e (i.e., the elements of the active list of e) are not sorted with respect to their distance (MINDIST or MAXNEARESTDIST) from q before testing for the possibility of insertion into L and enqueuing into *Queue* (lines 20–28). In particular, assuming data of a fixed dimension and that the active list contains T elements, there is no advantage in incurring the extra time needed to sort the child elements (i.e., $O(T \log T)$ time) since all that the sort can accomplish is avoiding the tests (i.e., $O(T)$ time). In other words, unlike the depth-first algorithm, in the best-first algorithm, there is no need to worry about ordering the processing of the elements of the active list.

We now prove a couple of important properties of our algorithm. First, there are no nonobject elements in L when the algorithm (i.e., MAXNEARESTDISTBF) terminates.

Theorem 6.1: There are no nonobject elements in L when the maxnearest best-first k-nearest neighbor algorithm (i.e., MAXNEARESTDISTBF) terminates.

Proof: The fact that each time that a nonobject element e is removed from the priority queue *Queue*, e is also explicitly removed from L if it is in L by virtue of its MAXNEARESTDIST value being less than D_k, and the fact that $\text{MINDIST}(q, e) \leq \text{MAXNEARESTDIST}(q, e)$ together ensure that there are no nonobject elements left in L when the best-first algorithm terminates (i.e., when the distance value associated with the first element in the priority queue *Queue* is greater than D_k), and thus the elements in L are the k nearest neighbors of q. ■

Theorem 6.1 is of interest as when the best-first algorithm terminates, it is quite likely that the priority queue *Queue* is not empty as we do not constantly check it for the presence of nonobject elements with associated distance values greater than D_k each time the value of D_k decreases.

Second, we prove our main result which is that the maximum number of nonobject elements that may be in the priority queue due to using the MAXNEARESTDIST upper bound is not increased.

Theorem 6.2: When the maxnearest best-first k-nearest neighbor algorithm (i.e., MAXNEARESTDISTBF) terminates, the maximum size attained by the priority queue *Queue* is at most as large as that of the conventional best-first algorithm, and may be less.

Proof: We know that D_k is nonincreasing as it is only updated when an object (nonobject) at a distance (MAXNEARESTDIST) less than D_k is inserted into L. This is true for both the conventional and maxnearest versions of the best-first k-nearest neighbor algorithm. Inserting a nonobject element e into L in the maxnearest algorithm causes D_k to decrease or at the worst to maintain the same value. Suppose that D_k has indeed decreased so that it now has the value d_e instead

of the previous value of d_p. This means that a subsequently processed nonobject child element n with minimum distance d_n such that $d_e < d_n < d_p$ will not be inserted into *Queue* whereas n would have been inserted into *Queue* had we not used the MAXNEARESTDIST upper bound and thus the size of *Queue* has decreased. ■

VII. EXPERIMENTAL RESULTS

In this section we demonstrate, with the aid of an example, situations where use of MAXNEARESTDIST can lead to additional pruning in the depth-first algorithm and likewise to a reduction in the size of the priority queue in the best-first algorithm. In particular, assuming a Euclidean distance metric, we applied the depth-first and best-first algorithms with and without the use of MAXNEARESTDIST to the set of 100 two-dimensional points given in Figure 4, stored in the R*-tree [2], which is an object hierarchy where the minimum bounding boxes are hyperrectangles instead of spheres as is the case for the SS-tree [29]. The R*-tree has the desirable property that overlap is kept low between minimum bounding boxes at the same level (i.e., they are more likely to be disjoint or close to disjoint).

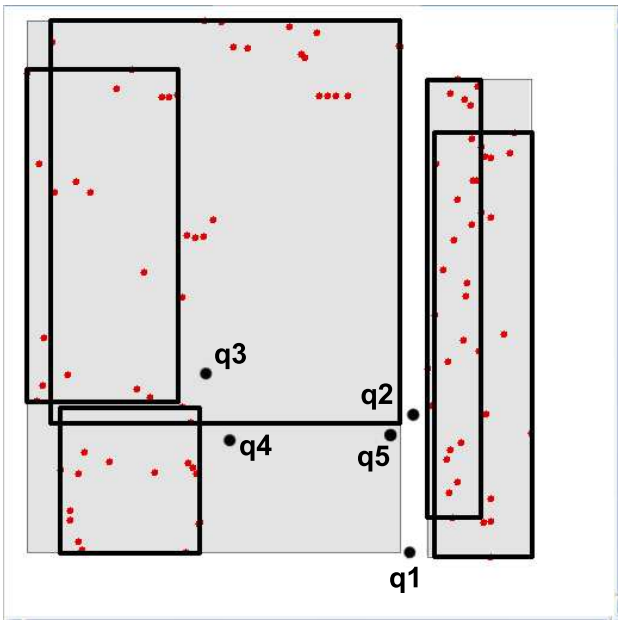

Figure 4: Block decomposition at the top 2 levels (depth 2 shown with darker borders than depth 1) in an R*-tree for 100 points in a 512×512 space with the origin at the upper-left corner where the fanout at each node lies between 2 and 4. The maximum depth of the tree is 6. q1, q2, q3, q4, and q5 correspond to query points.

Figure 5 shows the difference in performance of the depth-first (DF) and best-first (BF) algorithms with and without use of MAXNEARESTDIST for values of k, the number of nearest neighbors sought, ranging from 1 to 6 for five different query points labeled q1...q5 on the data in Figure 4. From this example, we see that the improvements/savings are maximized as k gets smaller vis-a-vis the maximum capacity (i.e., fanout) of the nodes of the search hierarchy that is used. This is not surprising as MAXNEARESTDIST, which is most effective at the initial stage of the search, cannot take effect until at least k nonobjects have been processed. Thus when the capacity is less than k, it does not come into effect until nodes at depth 1 have been processed, which reduces its pruning power both in terms of nodes to be processed (depth-first) and enqueued (best-first).

We also observe that the benefit of using MAXNEARESTDIST is more pronounced in the case of the best-first algorithm where one of our examples demonstrated a 400% improvement while 10–15% seems a more reasonable expectation for the best-first algorithm. The fact that MAXNEARESTDIST is more effective in the best-first algorithm than the depth-first algorithm is attractive as the best-first algorithm is I/O optimal and thus we are overcoming its only drawback, which is the potentially large queue size.

In general, from our example, it can be seen that the improvement/savings that can be obtained from use of MAXNEARESTDIST are heavily dependent on the underlying distribution of the data and on the positioning of the query point. However, most importantly, the performance of the algorithms can only be improved by using MAXNEARESTDIST, whereas this is not the case in some of the prior usage of MAXNEARESTDIST where it was used to order the processing of the elements of the active list in the depth-first algorithm (e.g., [13], [25]).

VIII. CONCLUDING REMARKS

We have shown how to use MAXNEARESTDIST, an upper bound corresponding to the maximum possible distance at which a nearest neighbor is guaranteed to be found, to enhance the performance of both a depth-first branch and bound and a best-first k-nearest neighbor finding algorithm by virtue of yielding tighter initial estimates of D_k, the distance at which the kth-nearest neighbor is found. This enables us to start pruning elements of the search hierarchy (both objects and nonobjects) in the depth-first algorithm and to avoid entering nonobject elements in the priority queue in the best-first algorithm. Thus we see that use of MAXNEARESTDIST enhances the performance of both the depth-first and best-first algorithms by addressing their shortcomings — that is, reducing the number of nonobject elements that need to be examined by the former and reducing the storage requirements of the latter at no extra cost in the execution time of the latter and no extra storage requirements for the former. Nevertheless, it is important to bear in mind that we are not saying anything about the relative performance of the two algorithms, which is a more general issue and beyond the scope of this paper.

Some implementations of the best-first nearest neighbor algorithm (e.g., [16]–[18], [24]) also store the objects in a priority queue thereby enabling the algorithms that employ this method to be incremental. This means that now both the objects and nonobjects are visited in increasing order of their distance from q, and the objects are also reported in increasing order of their distance from q. They are designed for the case

Number of Neighbors	Query Points									
	q1=(335,453)		q2=(339,343)		q3=(170,309)		q4=(190,365)		q5=(326,353)	
	DF	BF	DF	BF	DF	BF	DF	BF	DF	BF
1	15:11	12:3	13:9	13:5	9:9	8:5	9:9	8:4	17:13	18:5
2	15:11	12:3	13:9	13:5	9:9	8:5	9:9	8:4	17:13	18:5
3	21:15	12:6	21:19	13:10	13:13	10:5	11:9	10:6	21:19	18:8
4	26:19	12:9	23:20	13:10	15:13	11:7	13:11	10:6	23:19	18:11
5	26:24	12:9	23:21	17:13	16:14	11:7	14:11	10:7	23:21	18:14
6	26:25	12:10	24:23	17:13	16:14	11:10	16:14	10:7	26:23	18:16

Figure 5: The effect of using MaxNearestDist in the depth-first and best-first k-nearest neighbor finding algorithm for 5 different query points in the 100 data point R*-tree of Figure 4 where each node contains at least 2 and at most 4 points. Entries A:B in columns labeled DF indicate the number of recursive calls to the procedure using MaxNearestDist (B) and without using it (A). Entries C:D in columns labeled BF indicate the maximum size of the priority queue Queue using MaxNearestDist (D) and without using it (C).

that k is not known in advance, thereby making them inappropriate for use with the MAXNEARESTDIST upper bound as no nonobject elements can be excluded from *Queue* since they may all be eventually needed should k get sufficiently large (the same holds for probabilistic algorithms such as [9]).

Note, that the complexity of the process of computing the MAXNEARESTDIST upper bound depends on the nature of clustering process used to form the search hierarchy, as well as the domain of the data. For example, in d dimensions, using the Euclidean distance metric, its complexity is the same as that of MINDIST when the cluster elements are minimum bounding hyperspheres (i.e., $O(d)$), whereas when the cluster elements are minimum bounding hyperrectangles, the complexity of MAXNEARESTDIST is $O(d^2)$ while the complexity of MINDIST in this case is just $O(d)$.

The utility of the MAXNEARESTDIST upper bound depends on the distribution of the underlying data and also on the nature of the clustering methods that are applied in forming the search hierarchies. An interesting and open question is determining the type of data distributions and clustering methods for which MAXNEARESTDIST is most effective. For example, MAXNEARESTDIST may be most useful when using nonstandard clustering methods where objects are not necessarily associated with the closest cluster center (see object o in Figure 6). Similarly, as another example, consider clusters formed by the five interlocking Olympic rings. On the other hand, MAXNEARESTDIST is not particularly useful for uniformly-distributed data or when the query object is inside one of the clusters.

REFERENCES

[1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6):891–923, November 1998. Also see *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, Arlington, VA, January 1994.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference*, pages 322–331, Atlantic City, NJ, June 1990.

[3] R. E. Bellman. *Adaptive Control Processes*. Princeton University Press, Princeton, NJ, 1961.

[4] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 78–86, Tucson, AZ, May 1997.

[5] S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk-load operations. In *Advances in Database Technology—EDBT'98, Proceedings of the 1st International Conference on Extending Database Technology*, H.-J Schek, F. Saltor, I. Ramos, and G. Alonso, eds., vol. 1377 of Springer-Verlag Lecture Notes in Computer Science, pages 216–230, Valencia, Spain, March 1998.

[6] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *Proceedings of the 7th International Conference on Database Theory (ICDT'99)*, C. Beeri and P. Buneman, eds., vol. 1540 of Springer-Verlag Lecture Notes in Computer Science, pages 217–235, Berlin, Germany, January 1999.

[7] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, September 2001.

[8] A. J. Broder. Strategies for efficient incremental nearest neighbor search. *Pattern Recognition*, 23(1–2):171–178, January 1990.

[9] B. Bustos and G. Navarro. Probabilistic proximity searching algorithms based on compact partitions. *Journal of Discrete Algorithms*, 2(1):115–134, March 2004.

[10] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–322, September 2001.

[11] P. Ciaccia and M. Patella. PAC nearest neighbor queries: approximate and controlled search in high-dimensional and metric spaces. In *Proceedings of the 16th IEEE International Conference on Data Engineering*, pages 244–255, San Diego, CA, February 2000.

[12] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, eds., pages 426–435, Athens, Greece, August 1997.

[13] H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, and A. El Abbadi. Constrained nearest neighbor queries. In *Advances in Spatial and Temporal Databases—7th International Symposium, SSTD'01*, C. S. Jensen, M. Schneider, B. Seeger, and V. J. Tsotras, eds., vol. 2121 of Springer-Verlag Lecture Notes in Computer Science, pages 257–278, Redondo Beach, CA, July 2001.

[14] K. Fukunaga and P. M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Transactions on Computers*, 24(7):750–753, July 1975.

[15] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 20(2):170–231, June 1998.

[16] A. Henrich. A distance-scan algorithm for spatial access structures. In *Proceedings of the 2nd ACM Workshop on Geographic Information Systems*, N. Pissinou and K. Makki, eds., pages 136–143, Gaithersburg, MD, December 1994.

[17] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *Advances in Spatial Databases—4th International Symposium, SSD'95*, M. J. Egenhofer and J. R. Herring, eds., vol. 951 of Springer-Verlag

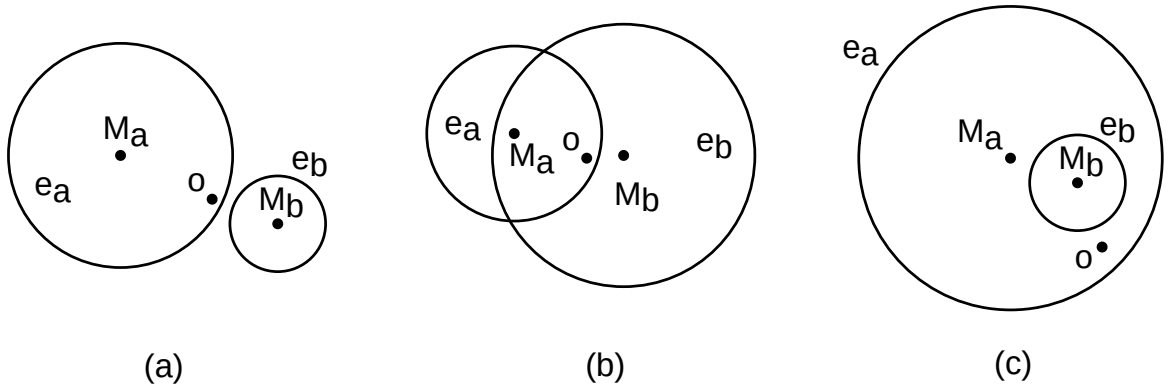

Figure 6: Examples illustrating that an object o is not necessarily associated with the element whose cluster center is closest to o. In particular, in the case of each of parts (a), (b), and (c), o is associated with element e_a whose cluster center M_a is farther away from o than cluster center M_b of element e_b.

Lecture Notes in Computer Science, pages 83–95, Portland, ME, August 1995.

[18] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999. Also University of Maryland Computer Science Technical Report TR–3919, July 1998.

[19] G. R. Hjaltason and H. Samet. Incremental similarity search in multimedia databases. Computer Science Technical Report TR–4199, University of Maryland, College Park, MD, November 2000.

[20] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems*, 28(4):517–580, December 2003.

[21] B. Kamgar-Parsi and L. N. Kanal. An improved branch and bound algorithm for computing *k*-nearest neighbors. *Pattern Recognition Letters*, 3(1):7–12, January 1985.

[22] S. Larsen and L. N. Kanal. Analysis of k-nearest neighbor branch and bound rules. *Pattern Recognition Letters*, 4(2):71–77, April 1986.

[23] C. Merkwirth, U. Parlitz, and W. Lauterborn. Fast exact and approximate nearest neighbor searching for nonlinear signal processing. *Physical Review E (Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics)*, 62(2):2089–2097, August 2000.

[24] G. Navarro. Searching in metric spaces by spatial approximation. *VLDB Journal*, 11(1):28–46, August 2002.

[25] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD Conference*, pages 71–79, San Jose, CA, May 1995.

[26] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, 2006.

[27] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics*, 4(3):182–222, July 1985. Also see *Proceedings of Computer Vision and Pattern Recognition'83*, pages 127–132, Washington, DC, June 1983 and University of Maryland Computer Science Technical Report TR–1372, February 1984.

[28] M. Stonebraker, T. Sellis, and E. Hanson. An analysis of rule indexing implementations in data base systems. In *Proceedings of the 1st International Conference on Expert Database Systems*, pages 353–364, Charleston, SC, April 1986.

[29] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th IEEE International Conference on Data Engineering*, S. Y. W. Su, ed., pages 516–523, New Orleans, LA, February 1996.

[30] P. Zezula, Amato G, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*. Springer-Verlag, Berlin, Germany, 2006.

Hanan Samet received the B.S. degree in engineering from the University of California, Los Angeles, and the M.S. Degree in operations research and the M.S. and Ph.D. degrees in computer science from Stanford University, Stanford, CA. He is a Fellow of the IEEE, ACM, and IAPR (International Association for Pattern Recognition).

In 1975 he joined the Computer Science Department at the University of Maryland, College Park, where he is now a Professor. He is a member of the Computer Vision Laboratory of the Center for Automation Research and also has an appointment in the University of Maryland Institute for Advanced Computer Studies. At the Computer Vision Laboratory he leads a number of research projects on the use of hierarchical data structures for geographic information systems. His research group has developed the QUILT system which is a GIS based on hierarchical spatial data structures such as quadtrees and octrees, the SAND system which integrates spatial and non-spatial data, the SAND Spatial Browser which enables browsing through a spatial database using a graphical user interface, the VASCO spatial indexing applet (found at `http://www.cs.umd.edu/~hjs/quadtree/index.html`), and a symbolic image database system.

His research interests are data structures, computer graphics, geographic information systems, computer vision, robotics, and database management systems. He is the author of the recent book titled "Foundations of Multidimensional and Metric Data Structures" (`http://www.mkp.com/multidimensional`) published by Morgan-Kaufmann, an imprint of Elsevier, in 2006, and of the first two books on spatial data structures titled "Design and Analysis of Spatial Data Structures", and "Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS", both published by Addison-Wesley in 1990. He is an Area Editor of "Graphical Models and Image Processing", and on the Editorial Board of "Image Understanding", "Journal of Visual Languages", "GeoInformatica", and "Journal of Spatial Cognition and Computation".