Proceedings of the 4th Symposium on Spatial Databases, Portland, Maine, Aug. 1995, eds. Egenhofer and Herring, Lecture Notes in Computer Science 951, Springer-Verlag, Berlin, 1995, pp. 83–95

Ranking in Spatial Databases*

Gísli R. Hjaltason and Hanan Samet

Computer Science Department, Center for Automation Research, and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA

Abstract. An algorithm for ranking spatial objects according to increasing distance from a query object is introduced and analyzed. The algorithm makes use of a hierarchical spatial data structure. The intended application area is a database environment, where the spatial data structure serves as an index. The algorithm is incremental in the sense that objects are reported one by one, so that a query processor can use the algorithm in a pipelined fashion for complex queries involving proximity. It is well suited for k nearest neighbor queries, and has the property that k needs not be fixed in advance.

1 Introduction

Indexes are used in databases to facilitate retrieval of records with similar values. For a particular attribute, an index yields an ordering of all records in increasing (or decreasing) order of the attribute value. Extending this idea to more than one attribute is a bit complex. One approach is to make the first attribute a primary attribute and the additional attribute a secondary attribute. Thus we first sort the records according to the value of the first attribute and we break ties by use of the second attribute. This is fine as long as we only want the records sorted by the value of the first attribute. If we want the records ordered by the value of the second attribute, then our index is useless as consecutive records obtained by the index are not necessarily ordered by the value of the second attribute. One solution is to build an additional index on the second attribute. This is feasible but does take up more space.

The solution of adding a second index is acceptable as long as queries do not make use of a combination of the attribute values. Such a combination is generally meaningless if the dimensional units of the attribute values differ. For example, if one attribute is age and the other is weight, then the corresponding dimensional units could be years and pounds. In this case, we are not likely to try to determine the nearest record to the one with name John Jones in terms of age and weight as we don't have a commonly accepted notion of the meaning of the year-pound unit.

Spatial databases are distinguished from conventional databases, in part, by the fact that some of the attributes are locational and in which case they have

* This work was supported in part by the National Science Foundation under grants IRI-92-16970 and ASC-93-18183.

the same dimensional unit. More importantly, this common dimensional unit is distance in space. The distance unit is the same regardless of the dimensionality of the space spanned by the locational (i.e., spatial) attributes of the records as long as they cover the same space. What this means is that if we combine the attributes, and seek to determine the nearest record of type t to the one with name Chicago, then the corresponding unit would be distance regardless of whether there are two or three (or even more) locational attributes associated with t. Note that just because an attribute has a dimensional unit of distance does not make it a locational attribute. For example, size attributes are also measured in terms of distance yet they are not locational. Thus attributes corresponding to a person's height and waist are not locational attributes and cannot be combined.

In addition, different spatial databases can be distinguished according to the types of records that they store. There are two types: points and objects. We define the former to have a zero volumetric measure, while the latter have a nonzero volumetric measure. In other words, the latter have an extent while the former do not (i.e., they are discrete). Note that the records in a conventional database are always discrete, and can be viewed as points in a higher dimensional space. The difference is that in the case of spatial data, the dimensional unit of the attribute is distance in space.

Regardless of the distinction between the types of data stored in a spatial database, we are often interested in ordering the records on the basis of some combination of the values of the locational attributes. This ordering is used to facilitate storage of the records as the storage methods are inherently one-dimensional. It is desirable for this ordering to also preserve proximity in the sense that records that are close to each other in the multidimensional space formed by ranges of the values of the locational attributes are also close to each other in the ordering. Of course, if there is just one locational attribute, then the ordering is the same as that used for a non-locational attribute.

An example of such an ordering technique is hashing. There are two variants of hashing, depending on whether the resulting ordering is explicit or implicit. An explicit ordering results from the use of a particular mapping from the higher dimensional space to a one-dimensional space. An example mapping is one that interleaves the individual bits in the binary representation of the locational attribute values. Such mappings result in what are known as space-filling curves [5] (e.g., Peano, Hilbert, Sierpinsky, etc.) although no curve has the property that all records that are close to each other in the multidimensional space formed by the ranges of the locational attribute values are also close to each other in the range of the mapping.

Bucketing methods are examples of an implicit ordering. In this case, the records are sorted on the basis of the space that they occupy (i.e., the space formed by the values of their locational attribute) and are grouped into cells (i.e., buckets) of a finite capacity. Of course, if there is just one locational attribute, then the implicit and explicit orderings are equivalent. When the records are such that they also have an extent (e.g., non-point spatial objects), then the notion of a bucket is more meaningful. In particular, there are two possible

approaches [14].

The first approach finds a minimum bounding box for the object. These boxes may be subsequently aggregated by use of hierarchies. In such a case, the minimum bounding boxes may not necessarily be disjoint. The drawback is that an object is associated with just one bounding box. Thus if we are given a particular point p, and we search for an object that contains p, then just because we don't find an object that contains p in a bounding box b containing p does not mean that objects in other bounding boxes do not contain p.

An alternative approach decomposes the objects so that the bounding boxes that contain them are disjoint. Once again, these boxes may be subsequently aggregated into hierarchies. Now, for each point p there is just one bounding box b that contains p and if none of the objects in b contain p, then none of the objects in the database will contain p and the query fails. The drawback is that an object can be decomposed into several pieces and hence associated with many boxes. Thus if we want to determine which objects are associated with a region that spans several bounding boxes, then we may report a particular object more than once. For such queries we must have a post-processing step that removes duplicate answers. The process of removing duplicate may require a process as complex as sorting although, depending on the nature of the object, other methods may be applicable (e.g., [2]).

The ordering provided by an index is useful for ranking the data based on its closeness to a particular value v of the attribute a. The ability to perform the ranking does not depend on whether a record r exists in the database such that attribute a of r has value v. Value v serves as a reference point for the ranking.

In this paper, we focus on the issue of ranking in spatial databases. For the moment, assume that we have just one attribute and that it can be locational or non-locational. In this case the explicit and implicit indexes are equivalent, and we can derive the ranking directly from the index for the attribute. In particular, the index is obtained by sorting the data with respect to a particular reference point (usually the smallest possible value — e.g., zero for an attribute whose value is of type ratio). For example, consider the non-locational attribute weight and its corresponding index. Suppose that the database records correspond to individuals, and we want to find all individuals in the order of the closeness of their weight to that of John Smith whose weight is 150 pounds. The answer is computed by looking up the value 150 in the index and then proceeding in two directions along the index to get the nearest individuals by weight in constant time. We do not have to rebuild the index if we want to be able to answer the next query which deals with Sam Jones whose weight is 200 pounds.

In the case of more than one locational attribute all of whose values are of type distance, we wish to obtain a ranking of the records in terms of their distance from a particular value v of the locational attributes. If the index is explicit, then we cannot derive this ranking directly from the index for the locational attributes. As an example, we could have built an index on the basis of the distance of the records from a particular reference point P_1 using a given distance metric. However, if we want to obtain the records in order with respect

to a new reference point P_2, we must resort them. In other words, we cannot simply say that their distance from P_2 is equal to the addition or subtraction of some constant equal to the distance from P_1 to P_2 depending on the relative position of the record with respect to P_1 and P_2, which is what is done when there is just one attribute (regardless of whether or not it is locational). Thus we have to rebuild the index, which is a costly process if we need to do it for each query. Thus, what is usually done is to use an implicit index such as the one discussed earlier that is based on sorting the objects with respect to the space that they occupy rather than with respect to each other or some fixed reference point.

Ranking queries are frequently used in spatial databases (e.g., in browsing applications). For example, we may wish to find all the houses in the database in the order of their distance from a point at location P. Often the desired ranking is partial. For example, we may wish to find the nearest city of population greater than 100,000 to Las Vegas. In this case, if we make use of the index on the locational attributes corresponding to the location of the cities, then we want to obtain the cities in the order of the cities' distance from Las Vegas. The population of the cities is examined in increasing order of their distance from Las Vegas. The process ceases once the condition on the value of the non-locational population attribute is satisfied. It should be clear that the query to find the closest city to Las Vegas is also a partial ranking query. Observe that the key to the utility of the ranking process is that if the closest record does not satisfy the query condition, then we can continue the search from where we computed the current answer. We do not restart the search again from reference point of the index.

In this paper we show how to respond to ranking queries in a spatial database when the spatial data is organized using an implicit index. There are a number of possible solutions depending on the nature of the implicit index. We present a general solution which is designed to minimize the number of blocks of the underlying decomposition that are examined. In order to be able to analyze its execution cost, we must have a concrete representation. We choose a representation that decomposes the objects so that the bounding boxes that contain them are disjoint. Moreover, we assume a regular decomposition such as that provided by the PMR quadtree [9]. Of course, other representations (e.g., the R^+-tree [15]) could also have been used as well as an implementation where the bounding boxes are not disjoint (e.g., an R-tree [6]). Our methods are equally applicable to these representations.

2 Data Structure

As we mentioned earlier, our algorithm was developed for the PMR quadtree but can be adapted to many other hierarchical spatial data structures that make use of what we term *container block*. This term is used here to denote an area in space which may itself be decomposed further on basis of the number or particular nature of the spatial objects that it contains. Examples of such structures include

R-trees [6], R$^+$-trees [15], and k-d-b-trees [10].

The PMR quadtree uses a regular decomposition of space to index spatial objects. Each quadtree block is a square, or a hypercube in higher dimensions. Leaf blocks contain the spatial objects (or pointers to them), whereas non-leaf blocks are decomposed into 2^d sub-blocks, where d is the number of dimensions. Fig. 1 presents an example two-dimensional PMR quadtree with a splitting threshold of one where the objects are points representing cities. The cities are inserted in the order Chicago, Mobile, Toronto, Buffalo, Denver, Omaha, Atlanta, Miami. The inherent definition of a quadtree is a representation that recursively decomposes space into congruent blocks until some condition is satisfied. The retrieval of the blocks that comprise the quadtree is facilitated using a number of different access structures [13]. The most common access structures are a tree having four sons at each level (see Fig. 1b) or a tree such as a B$^+$-tree [4] that is based on finding an ordering on the blocks. An example of such an ordering is that achieved by interleaving the bits comprising the binary representations of the x and y coordinates of the upper-left corners of each block. These numbers are then used as keys in the B$^+$-tree. We use the former implementation in the discussion of our algorithm, although it also works for the latter implementation.

PMR quadtrees differ from other quadtree variants in the way in which object insertions trigger decomposition of quadtree blocks. In particular, if, upon insertion of an object, the number of objects in a leaf block l intersected by the object exceeds a threshold value s (similar to a bucket capacity but not quite the same concept), then l is split once and the objects in l are reinserted into the new sub-blocks of l that they intersect. Note that the number of objects in a leaf block may exceed the threshold value. However, the number of objects in a leaf block at depth i is bounded by $s + i$, assuming there is no limit on the depth of the tree.

3 Overview of the Algorithm

We present a top-down solution. An alternative is to use a bottom-up solution. In this case, the algorithm locates the block b containing the query object q and then finds the nearest object o by examining the adjacent neighboring blocks of b in a clockwise order. Depending on the nature of the distance metric that is employed, we may have to examine blocks that are not immediately adjacent to b. This technique is termed *bottom-up* because we are obtaining the neighbors using neighbor-finding techniques [12] that do not restart the search at the root of the tree. In the case of a pointer-based (i.e., a tree) quadtree representation, they have been shown to visit a constant number of blocks for each neighbor-finding operation. This method could be very fast especially if o is in block b or one of the brothers of b. However, it may have to visit all of the blocks around the node [7]. Worse of all, if we need the next closest object, then we have to restart the search from the beginning rather than from where we last left off, making it unsuitable for ranking. In contrast, our algorithm can simply continue the search from the object it last found.

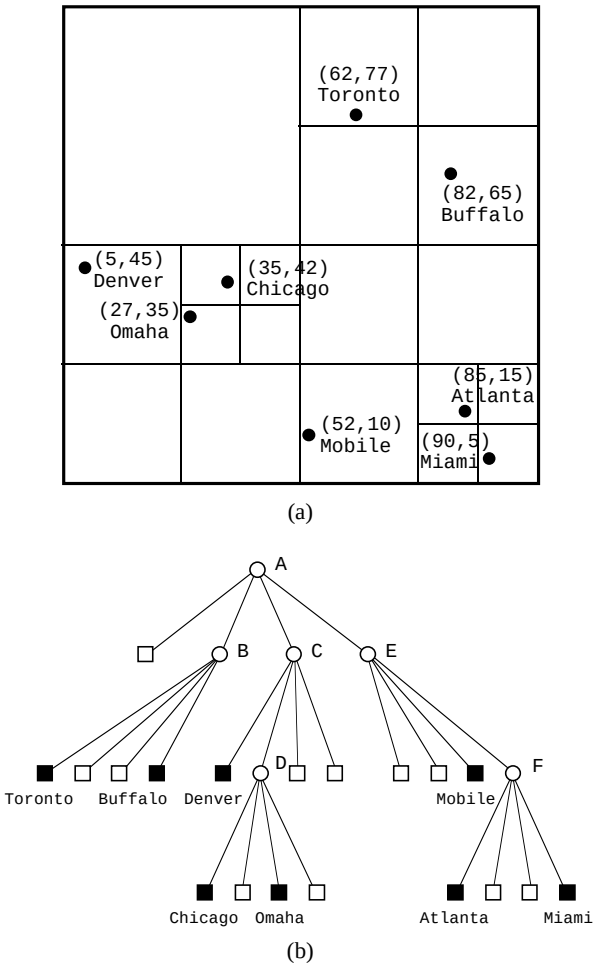


Fig. 1. A PMR quadtree representing points corresponding to cities. (a) The block decomposition induced by the quadtree, and (b) a tree access structure for the blocks in (a)

The key to the efficiency of the bottom-up method is that it works on the principle that if block b is empty, then the three siblings of b must contain at least $s+1$ objects or we would not have decomposed the space. This acts as a pruning device on the search. However, as we are interested in obtaining a ranking, we make use of the top-down method. First, we find leaf nodes containing q. We then use the recursion to keep track of what blocks have already been seen. Once we visit a leaf node, we also want to remember the objects that we have already encountered in the block which may still not yet be the closest ones. We achieve this by modifying the top-down algorithm to maintain a priority queue to record the blocks whose descendants have not been visited yet as well as the objects

which have not yet been visited.

Using the top-down method, it is easy to find a leaf node containing q. Nevertheless, we need to be able to extend this technique to find the nearest object as the leaf may be empty or the other object in the leaf may be quite far from the query object. The problem here is that we have to unwind the recursion to find the nearest object. However, if we want to find the second nearest object, then the solution becomes even tougher. To resolve this problem, we replace the recursion stack where the next block to be examined is the block nearest to q with a priority queue. The key to our solution is that the objects are also stored in the priority queue. Once a leaf block b is encountered, we attempt to insert the objects stored in b into the priority queue. We can only insert an object o if it has not already been reported. This can be determined by checking if o's distance from the query object q is less than the distance of b from q. In this case, o was contained in a block c which was closer to q than b, and hence already processed earlier.

Observe that the data objects as well as the query objects can be of arbitrary type (e.g., points, rectangles, polygons, etc.). The only requirement is that there be a distance function between the query object type and the object type stored in the index (feature metric), and the query object type and the container block type (block metric). The two distance functions must be consistent with each other. Consistency means that for a feature f with a distance d from the query object q, there must exist a block b containing f such that the distance from b to q is less than or equal to d. This will hold if both distance functions are based on the same distance metric, of which some common examples are the Euclidean, Manhattan and Chessboard metrics. The consistency assumption also means that the distance from a query object to a block that contains it is zero.

The algorithm works for any dimension, although the examples we give are restricted to two dimensions. Also the query object need not be in the space of the dataset.

4 Algorithm

We first consider a regular recursive top-down traversal of the index to locate a leaf block containing a query object. Note that there could be more than one such block. The traversal is initiated with the root block as the second argument.

```
FindLeaf(QueryObject, Block)
1 if QueryObject is in container block Block then
2   if Block is a leaf block then
3     Report leaf block Block
4   else
5     for each Child block of container block Block do
6       FindLeaf(QueryObject, Child)
7     enddo
8   endif
9 endif
```

The first task is to extend the algorithm to find the nearest object to the query object. In particular, once the leaf block containing the *QueryObject* has been found in line 3, we could start by examining the objects contained in that block. The object closest to the query object might reside in another quadtree block. Finding that block may in fact require unwinding the recursion to the top and descending again deeper into the tree. Furthermore, once that block has been found, it doesn't aid in finding the next nearest object.

To resolve this dilemma, we replace the recursion stack of the regular top-down traversal with a priority queue. In addition to using the priority queue for container blocks, objects are also put on the queue as leaf blocks are processed. The key used to order the elements on the queue is their distance from a query object. In order to distinguish between two elements at an equal distance from the query object, we adopt the convention that blocks are ordered before objects, while different objects are ordered according to some arbitrary (but unique) rule. This makes it possible to avoid reporting a particular object more than once, which is necessary when using a disjoint decomposition where an object may be associated with more than one block (e.g., PMR quadtree, R^+-tree).

A container block is not examined until it reaches the head of the queue. At this time, all blocks and objects closer to the query object have been looked at. Initially, the container block spanning the whole index space is the sole element in the priority queue. In subsequent steps, the element at the head of the queue (i.e., the closest element not yet examined) is retrieved until the queue has been emptied.

```
    IncNearest(QueryObject, SpatialIndex)
 1  Queue ← NewPriorityQueue()
 2  Block ← RootBlock(SpatialIndex)
 3  Enqueue(Queue, Dist(Block, QueryObject), Block)
 4  while not IsEmpty(Queue) do
 5     Element ← Dequeue(Queue)
 6     if Element is a spatial object then
 7        while Element = First(Queue) do
 8           DeleteFirst(Queue)
 9        enddo
10        Report Element
11     elsif Element is a leaf block then
12        for each Object in leaf block Element do
13           if Dist(Object, QueryObject) ≥ Dist(Element, QueryObject) then
14              Enqueue(Queue, Dist(Object, QueryObject), Object)
15           endif
16        enddo
17     else /* Element is a non-leaf container block */
18        for each Child block of container block Element in SpatialIndex do
19           Enqueue(Queue, Dist(Child, QueryObject), Child)
20        enddo
```

21 **endif**
22 **enddo**

Lines 1–3 initialize the queue. In line 10, the next closest object is reported. At that point, some other routine (such as a query processor) could take control, possibly resuming the algorithm at a later time to get the next closest object, or alternately terminate it if no more objects are desired.

Recall that for some representations, a spatial object may span several container blocks. The algorithm must thus guard against objects being reported more than once [2]. The test (i.e., the **if** statement) in line 13 ensures that objects that have already been reported are not put on the queue again. For this to work properly, blocks must be retrieved from the queue before spatial objects at the same distance. Otherwise, a feature may be retrieved from the queue before a block b containing it that is at the same or less distance from the query object. When the feature then is encountered again in block b, there is no way of knowing that it has already been reported. The loop in lines 7–9 eliminate duplicate instances of an object from the queue. By inducing an ordering on features that are at the same distance from the query object, all of the instances of an object will be clustered at the front of the queue when the first instance reaches the front. The reason we check for duplicates in this manner is that for many representations of a priority queue it is not efficient to test for membership. Thus, the removal of duplicates is largely a byproduct of the algorithm.

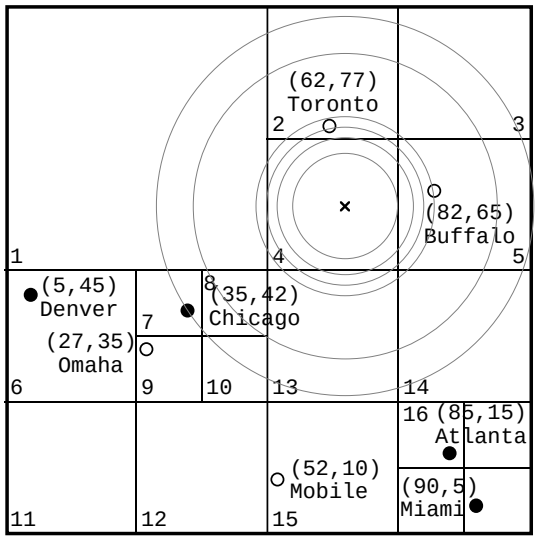
We now give an example to illustrate how the algorithm works. Consider the simple database given in Fig. 2a containing two-dimensional point data. Assuming a Euclidean distance metric, we want to "find the city closest to the point `(65,62)` which has a population of at least 1 million". In our scenario, a query processor interacts with our algorithm to retrieve cities in the order of their distance from the point. Note that the algorithm inserts a city c into the queue even if its population is not high enough to satisfy our query condition. The reason is that checking for the satisfaction of this condition would require a database access. Such an access might be unnecessary as c's distance from the query point may result in c not coming to the front of the queue by the time the algorithm terminates (i.e., by the time enough answers have been output).

Figure 2b shows a PMR quadtree with a splitting threshold value of 1 containing the points corresponding to the cities. Cities with a population of more than 1 million are denoted with solid dots and the query point is denoted with with an 'x'. Several concentric circles are drawn around the query point to make relative distances more obvious. Most of the leaf blocks are labelled with a number. In the description below, a PMR quadtree block is denoted by its depth and the label in its North-Westernmost descendant leaf block. The root block is thus denoted by 0/1 and its NE son by 1/2. The elements in the priority queue are listed within brackets in the order of their distance from the query point.

Initially, the queue contains only the root block, i.e., [0/1]. In the first step, the root block is retrieved from the queue, and as it is a non-leaf block, its sub-blocks are put on the queue: [1/2, 1/13, 1/1, 1/6]. Next, the block 1/2 is dequeued, and its sub-blocks enqueued: [2/4, 2/5, 1/13, 2/2, 1/1, 2/3, 1/6]. In

City	Pop.	Pos.
Atlanta	4,129	(85,15)
Buffalo	764	(82,65)
Chicago	6,532	(35,42)
Denver	1,381	(5,45)
Mobile	504	(52,10)
Omaha	416	(27,35)
Toronto	904	(62,77)
Miami	5,250	(90,5)

(a)



(b)

Fig. 2. Example data set and nearest neighbor query

the next step, the leaf block 2/4 is dequeued, but it contains no objects. The leaf block 2/5, however, contains Buffalo, so Buffalo is inserted in the queue: [1/13, 2/2, 1/1, 2/3, Buffalo, 1/6]. In the next three steps, the sub-blocks of 1/13 are put on the queue, the leaf 2/13 is retrieved from the queue but contains no objects, and the city Toronto is enqueued as the leaf block 2/2 is processed: [1/1, Toronto, 2/14, 2/3, Buffalo, 1/6, 2/15, 2/16]. No action is taken as 1/1 is dequeued since it is empty, but Toronto is the first city to be reported to the query processor. The query processor discards it as it has a population less than 1 million and requests the next closest city. The top two elements on the queue, 2/14 and 2/3, are empty leaf blocks, so no action is taken. Next, Buffalo is reported to the query processor but its population is too low. At this point, the queue contains [1/6, 2/15, 2/16]. Now, 1/6 is taken off the queue and its sub-blocks enqueued, resulting in [2/7, 2/15, 2/16, 2/12, 2/6, 2/11]. The sub-blocks of 2/7 are then put on the queue, resulting in [3/8, 3/10, 3/7, 3/9, 2/15, 2/16, 2/12, 2/6, 2/11]. The blocks 3/8 and 3/10 are empty, but 3/7 contains Chicago, so it is put on the queue: [Chicago, 3/9, 2/15, 2/16, 2/12, 2/6, 2/11]. Finally, Chicago is reported to the query processor, which terminates the search and returns Chicago as the result of the query.

5 Analysis

Our solution to the problem of finding the nearest object is not more efficient than other known methods [3, 7]. However, it is more general in several respects.

The algorithm presented in [3] only works with point data and relies on a specialized data structure to achieve optimality in execution time for approximate nearest neighbor queries. This structure is static and must be rebuilt if more points are introduced. In addition, it is not amenable to practical implementation. Thus the authors implemented a greatly simplified data structure (thereby sacrificing the optimality guarantee of their algorithm while still yielding an approximate answer) that resembles k-d trees, and also use a priority queue in the query process. In contrast, our algorithm can be used for arbitrary data objects as well as a large class of spatial indices. Of course, its level of efficiency may depend on the type of spatial index used. The main advantage of our method compared to the one proposed in [7] is that the latter can not be efficiently used to find several of the nearest objects, only the nearest. Also, that method relies on a quadtree-like decomposition. The algorithm presented in [11] is limited to points as query objects and the R-tree as spatial index, although it may possibly be extended to work for a wider class of query objects and spatial indices.

The algorithm that we presented can be used to find the k nearest neighbors to a query object. However, in our case, the k is not fixed a-priory. This is in contrast with the algorithm in [11] for finding k nearest neighbors. In particular, once it has computed the k nearest neighbors, if the $k+1^{\text{st}}$ nearest neighbor is desired, then the algorithm must be restarted anew.

The analysis below, although incomplete, gives an indication of the worst-case behavior of the algorithm. Various simplifications are made to ease the task. First, we assume that calculating the distance metric takes a constant amount of time. This is true for simple objects such as points and lines, but may not be true for more complex ones (e.g., polygons).

Second, the spatial index is assumed to have some of the properties of the PMR quadtree. Suppose that there are N objects. For some object types (e.g., points, lines) it can be shown that under certain assumptions on the data distribution and the tree depth, the number of blocks in a PMR quadtree is proportional to N [8]. We also assume that the objects in question are already stored in a spatial index and ignore the cost of the preprocessing needed to build the index.

In order to complete our analysis of the space requirements of the algorithm, we need to know the maximum size of the priority queue. Let us consider the queue at an arbitrary time during the execution of the algorithm, and let d be the distance from the object at the head of the queue to the query object. All of the objects in the queue are at a distance of at least d from the query object and are contained in blocks at a distance of at most d from the query object (these are blocks that have been retrieved from the queue and processed). A worst-case scenario is such that all leaf blocks containing objects are closer to the query object than is the nearest object. In this case, all objects will be inserted into the queue before the nearest one is found. This gives a worst-case bound of $O(N)$ on the size of the queue. However, this is a pathological case, which is unlikely to arise. In practice, the maximum size of the queue is much smaller.

If all the objects need to be ranked by their distance from the query object,

then the execution time of the algorithm is at worst $O(N \log M)$ where N is the number of leaf blocks in the spatial index and M is the maximum number of items in the priority queue. This assumes a priority queue implementation where update operations take $O(\log M)$ time. As discussed above, M is $O(N)$ in the worst case, which gives a bound of $O(N \log N)$. This compares favorably with one-dimensional sorting algorithms.

An alternative solution for ranking all the objects is to compute the distance for all of them from the query object and then to sort them using a conventional sorting technique. The cost of this is $O(N \log N)$ where N is the number of objects. In contrast, our ranking algorithm has the advantage that it doesn't have to retrieve all of the objects at once. It is dynamic. Also, we can achieve a better result than $O(N \log N)$ in practice as often we don't sort on the objects; instead, we sort on the container blocks. This is quite important when executing in a disk-based environment as the inspection of a container blocks often does not require us to examine their contents which may require a disk access.

In our ranking algorithm, container blocks are inserted in the priority queue even though they may be empty leaf blocks. We could examine blocks before putting them on the queue and just insert the non-empty ones. The problem here is that if we were executing in a disk-based system, then we would require a disk access every time we check if a container is empty. In contrast, when we insert all the blocks into the priority queue without regard to their contents, we may not have to look at many of them as they may get pruned from the search by virtue of their distance from the query object (i.e., if the search is terminated after finding an object closer to the query object). However, if we want a ranking of all of the objects, then it may be advantageous to inspect blocks before putting them on the queue, since then fewer priority queue operations are needed in addition to the queue being smaller.

For a partial ranking of the objects, our algorithm visits a minimal number of container blocks in the sense that given that the k^{th} nearest neighbor is at a distance of d_k from the query object q, only the container blocks that lie completely or in part within d_k of q have had their contents examined by the time the k nearest neighbors have been found. However, note that all of the container blocks could be within d_k of q, regardless of the value of k. Thus the worst-case execution time is the same as for finding a total ranking, $O(N \log N)$.

6 Conclusion

The algorithm presented in this paper was designed to work in the SAND [1] spatial database environment, where a PMR quadtree is used as the underlying spatial index. However, the algorithm is not limited to a PMR quadtree. It should work (with minor modifications) for a wide class of spatial indices, that includes R-tree variants and k-d-b-trees. We have already successfully adapted the algorithm to work with an R*-tree index. The basic requirement that a spatial index must satisfy for the algorithm to be useful is that the spatial index decomposes space into blocks that are organized hierarchically in a tree-like

fashion. Of course, much of our analysis of the execution time of the algorithm depends on characteristics of the PMR quadtree, and may change for other spatial data structures.

References

1. W. G. Aref and H. Samet. Extending a DBMS with spatial operations. In O. Günther and H. J. Schek, editors, *Advances in Spatial Databases - 2nd Symposium, SSD'91*, pages 299–318, Berlin, 1991. Springer-Verlag. (also Lecture Notes in Computer Science 525).
2. W. G. Aref and H. Samet. Uniquely reporting spatial objects: yet another operation for comparing spatial data structures. In *Proceedings of the Fifth International Symposium on Spatial Data Handling*, pages 178–189, Charleston, South Carolina, August 1992.
3. S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, Arlington, VA., January 1994.
4. D. Comer. The ubiquitous B–tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.
5. L. M. Goldschlager. Short algorithms for space-filling curves. *Software - Practice and Experience*, 11(1):99, January 1981.
6. A. Guttman. R–trees: a dynamic index structure for spatial searching. In *Proceedings of the SIGMOD Conference*, pages 47–57, Boston, June 1984.
7. E. G. Hoel and H. Samet. Efficient processing of spatial queries in line segment databases. In O. Günther and H. J. Schek, editors, *Advances in Spatial Databases - 2nd Symposium, SSD'91*, pages 237–256. Springer-Verlag, Berlin, 1991. (also Lecture Notes in Computer Science 525).
8. M. Lindenbaum and H. Samet. A probabilistic analysis of trie-based sorting of large collections of line segments. Department of Computer Science CS-TR-3455, University of Maryland, College Park, MD, April 1995.
9. R. C. Nelson and H. Samet. A population analysis for hierarchical data structures. In *Proceedings of the SIGMOD Conference*, pages 270–277, San Francisco, May 1987.
10. J. T. Robinson. The *k–d–b*–tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the SIGMOD Conference*, pages 10–18, Ann Arbor, MI, April 1981.
11. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 71–79, San Jose, CA, May 1995.
12. H. Samet. Neighbor finding techniques for images represented by quadtrees. *Computer Graphics and Image Processing*, 18(1):37–57, January 1982.
13. H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.
14. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.
15. M. Stonebraker, T. Sellis, and E. Hanson. An analysis of rule indexing implementations in data base systems. In *Proceedings of the First International Conference on Expert Database Systems*, pages 353–364, Charleston, SC, April 1986.