Depth-First K-Nearest Neighbor Finding Using the MaxNearestDist Estimator[*]

Hanan Samet
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

A description is given of how to use an estimate of the maximum possible distance at which a nearest neighbor can be found to prune the search process in a depth-first branch and bound k-nearest neighbor finding algorithm.

1 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. The most common strategy for nearest neighbor finding employs the depth-first *branch and bound* method (e.g., [5]). Nearest neighbor finding algorithms that incorporate this strategy can be easily extended to find the k-nearest neighbors and this is how we describe them here. These algorithms are generally applicable to any index based on hierarchical clustering. They partition the data into clusters which are aggregated to form other clusters, with the total aggregation being represented as a tree. The search hierarchies used by these algorithms are partly specific to vector data [3], but they can be easily adapted to non-vector data (e.g., [1, 2, 11]), and this is how we present them here.

An alternative strategy is the best-first method (e.g., [4, 6]) which explores the nonobject elements of the search hierarchy in increasing order of their distance from the query object q (hence the name "best-first"). This is achieved by storing the nonobject elements of the search hierarchy in a priority queue in this order. In addition, some of the algorithms (e.g., [4, 6]) also store the objects in a priority queue thereby enabling the algorithms to report the neighbors 1-by-1, and thus there is no need for k to be known in advance.

In contrast, in the depth-first method, the order in which the elements of the search hierarchy are explored is a result of performing a depth-first traversal of the hierarchy using the distance D_k from q to the current k^{th}-nearest object to prune the search. The advantage of this approach over the best-first method is that the amount of storage is bounded by k in contrast to possibly having to keep track of all of the nonobjects (and thus of all of the objects) if all of their distances from q are approximately the same. On the other hand, the advantage of the best-first approach is avoiding to visit nonobject elements that will eventually be determined to be too far from q due to poor initial estimates of D_k.

Implementations of both the depth-first (e.g., [3, 7]) and best-first (e.g., [4, 6]) methods have traditionally used the estimate of the minimum distance at which a nearest neighbor can be found to prune the search. In this paper, we describe the use of an estimate of the maximum possible distance at which a nearest neighbor must be found to prune the search for finding the k nearest neighbors. The main focus is on the depth-first method although we do point out that this estimate can also be used with the best-first method.

The rest of this paper is organized as follows. Section 2 reviews the basic depth-first k-nearest neighbor algorithm. Section 3 describes an estimator of the maximum possible distance at which the nearest neighbor is found, while Section 4 shows how to incorporate it in a depth-first k-nearest neighbor algorithm to eliminate some elements from further consideration. Concluding remarks are made in Section 5.

2 Basic Algorithm

The depth-first k-nearest neighbor algorithm makes use of a list L, initially empty, containing the k current candi-

[*]This work was supported in part by the National Science Foundation under grants EIA-99-00268, EIA-99-01636, IIS-00-86162, and EIA-00-91474.

date nearest neighbors sorted by their distance from query object q. Variable D_k indicates the distance of the current k^{th}-nearest neighbor of q and is initialized to ∞ as we have no candidate neighbors initially. The algorithm is realized by the recursive procedure DFTRAV which is invoked with parameter e initialized to the root of the search hierarchy. In DFTRAV, 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 k^{th}-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 INSERTL(not given here) which also resets D_k if necessary. Otherwise (i.e., e is not a leaf), DFTRAV generates the immediate successors of e, places them in a list $A(e)$, known as the *active list* of child elements of e, and then proceeds to process them one-by-one by calling itself recursively.

```
 1 recursive procedure DFTRAV($e$)
 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 $d(q,o) < D_k$ then INSERTL($o, d(q,o)$)
 6       endif
 7    enddo
 8 else
 9    Generate active list $A$ containing child elements of $e$
10    foreach element $e_p$ of $A$ do DFTRAV($e_p$)
11    enddo
12 endif
```

DFTRAV visits every element of the search hierarchy. Clearly, better performance could be obtained by not visiting every element and its objects when we can show that it is impossible for an element to contain any of the k nearest neighbors of q [3]. For example, letting d be a distance function, 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 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).

Furthermore, given that $d(q,e) \leq d(q,e_0)$ for every object e_0 in e for all nonobject elements e, if we process the elements e_p of the active list $A(e)$ in order of increasing values of $d(q,e_p)$ (i.e., a MINDIST ordering), then once we have found one element e_i in $A(e)$ 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 elements need to be processed, and we exit the loop and backtrack to the parent of e, or terminate if e is the root of the search hierarchy.

[1]This stopping condition ensures that all objects at the distance of the k^{th} nearest neighbor are examined. Of course, 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.

3 The MaxNearestDist Estimator

The modifications to the basic algorithm described in Section 2 use estimates of the minimum possible distance at which a nearest neighbor can be found (i.e., MINDIST) to prune the search process. Fukunaga and Narendra [3] proposed another modification which was to use the maximum possible distance from q to an object in e (referred to as MAXDIST) to tighten the value of the estimate of the distance to the nearest neighbor (i.e., D_1). Larsen and Kanal [8] point out that a better estimate is to use the maximum possible distance from q to its nearest neighbor in e (referred to as MAXNEARESTDIST).

To see the distinction between MINDIST, MAXDIST, and MAXNEARESTDIST, suppose that the search hierarchy consists of minimum bounding hyperspheres. In this case, let r_{max} be the radius of the minimum bounding hypersphere of the objects in e whose center has been determined to be M. It is easy to see that the maximum possible distance from the query object q to an object o in e which serves as the nearest neighbor of q arises when o lies in one of two antipodal positions a and b located diametrically opposite each other on the surface of the hypersphere so that the $(d-1)$-dimensional hyperplane passing through a, b, and M is perpendicular to the line joining M and q. Observe that a and b are equidistant from q and that the distance from q to either of them is $\sqrt{d(q,M)^2 + r_{max}^2}$, the value of MAXNEARESTDIST, which is clearly $\leq d(q,M) + r_{max}$ (e.g., by the triangle inequality), the value of MAXDIST (Figure 1).

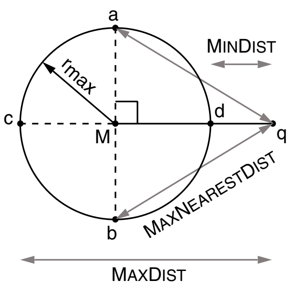

Figure 1: Example of MinDist, MaxDist, and MaxNearestDist for a minimum bounding hypersphere centered at M with radius r_{max} and query point q.

4 Algorithm Using MaxNearestDist

Using MAXNEARESTDIST to tighten the estimate D_k when finding the k nearest neighbors instead of just the nearest neighbor (i.e., $k = 1$) is not a simple matter, although neither Fukunaga and Narendra [3] nor Larsen and Kanal [8] give it any mention. Note that we cannot simply reset D_k to MAXNEARESTDIST(q,e) whenever MAX-

NEARESTDIST$(q,e) < D_k$. The problem is that the distance s from q to some of its k nearest neighbors may lie within MAXNEARESTDIST$(q,e) < s \leq D_k$, and thus resetting D_k to MAXNEARESTDIST(q,e) may cause them to be missed, especially if child element e contains just one object. In other words, we must also examine the values of D_i $(1 \leq i < k)$.

An alternative solution is that whenever we find that MAXNEARESTDIST$(q,e) < D_k$, we reset D_k to MAXNEARESTDIST(q,e) if $D_{k-1} \leq$MAXNEARESTDIST(q,e); otherwise, we reset D_k to D_{k-1}. Nevertheless, this solution is problematic when $D_{k-1} >$MAXNEARESTDIST(q,e) since at this point both D_k and D_{k-1} are the same (i.e., $D_k = D_{k-1}$), and from now on we will never be able to obtain a lower bound on D_k than D_{k-1}. The remedy is to add another explicit check to determine if $D_{k-2} \leq$MAXNEARESTDIST(q,e_p), in which case we reset D_{k-1} to MAXNEARESTDIST(q,e_p); otherwise, we reset D_{k-1} to D_{k-2}. Nevertheless, this remedy is only temporary as it will break down again if $D_{k-2} >$MAXNEARESTDIST(q,e_p). However, we can repeatedly apply the same method until we find the smallest $i \geq 1$ such that $D_i >$MAXNEARESTDIST(q,e_p). Once we locate this value of i, we set D_i to MAXNEARESTDIST(q,e_p) after resetting D_j to D_{j-1} $(k \geq j > i)$.

Unfortunately, the above solution does not guarantee that objects associated with the different $D_j (1 \leq j \leq k)$ values are unique. The problem is that the same object o may be responsible for the MAXNEARESTDIST value associated with both elements e_p and e_a of the search hierarchy that caused MAXNEARESTDIST$(q,e_p) < D_k$ and MAXNEARESTDIST$(q,e_a) < D_k$, respectively, at different instances of time. Of course, this situation can only occur when e_p is an ancestor of e_a. However, it must be taken into account as otherwise the results of the algorithm are wrong.

Another problem is the way the MAXNEARESTDIST estimator is presented, 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 the MAXNEARESTDIST estimator is the minimum of the maximum possible distances to the k-nearest neighbor of the query object, which is not true. Instead, the way in which the MAXNEARESTDIST estimator should be used in k-nearest neighbor finding is to provide bounds for a number of different clusters. Only once we have obtained k distinct such bounds, do we have an estimate on the distance to the k^{th} nearest neighbor.

In order to avoid these problems, and to be able to make use of the MAXNEARESTDIST estimator, we expand the role played by the list L of the k nearest neighbors encountered so far so that it also contains nonobject elements corresponding to the elements of the active list along with their corresponding MAXNEARESTDIST values, as well as objects with their 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 child elements that comprise e's active list along with their corresponding MAXNEARESTDIST values. Moreover, in order to ensure that no ancestor-descendent relationship could possibly hold for any pair of items in L, before we process a nonobject element e_p of the search hierarchy (line 10 in procedure DFTRAV), we remove e_p's corresponding element from L using procedure REMOVEQUEUE (not given here). Therefore, the object o associated with the nonobject element u of L at a distance of MAXNEARESTDIST is guaranteed to be unique. In other words, o is not already in L nor is o associated with any other nonobject element in L. It is also important to note that each of the entries u in L ensures that there is at least one object in the data set whose maximum possible distance from q is the one associated with u.

The new algorithm is given by procedures OPTDFTRAV and OPTINSERTL, which replace DFTRAV and INSERTL, respectively. It lies somewhere between the depth-first and best-first approaches, and thus we call it a *maxnearest depth-first* k-nearest neighbor algorithm. Notice that OPTDFTRAV processes the elements of the active list in increasing order with respect to q using MINDIST. An alternative is to order these elements using MAXNEARESTDIST. However, this ordering has been shown to be not as good as the MINDIST ordering [6, 9, 10], and thus we do not use it.

L is implemented using a priority queue so that accessing the farthest of the k nearest neighbors as well as updating (i.e., inserting and deleting the k nearest neighbor) can be performed without needless exchange operations as would be the case if L was 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., $d(q,i)$), respectively, and fields corresponding to control information specific to the data structure used to implement the priority queue (e.g., a binary heap, etc.). If there are k candidate nearest neighbors (determined by SIZE, not given here), then OPTINSERTL precedes the insertion, which is performed by ENQUEUE (not given here), by first dequeueing the current farthest member (i.e., the *kth*-nearest member) from L using DEQUEUE (not given here). Next, if there are k candidate nearest neighbors after the insertion, then OPTINSERTL resets D_k to the distance of the current farthest nearest neighbor, accessed by MAXL (not given here).

```
1 recursive procedure OPTDFTRAV($e$)
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
```

```
            OPTINSERTL($o,d(q,o)$)
        **endif**
    **enddo**
**else**
    Generate active list $A$ with child elements $e_p$ of $e$
    /* $A$ is sorted in increasing order with respect to $q$
        using MINDIST and processed in this order */
    **foreach** element $e_p$ of $A$ **do**
        /* Attempt to apply MAXNEARESTDIST */
        **if** MAXNEARESTDIST$(q,e_p) < D_k$ **then**
            OPTINSERTL($e_p$, MAXNEARESTDIST$(q,e_p)$)
        **endif**
    **enddo**
    **foreach** element $e_p$ of $A$ **do**
        /*Process $A$ in increasing order */
        **if** MINDIST$(q,e_p) > D_k$ **then**
            **exit_for_loop** /* Prune $e_p$ */
        **else**
            **if** MAXNEARESTDIST$(q,e_p) < D_k$ **or**
              (MAXNEARESTDIST$(q,e_p) = D_k$ **and**
               $D$(MAXL$(L)) = D_k$ **and**
               **not** ISOBJECT($E$(MAXL$(L)$)))) **then**
               REMOVEQUEUE($e_p,L$)
            **endif**
            OPTDFTRAV($e_p$)
        **endif**
    **enddo**
**endif**
```

```
**procedure** OPTINSERTL($e,s$)
/* Insert element (object or nonobject) $e$ at distance $s$
    from query object $q$ into the priority queue $L$ using
    ENQUEUE which assumes that objects have prece-
    dence over nonobjects at the same distance. */
**if** SIZE$(L) = k$ **then**
    $h \leftarrow$DEQUEUE($L$)
    **if not** ISOBJECT($E(h)$) **then**
        **while not** ISEMPTY($L$)
                **and not** ISOBJECT($E$(MAXL$(L)$))
                **and** $D$(MAXL$(L)) = D(h)$ **do**
            DEQUEUE($L$)
        **enddo**
    **endif**
**endif**
ENQUEUE($e,s,L$)
**if** SIZE$(L) = k$ **then**
    **if** $D$(MAXL$(L)) < D_k$ **then**
        $D_k \leftarrow D$(MAXL$(L)$)
    **endif**
**endif**
```

D_k keeps track of the minimum of the distances that have been associated with the entry in L corresponding to q's k^{th}-nearest neighbor. Note that D_k is not necessarily the same as the value currently associated with the entry in L corresponding to q's k^{th}-nearest neighbor, which we denote by $D(L_k)$ (i.e., D(MAXL(L))). The reason is that we cannot guarantee that the MAXNEARESTDIST values of all of e's immediate descendents (i.e., the elements of the active list of e) are smaller than the MAXNEARESTDIST value of e. All we know for sure is that the distance from q to the nearest object in e and its descendents is bounded from above by the MAXNEARESTDIST value of e. In other words, D_k is non-increasing, while $D(L_k)$ can increase and decrease as items are added and removed from L. For example, we see that $D(L_k)$ must increase when element $E(L_k)$ has just two sons e_a and e_b both of whose MAXNEARESTDIST values are greater than $D(L_k)$ (see Figure 2 with donut-like nonobjects e_i where MAXNEARESTDIST$(q,e_i) = d(q,M_i) + r_{i,min}$).

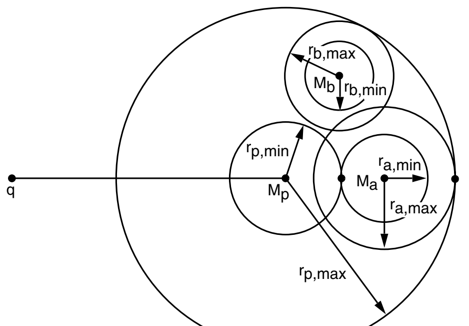

Figure 2: Example illustrating how $D(L_k)$ increases when the MaxNearestDist values of the tw o sons e_a and e_b of e_p are greater than the current value of $D(L_k)$, which is the MaxNearestDist value of e_p.

We incorporate the MAXNEARESTDIST estimator by resetting D_k to $D(L_k)$ whenever upon insertion of a nonobject element e_p into L, with its corresponding MAXNEAREST-DIST value, we find that L has at least k entries and that $D(L_k)$ is less than D_k as this corresponds to the situation that MAXNEARESTDIST$(q,e_p) < D_k$. Note that D_k is also reset in procedures DFTRAV and INSERTL upon insertion of an object into L when L has k entries by noting that MAX-NEARESTDIST of an object o is just $d(q,o)$. Thus the unexplored nonobject elements of the active list can be used for pruning in the sense that the distances from q of the farthest objects within them that can serve as the nearest neighbors enable us to calculate an upper bound D_k on the distance of the k^{th}-nearest neighbor. It is important to distinguish between D_k which is used when pruning with MINDIST and $D(L_k)$ which serves as the basis of the MAXNEARESTDIST estimator. In particular, we see that defining D_k in terms of the minimum of D_k and $D(L_k)$ ensures that there are k distinct objects (even though they may not all have been identified) whose maximum distance from q is $\leq D_k$. Of course, we do not reset D_k upon explicitly removing a nonobject element from L as D_k is already a minimum and thus cannot decrease further as a result of the removal of a nonobject element although it may decrease upon the subsequent insertion of an object or nonobject.

Note that nonobjects can only be pruned on the basis of their MINDIST values, in which case they should also be removed from L. Moreover, the fact that we are using the MINDIST ordering means that once we prune one of the child elements of nonobject element e in the active list $A(e)$, we can prune all remaining elements in $A(e)$ since they all have larger MINDIST values as $A(e)$ is sorted in increasing MINDIST order. Therefore, all of these elements should be removed from L as well since each of their MAXNEAREST-DIST values is greater than its corresponding MINDIST value and hence also greater than D_k. However, there is really no harm in not removing any of the pruned nonobjects from L as neither the pruned nonobjects nor their descendents will ever be examined again. Nevertheless, the pruned nonobjects take up space in L, which is why it may be desirable to remove them anyway.

The drawback of the solution that we have described is that the maximum size of L has grown considerably since it is no longer k. Instead, assuming that pruned nonobjects have been removed from L, the maximum size of L is k plus the maximum number of possible active list elements over all levels of the search hierarchy. For example, assuming N data items and that the clustering method makes use of a tree-like search hierarchy with m as the branching factor, then the maximum size of L is $O(k+m\cdot\log N)$ and is attained when the depth-first search algorithm first encounters an object at the maximum depth, which is $O(\log N)$. Actually, it is interesting to note that the cost of increasing the size of L to include the maximum number of possible active list elements is not such a drawback as this amount of storage is needed by the recursion anyway due to the unexplored paths that remain at each level while descending to the deepest level of the search hierarchy.

Nevertheless, the facts that only nonobject elements with MINDIST values $\leq D_k$ are ever processed by the algorithm and that all of these nonobject elements have the same or larger corresponding MAXNEARESTDIST values, mean that any nonobject element e whose MAXNEARESTDIST value is greater than the current value of D_k should not be inserted in L as D_k is nonincreasing, and thus were e to be inserted in L it would never be examined subsequently thereby implying that there will never be an explicit attempt to remove it. In fact, there is also no need to insert e in L when its MAXNEARESTDIST value is equal to the current value of D_k, regardless of how many elements are currently in L, as such an insertion won't enable us to lower the known value of D_k so that more pruning will be possible in the future. Otherwise, there is no reason to use L to keep track of the MAXNEARESTDIST values of the nonobjects.

Moreover, the fact that only the first k elements of L are ever examined by the algorithm (i.e., retrieved from L) when updating D_k in the case of an insertion into L means that there is no need for L to ever contain more than k elements. This simplifies the algorithm 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 child elements that comprise e's active list along with their corresponding MAXNEARESTDIST values, 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 whose corresponding MAX-NEARESTDIST values are lower than that of e and thereby resulted in resetting D_k.

The only tricky case is ensuring that a nonobject e is actually in L when we are about to attempt to explicitly remove e from L. In other words, we want to ensure that e has not already been removed implicitly. Of course, if e's MAXNEARESTDIST value is $> D_k$, then there is no need for action as it is impossible for e to be in L, and thus we are guaranteed that e was implicitly removed from L. However, when e's MAXNEARESTDIST value is $\leq D_k$, and there are several elements in L with distance D_k, we do not want to needlessly search for e as may be the case if e had already been implicitly removed from L by virtue of the insertion of a closer object or a nonobject with a smaller MAXNEARESTDIST value. This needless search can be avoided by adopting some convention as to which element of L at distance D_k should be removed when there are several nonobjects in L with 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 have the same distance, the nonobjects appear closer to the maximum entry in the priority queue L (i.e., MAXL(L)). In particular, we stipulate that whenever insertion into a full priority queue results in dequeueing a nonobject element b, 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', or L is empty. Note that D_k is only reset if exactly one entry has been dequeued and the distance of the new MAXL(L) entry is less than D_k. 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.

Following this convention, when a nonobject e is to be removed explicitly from L and e's MAXNEARESTDIST value is $< D_k$, then e has to be in L as it is impossible for e to have been removed implicitly as D_k is nonincreasing. Therefore, we remove e and decrement the size of L. On the other hand, the situation is more complicated when e's MAXNEARESTDIST value is equal to D_k. First, if the maximum value in L (i.e., MAXL(L)) is less than D_k, then e cannot be in L, and we do not attempt to remove e. Such a situation arises, for example, when we dequeue more than one nonobject due to having several nonobjects at distance D_k. Second, if the maximum value in L (i.e., 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. 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.

The advantage of expanding the role of L to contain nonobjects as well, instead of just containing objects, is that without this expanded role, when L contains h ($h < k$) objects, then all remaining entries in L (i.e., L_i ($h < i \leq k$) are ∞. Therefore, as long as the remaining $k-h$ entries in L correspond to some nonobjects, we have a lower bound D_k than ∞. Moreover, the nonobjects in L often enable us to provide a lower bound D_k than if all entries in L were objects. In particular, this is the case when we have nonobjects with smaller MAXNEARESTDIST values than the k objects with the k smallest distance values encountered so far.

Observe that the way in which we incorporated the MAXNEARESTDIST estimator in OPTDFTRAV enables the use of its result at a deeper level than the one at which it is calculated. In particular, the use of L to store the MAXNEARESTDIST values of some active nonobject elements means that a MAXNEARESTDIST value of an unexplored nonobject at depth i can be used to help in pruning objects and nonobjects at depth $j > i$. This is a significant improvement over the depth-first algorithm in DFTRAV where the MAXNEARESTDIST value of a nonobject element at depth i could only be used to tighten the distance to the nearest neighbor (i.e., for $k = 1$), and to prune nonobject elements at larger MINDIST values at the same depth i.

5 Concluding Remarks

Using the MAXNEARESTDIST estimator in the depth-first k-nearest neighbor algorithm provides a middle ground between a pure depth-first and a best-first k-nearest neighbor algorithm. In particular, assuming N data items, the priority queue implementation of L in the maxnearest depth-first algorithm behaves similarly to the priority queue *Queue* in the best-first k-nearest neighbor algorithm except that the upper bound on L's size is k, while the upper bound on *Queue*ś size is $O(N)$. In contrast, in both the pure and the maxnearest depth-first algorithms, the worst-case storage requirements only depend on the nature of the search hierarchy (i.e., the maximum height of the search hierarchy which is $O(\log N)$), instead of on the size of the data set, as is the case for the best-first algorithm. The best-first algorithm can also be adapted to use MAXNEARESTDIST [10].

References

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

[2] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. 23rd Int. Conf. on Very Large Data Bases (VLDB)*, pages 426–435, Athens, Greece, Aug. 1997.

[3] K. Fukunaga and P. M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Trans. Comp.*, 24(7):750–753, 1975.

[4] A. Henrich. A distance-scan algorithm for spatial access structures. In *Proc. 2nd ACM Workshop on Geog. Inf. Sys.*, pages 136–143, Gaithersburg, MD, Dec. 1994.

[5] F. S. Hillier and G. J. Lieberman. *Introduction to Operations Research*. Holden-Day, San Francisco, 1967.

[6] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Sys.*, 24(2):265–318, 1999. Initial version in *Advances in Spatial Databases — 4th Int. Symp. SSD'95*, pages 83–95, Portland, ME, August 1995 and also Springer-Verlag Lecture Notes in Computer Science 951.

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

[8] S. Larsen and L. N. Kanal. Analysis of k-nearest neighbor branch and bound rules. *Pat. Rec. Ltrs.*, 4(2):71–77, 1986.

[9] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. ACM SIGMOD Conf.*, pages 71–79, San Jose, CA, May 1995.

[10] H. Samet. *Foundations of Multidimensional Data Structures*. To appear.

[11] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Proc. Ltrs.*, 40(4):175–179, 1991.