# Distance Browsing in Spatial Databases[1]

Gísli R. Hjaltason and Hanan Samet
Computer Science Department
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742
grh@cs.umd.edu and hjs@cs.umd.edu

**Abstract**

Two different techniques of browsing through a collection of spatial objects stored in an R-tree spatial data structure on the basis of their distances from an arbitrary spatial query object are compared. The conventional approach is one that makes use of a $k$-nearest neighbor algorithm where $k$ is known prior to the invocation of the algorithm. Thus if $m > k$ neighbors are needed, the $k$-nearest neighbor algorithm needs to be reinvoked for $m$ neighbors, thereby possibly performing some redundant computations. The second approach is incremental in the sense that having obtained the $k$ nearest neighbors, the $k + 1^{st}$ neighbor can be obtained without having to calculate the $k + 1$ nearest neighbors from scratch. The incremental approach finds use when processing complex queries where one of the conditions involves spatial proximity (e.g., the nearest city to Chicago with population greater than a million), in which case a query engine can make use of a pipelined strategy. A general incremental nearest neighbor algorithm is presented that is applicable to a large class of hierarchical spatial data structures. This algorithm is adapted to the R-tree and its performance is compared to an existing $k$-nearest neighbor algorithm for R-trees [45]. Experiments show that the incremental nearest neighbor algorithm significantly outperforms the $k$-nearest neighbor algorithm for distance browsing queries in a spatial database that uses the R-tree as a spatial index. Moreover, the incremental nearest neighbor algorithm also usually outperforms the $k$-nearest neighbor algorithm when applied to the $k$-nearest neighbor problem for the R-tree, although the improvement is not nearly as large as for distance browsing queries. In fact, we prove informally that, at any step in its execution, the incremental nearest neighbor algorithm is optimal with respect to the spatial data structure that is employed. Furthermore, based on some simplifying assumptions, we prove that in two dimensions, the number of distance computations and leaf nodes accesses made by the algorithm for finding $k$ neighbors is $O(k + \sqrt{k})$.

Keywords: distance browsing, ranking, nearest neighbors, R-trees, spatial databases, hierarchical spatial data structures.

This is a slightly modified version of an article that appeared in *ACM Transactions on Database Systems* 24, 2 (June 1999), pp. 265–318. The journal version contains two minor typographic errors. In particular, the errors were that the $\sqrt{}$ symbol was left off the second $k$ in the formulas $O(k + \sqrt{k})$ and $O(k + \sqrt{k} + \log N)$, appearing in the last line of the abstract and the second paragraph of the conclusions (Section 8).

[1]This work was supported in part by the National Science Foundation under grants IRI-9712715 and the Department of Energy under Contract DEFG0295ER25237.

## 1 Introduction

In this paper, we focus on the issue of obtaining data objects in their order of distance from a given query object (termed *ranking*). This issue is of primary interest in a spatial database although it also finds use in other database applications including multimedia indexing [36], CAD, and molecular biology [37]. The desired ranking may be full or partial (e.g., only the first $k$ objects). This problem can also be posed in a conventional database system. For example, given a table of individuals containing a weight attribute, we can ask "who has a weight closest to $w$ lbs.?", or "rank the individuals by how much their weight differs from $w$ lbs.". If no index exists on the weight attribute, then to answer the first query, a scan of all tuples must be performed. However, if an appropriate index structure is used, then more efficient methods can be employed. For example, using a B$^+$-tree, the query can be answered by a single descent to a leaf, for a cost of $O(\log n)$ for $n$ tuples. The correct answer will be found either in that leaf or an adjacent one. To rank all the individuals, the search would proceed in two directions along the leaves of the B$^+$-tree, with a constant cost for each tuple. The index can be used for any such query regardless of the reference weight $w$.

For multidimensional data, things are not so simple. Consider, for example, a set of points in two dimensions representing cities. Queries analogous to the previous ones are "what city is closest to point $p$?" and "rank the cities by their distances from point $p$". In a database context, we wish to know what kind of index structures will aid in processing these queries. For a fixed reference point $p$ and distance metric, we might build a one-dimensional index on the distances of the cities from the point $p$. This would provide an efficient execution time for this particular point (i.e., for $p$), but for any other point or distance metric it would be useless. Thus we have to rebuild the index, which is a costly process if we need to do it for each query. Contrast this to the one-dimensional case, where there is generally only one choice of metric. Furthermore, for a given reference point, any other point can have only two positions in relation to it, larger or smaller. It is not possible to define such a simple relationship in the multidimensional case.

As another example, suppose we want to find the nearest city to Chicago that has more than a million inhabitants. There are several ways to proceed. An intuitive solution is to guess some area range around Chicago and check the populations of the cities in the range. If we find a city with the requisite population, we must make sure that there are no other cities that are closer and that meet the population condition. This approach is rather inefficient as we have to guess the size of the area to be searched. The problem with guessing is that we may choose too small a region or too large a region. If the size is too small, the area may not contain any cities satisfying the population criterion, in which case we need to expand the region being searched. If the size is too large, we may be examining many cities needlessly.

A radical solution is to sort all the cities by their distances from Chicago. This is not very practical as we need to re-sort them each time we pose a similar query with respect to another city. Moreover, sorting requires a considerable amount of extra work, especially when usually all that is needed to obtain the desired result is to inspect the first few nearest neighbors.

A less radical solution is to retrieve the closest $k$ cities and determine if any of them satisfy the population criterion. The problem here lies in determining the value of $k$. As in the area range solution, we may choose too small or too large a value of $k$. If $k$ is too small, failure to find a city satisfying the population criterion means that we have to restart the search with a value larger than $k$, say $m$. The drawback of this solution is that such a search forces us to expend work in finding the $k$ nearest neighbors (which we already did once before) as part of the cost of finding the $m > k$ nearest neighbors. On the other hand, if $k$ is too large, we waste work in calculating neighbors whose populations we will never check.

A logical way to overcome the drawbacks of the second and third solutions is to obtain the neighbors incrementally (i.e., one by one) as they are needed. In essence, what we are doing is browsing through the database on the basis of distance and we shall use the term *distance browsing* to describe this operation. The result is an incremental ranking of the cities by distance where we cease the search as soon as the secondary population condition is satisfied. The idea is that we want only a small but unknown number of neighbors. The incremental solution finds application in a much more general setting than our specialized query example. In particular, this includes queries that require the application of the "nearest" predicate to a subset $s$ of the attributes of a relation (or object class) $r$. This class of queries is part of a more restricted, but very common, class that imposes an additional condition $c$ usually involving attributes other than $s$. This means that the "nearest" condition serves as a primary condition, while condition $c$ serves as a secondary condition. Using an incremental solution enables such a query to be processed in a pipelined fashion.

Of course, in the worst case, we will have to examine all (or most) of the neighbors even when using an incremental approach. This may occur if few objects satisfy the secondary condition (e.g., if none of the cities have the requisite population). In this case, it may actually be better to first select on the basis of the secondary condition (the population criterion in our example) before considering the "spatially nearest" condition, especially if an index exists that can be used to compute the secondary condition. Using a $k$-nearest neighbor algorithm may also be preferable, provided it is more efficient than the incremental algorithm for large values of $k$. It makes sense to choose this solution only if we know in advance how many neighbors are needed (i.e., the value of $k$), but this value can be estimated based on the selectivity of the secondary condition. These issues demonstrate the need for a query engine to make estimates using selectivity factors (e.g., [3, 40, 49]) involving the numbers of values that are expected to satisfy various parts of the query and the computational costs of the applicable algorithms.

In this paper we compare the incremental and $k$-nearest neighbor approaches for browsing through a collection of spatial objects stored in an R-tree spatial data structure on the basis of their distances from an arbitrary spatial query object. In the process we present a general incremental nearest neighbor algorithm that is applicable to a large class of hierarchical spatial data structures, and show how to adapt this algorithm to the R-tree. Its performance is compared to an existing $k$-nearest neighbor algorithm for R-trees [45]. In addition, we demonstrate that the $k$-nearest neighbor algorithm of [45] can be transformed into a special case of our R-tree adaptation of the general incremental nearest neighbor algorithm. The transformation process also reveals that the R-tree incremental nearest neighbor algorithm achieves more pruning than the R-tree $k$-nearest neighbor algorithm. Moreover, our R-tree adaptation leads to a considerably more efficient (and conceptually different) algorithm. This is because the presence of object bounding rectangles in the tree enables their use as pruning devices to reduce disk I/O for accessing the spatial descriptions of objects (stored external to the tree). Experiments show that the incremental nearest neighbor algorithm significantly outperforms the $k$-nearest neighbor algorithm for distance browsing queries in a spatial database that uses the R-tree as a spatial index. Moreover, the incremental nearest neighbor algorithm also usually outperforms the $k$-nearest neighbor algorithm when applied to the $k$-nearest neighbor problem for the R-tree, although the improvement is not nearly as large as for distance browsing queries.

The rest of this paper is organized as follows. Section 2 discusses algorithms related to nearest neighbor queries. Section 3 reviews the structure of R-trees. Section 4 describes the incremental nearest neighbor algorithm as well as its adaptation to the R-tree. Section 5 introduces the $k$-nearest neighbor algorithm. Section 6 presents the results of an empirical study comparing the incremental nearest neighbor algorithm with the $k$-nearest neighbor algorithm. Section 7 discusses issues that arise in high-dimensional spaces, while conclusions are drawn in Section 8.

## 2 Related Work

Numerous algorithms exist for answering nearest neighbor and $k$-nearest neighbor queries. This is motivated by the importance of these queries in fields including geographical information systems (GIS), pattern recognition, document retrieval, and learning theory. Almost all of these algorithms, many of them coming from the field of computational geometry, are for points in a $d$-dimensional vector space [12, 16, 21, 22, 33, 45, 51], but some allow for arbitrary spatial objects [26, 30], although most are still limited to a point as the query object. In many applications, a rough answer suffices, so that algorithms have been developed that return an approximate result [4, 10, 54], thereby saving time in computing it. Many of the above algorithms require specialized search structures [4, 10, 16, 22, 33], but some employ commonly used spatial data structures. For example, algorithms exist for the k-d tree [12, 21, 41, 51], quadtree-related structures [29, 30], the R-tree [45, 54], the LSD-tree [26] and others. In addition, many of the algorithms can be applied to other spatial data structures.

To our knowledge, only three incremental solutions to the nearest neighbor problem exist in the literature [12, 26, 29]. All these algorithms employ priority queues (see Section 4). The algorithm of [12] was developed for the k-d tree [7]. It is considerably different from the other two algorithms in that the algorithm of [12] stores only the data objects in the priority queue, and uses a stack to keep track of the subtrees of the spatial data structure which have yet to be completely processed. This makes it necessary to use an elaborate mechanism to avoid processing the contents of a node more than once. The algorithm of [26] was developed for the LSD-tree [28]. It is very similar to our method (presented in [29]) and was published at about the same time. The principal difference between [26] and our method is that the LSD-tree algorithm uses two priority queues, one for the data objects and another for the nodes of the spatial data structure. This makes the algorithm somewhat more complicated than ours, while the use of two priority queues does not offer any performance benefits according to our experiments. Our algorithm [29] was initially developed for the PMR quadtree [42] although its presentation was general. In this paper we expand considerably on our initial solution by showing how it can be adapted to the R-tree [24] as well as comparing it with a solution that makes use of an existing $k$-nearest neighbor algorithm [45]. In addition, we show how this $k$-nearest neighbor algorithm [45] can be transformed into a special case of our R-tree adaptation of the general incremental nearest neighbor algorithm. A byproduct of the transformation process is that the $k$-nearest neighbor algorithm has been simplified considerably.

The term *distance scan* [5] has also been used for what we term distance browsing. Becker and Güting [5] introduce the concept of a distance scan and motivate its use. This is done along similar lines to those of Section 1, i.e., in the context of finding the closest object to a query point where additional conditions may be imposed on the object. In addition, that paper provides optimization rules for mapping a "closest" operator into a "distance scan" operation in an example GIS query language.

All the algorithms mentioned thus far assume that the objects exist in a $d$-dimensional Euclidean space, so that distances are defined between every two objects in a data set as well as between an object and any point in the space. Another class of nearest neighbor algorithms operates on more general objects, in what is commonly called the metric space model. The only restriction on the objects is that they reside in some metric space, i.e., a distance metric is defined between any two objects. However, in this general case, it is not possible to produce new objects in the metric space, e.g., to aggregate or divide two objects (in a Euclidean space, bounding rectangles are often used for this purpose). Various methods exist for indexing objects in the metric space model as well as for computing proximity queries [11, 13, 14, 52, 53]. These methods can only make use of the properties of distance metrics (nonnegativity, symmetry, and the triangle inequality), and operate without any knowledge of how objects are represented or how the distances between objects are computed. Such a general approach is usually slower than methods based on spatial properties of objects, but must be used for objects for which such properties do not exist (e.g., images, chemical data, time series, etc.). This approach has also been advocated for high-dimensional vector spaces. It may often be possible to map general objects into geometric space, thereby reaping the benefit of more efficient search methods. Most such mapping approaches are domain-specific [25, 36], but general approaches have also been proposed [18].

## 3 R-trees

The R-tree (e.g., Figure 1) [24] is an object hierarchy in the form of a balanced structure inspired by the $B^+$-tree [15]. Each R-tree node contains an array of (*key*, *pointer*) entries where *key* is a hyper-rectangle that minimally bounds the data objects in the subtree pointed at by *pointer*. In an R-tree leaf node, the *pointer* is an object identifier (e.g., a tuple ID in a relational system), while in a nonleaf node it is a pointer to a child node on the next lower level. The maximum number of entries in each node is termed its *node capacity* or *fan-out* and may be different for leaf and nonleaf nodes. The node capacity is usually chosen such that a node fills up one disk page (or a small number of them). It should be clear that the R-tree can be used to index a space of arbitrary dimension and arbitrary spatial objects rather than just points.

As described above, an R-tree leaf node contains a minimal bounding rectangle and an object identifier for each object in the node, i.e., the geometric descriptions of the objects are stored external to the R-tree itself. Another possibility is to store the actual object, or its geometric description, in the leaf instead of its bounding rectangle. This is usually useful only if the object representation is relatively small (e.g., similar in size to a bounding rectangle) and is fixed in length. If all the data about the object (i.e., all its relevant attributes) are stored in the leaf nodes, the object identifiers need not be stored. The disadvantage of this approach is that objects will not have fixed addresses, as some objects must be moved each time an R-tree node is split.

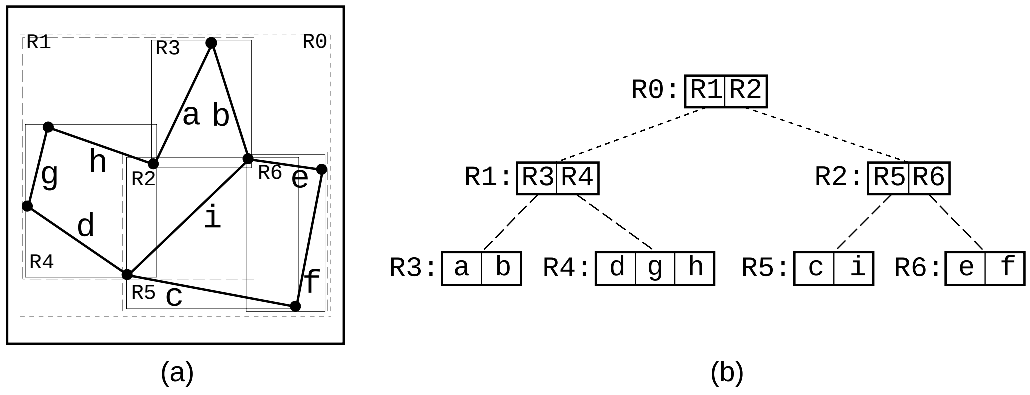

Figure 1: An R-tree index for a set of nine line segments. (a) Spatial rendering of the line segments and bounding rectangles; (b) a tree access structure for (a). The bounding rectangles for the individual line segments are omitted from (a) in the interest of clarity.

Several variations of R-trees have been devised, differing in the way nodes are split or combined during insertion or deletion. In our experiments we make use of a variant called the $R^*$-tree [6]. It differs from the conventional R-tree in employing more sophisticated insertion and node-splitting algorithms that attempt to minimize a combination of overlap between bounding rectangles and their total area. In addition, when R-tree node $p$ overflows, instead of immediately splitting $p$, the $R^*$-tree insertion algorithm first tries to see if some of the entries in $p$ could possibly fit better in another node. This is achieved by reinserting a fixed fraction of the entries in $p$. This increases the construction time for the index, but usually results in less node overlap and therefore in improved query response time.

## 4 Incremental Nearest Neighbor Algorithm

Most algorithms that traverse tree structures in a top-down manner use some form of depth-first or breadth-first tree traversal. Finding a leaf node containing a query object $q$ in a spatial index can be done in a depth-first manner by recursively descending the tree structure. With this method, the recursion stack keeps track of what nodes have yet to be visited. Having reached a leaf, we need to be able to extend this technique to find the nearest object, as the leaf may not actually contain the nearest neighbor. The problem here is that we have to unwind the recursion to find the nearest object. Moreover, if we want to find the second nearest object, the solution becomes even tougher. With breadth-first traversal, the nodes of the tree are visited level by level, and a queue is used to keep track of nodes that have yet to be visited. However, with this technique, a lot of work has to be done before reaching a leaf node containing $q$. To resolve the problems with depth-first and breadth-first traversal, the incremental nearest neighbor algorithm employs what may be termed best-first traversal. When deciding what node to traverse next, it picks the node with the least distance in the set of *all* nodes that have yet to be visited. This means that instead of using a stack or a plain queue to keep track of the nodes to be visited, we use a priority queue where the distance from the query object is used as a key. The key feature of our solution is that the objects as well as the nodes are stored in the priority queue.

This section is organized as follows: In Section 4.1 we specify what conditions must hold for our incremental nearest neighbor algorithm to be applicable (e.g., conditions on the index, spatial object types, distance functions, etc.). In Section 4.2 we present the general incremental nearest neighbor algorithm in detail. In Section 4.3 we discuss ways to exploit the particular nature of the R-tree spatial index, while in Section 4.4 we give an example of the execution of the algorithm on a simple R-tree structure. Several variants of the algorithm are described in Section 4.5. In Section 4.6 we present some analytical results for the algorithm, while in Section 4.7 we prove its correctness. Finally, in Section 4.8 we show how to deal with a large priority queue.

### 4.1 Introduction

Our incremental nearest neighbor algorithm can be applied to virtually any hierarchical spatial data structure. In fact, it is generally applicable to any data structure based on hierarchical containment/partitioning (e.g., see [1]). In our description, we will assume a tree structure (although our method is applicable to more general structures), where each tree node represents some regions of space and where objects (or pointers to them in an external table) are stored in the leaf nodes whose regions intersect the objects. In the remainder of this section, we do not make a distinction between a node and the region that it represents; the meaning should be clear from the context. A basic requirement for the method to be applicable is that the region covered by a node must be completely contained within the region(s) of the parent node(s)[1]. Examples of structures that satisfy this requirement include quadtrees [47], R-trees [24], R$^+$-trees [50], LSD-trees [28], and k-d-B-trees [44]. In all these examples, the node region is rectangular, but this is not a requirement. Our algorithm handles the possibility of an object being represented in more than one leaf node, as in the PMR quadtree [42] and R$^+$-tree [50]. Although we assume in our exposition that each node has only one parent and that only leaf nodes store objects, the algorithm could easily be adapted to handle other cases (such as the hB-tree [39] and the cell tree with oversize shelves [23]).

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 consistent distance functions $d_o$ and $d_n$ be used for calculating the distance from the query object $q$ to data objects and to nodes. This is to ensure that each object is encountered in at least one node that is no farther from the query object than the object itself; otherwise, the

[1]For structures in which each node can have more than one parent (e.g., the hB-tree [39] or Partition Fieldtree [19]) the node region must be fully contained in the union of the regions of the parent nodes.

strictly nondecreasing distances of elements retrieved from the queue cannot be guaranteed. Consistency can be defined formally as follows: (In the definition, we do not make any assumptions about the nature of the index hierarchy.)

> **Definition** Let $d$ be the combination of functions $d_o$ and $d_n$, and let $e \sqsubseteq N$ denote the fact that item $e$ is contained in exactly the set of nodes $N$ (i.e., if $e$ is an object, $N$ is the set of leaf nodes referencing the object, and if $e$ is a node, $N$ is its set of parent nodes[2]). The functions $d_o$ and $d_n$ are *consistent* iff for any query object $q$ and any object or node $e$ in the hierarchical data structure there exists $n$ in $N$, where $e \sqsubseteq N$, such that $d(q, n) \leq d(q, e)$.

This definition is strictly tied to the hierarchy defined by the data structure. However, since this hierarchy is influenced by properties of the node regions and data objects, we can usually recast the definition in terms of these properties. For example, in spatial data structures the containment of objects in leaf nodes and child nodes in parent nodes is based on spatial containment; thus the $\sqsubseteq$ in the definition also denotes spatial containment. In other words, $e \sqsubseteq N$ means that the union of the node regions for the nodes in $N$ completely encloses the region covered by the object or node $e$. Informally, our definition of consistency means that if $p$ is the point in $e$ (or, more accurately, in the region that corresponds to it) closest to $q$, then $p$ must also be contained in the region covered by some node in $N$. Note that since we assume spatial indexes that form a tree hierarchy (i.e., each nonroot node has exactly one parent), in the case of nodes the definition above simplifies to the following condition: if $n'$ is a child node of node $n$, then $d_n(q, n) \leq d_n(q, n')$.

An easy way to ensure consistency is to base both functions on the same metric $d_p(p_1, p_2)$ for points; common choices of metrics include the Euclidean, Manhattan and Chessboard metrics. We then define $d(q, e) := \min_{p_1 \in q, p_2 \in e} d_p(p_1, p_2)$, where $e$ is either a spatial object or a node region. It is important to note that this is not the only way to define consistent distance functions. When $d$ is defined based on a metric $d_p$, its consistency is guaranteed by the properties of $d_p$, specifically, nonnegativity and the triangle inequality. The nonnegativity property states, among other things, that $d_p(p, p) = 0$, and the triangle inequality states that $d_p(p_1, p_3) \leq d_p(p_1, p_2) + d_p(p_2, p_3)$. Since $e$ is spatially contained in $N$, $e$ and $N$ have points in common, so their distance is zero. Thus, according to the triangle inequality, $d(q, e) \leq d(q, N) + d(N, e) = d(q, N)$, using a broad definition of $d$ (to allow $d(N, e)$, which equals 0). Note that if the distance functions are defined in this way, the distance from a query object to a node that intersects it is zero (i.e., it is not equal to the distance to the boundary of the node region).

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

### 4.2 Algorithm Description

We first consider a regular recursive top-down traversal of the index to locate a leaf node containing the query object. Note that there may be more than one such node. The traversal is initiated with the root node of the spatial index (i.e., the node spanning the whole index space) as the second argument.

```
FINDLEAF(QueryObject, Node)
1 if QueryObject is in node Node then
2    if Node is a leaf node then
3       Report leaf node Node
4    else
```

[2]In most spatial data structures, each node has only one parent node; the hB-tree is an exception.

```
5         for each Child of node Node do
6             FINDLEAF(QueryObject, Child)
7         enddo
8     endif
9 endif
```

The first task is to extend the algorithm to find the object nearest to the query object. In particular, once a leaf node containing *QueryObject* has been found in line 3, we could start by examining the objects contained in that node. However, the object closest to the query object might reside in another node. Finding that node may in fact require unwinding the recursion to the top and descending again deeper into the tree. Furthermore, once that node has been found, it does not 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 nodes, objects are also put on the queue as leaf nodes are processed. The key used to order the elements on the queue is distance from the query object. In order to distinguish between two elements at equal distances from the query object, we adopt the convention that nodes are ordered before objects, while objects are ordered according to some arbitrary (but unique) rule. This secondary ordering makes it possible to avoid reporting an object more than once, which is necessary when using a disjoint decomposition, e.g., a PMR quadtree [42] or an R$^+$-tree [50], in which nonpoint objects may be associated with more than one node.

A node is not examined until it reaches the head of the queue. At this time, all nodes and objects closer to the query object have been examined. Initially, the node spanning the whole index space is the sole element in the priority queue. At subsequent steps, the element at the head of the queue (i.e., the closest element not yet examined) is retrieved, and this is repeated until the queue has been emptied. Informally, we can visualize the progress of the algorithm for a query object $q$ as follows, when $q$ is a point (see Figure 2). We start by locating the leaf node(s) containing $q$. Next, imagine a circle centered at $q$ being expanded from a starting radius of 0; we call this circle the *search region*. Each time the circle hits the boundary of a node region, the contents of that node are put on the queue, and each time the circle hits an object, we have found the object next nearest to $q$. Note that when the circle hits a node or an object, we are guaranteed that the node or object is already in the priority queue, since the node that contains it must already have been hit (this is guaranteed by the consistency condition).

Figure 3 presents the algorithm. Lines 1–2 initialize the queue. Notice that it is not really necessary to provide the correct distance when enqueueing the root node, since it will always be dequeued first. In line 9, the next closest object is reported. At that point, some other routine (such as a query engine) can take control, possibly resuming the algorithm at a later time to get the next closest object, or alternately terminating it if no more objects are desired.

Recall that for some types of spatial indexes, a spatial object may span several nodes. In such a case, the algorithm must guard against objects being reported more than once [2]. The test (i.e., the **if** statement) in line 12 ensures that objects that have already been reported are not put on the queue again. (Note that this test is not needed in the case when *Element* is a nonleaf node, as it holds implicitly by the assumption that child nodes are fully contained in their parent nodes.) For this to work properly, nodes must be retrieved from the queue before spatial objects at the same distance. Otherwise, an object may be retrieved from the queue before a node $n$ containing it that is at the same distance from the query object (this means that the object was contained in another node that has already been dequeued). When the object is then encountered again in node $n$, there is no way of knowing that it has already been reported. The loop in lines 6–8 eliminates duplicate instances of an object from the queue. By inducing an ordering on objects 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. We explicitly check for duplicates in this manner because for many

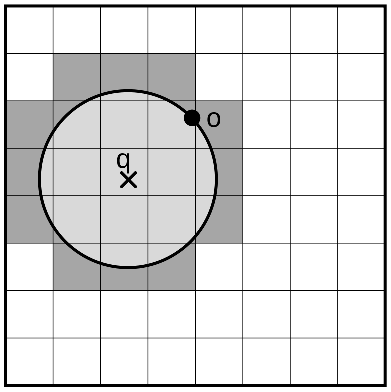

Figure 2: The circle around query object $q$ depicts the search region after reporting $o$ as next nearest object. For simplicity, the leaf nodes are represented by a grid; in most spatial indexes, the shapes of the leaf nodes are more irregular than in a grid. Only the shaded leaf nodes are accessed by the incremental nearest neighbor algorithm. The region with darker shading is where we find the objects in the priority queue.

```
IncNearest(QueryObject, SpatialIndex)
 1 Queue ← NewPriorityQueue()
 2 Enqueue(Queue, SpatialIndex.RootNode, 0)
 3 while not IsEmpty(Queue) do
 4    Element ← Dequeue(Queue)
 5    if Element is a spatial object then
 6       while Element = First(Queue) do
 7          DeleteFirst(Queue)
 8       enddo
 9       Report Element
10    elseif Element is a leaf node then
11       for each Object in leaf node Element do
12          if Dist(QueryObject, Object) $\geq$ Dist(QueryObject, Element) then
13             Enqueue(Queue, Object, Dist(QueryObject, Object))
14          endif
15       enddo
16    else /* Element is a nonleaf node */
17       for each Child node of node Element in SpatialIndex do
18          Enqueue(Queue, Child, Dist(QueryObject, Child))
19       enddo
20    endif
21 enddo
```

Figure 3: Incremental nearest neighbor algorithm.

priority queue implementations (e.g., binary heap), it is not efficient to detect duplicates among the queue elements, as these implementations only maintain a partial among the elements. A possible alternative is to use a priority queue implementation that maintains a total order among all the queue elements (e.g., a balanced binary tree) and thus is able to detect duplicates efficiently.

### 4.3 Adapting to R-trees

In this section, we demonstrate how to adapt the general incremental algorithm presented above to R-trees by exploiting some of the unique properties of R-trees. If the spatial objects are stored external to the R-tree, such that leaf nodes contain only bounding rectangles for objects, then this adaptation leads to a considerably more efficient (and conceptually different) incremental algorithm. This enables the bounding rectangles to be used as pruning devices, thereby reducing the disk I/O needed to access the spatial descriptions of the objects. In addition, R-trees store each object just once, making it unnecessary to worry about reporting an object more than once. This also removes the need to enforce the secondary ordering on the priority queue used by the general algorithm (see Section 4.2).

The inputs to the R-tree incremental nearest neighbor algorithm are a query object $q$ and an R-tree $R$ containing a set of spatial data objects. As with the general incremental nearest neighbor algorithm, the data objects as well as the query object may be of any dimension and of arbitrary type (e.g., points, rectangles, polygons, etc.), as long as consistent distance functions are used for calculating the distance from $q$ to data objects and bounding rectangles. In the case of an R-tree, this means that if $e$ is a data object or a rectangle completely contained in rectangle $r$, then $d(q, r) \leq d(q, e)$.

The general algorithm can be used virtually unchanged if object geometry is stored in the R-tree leaf nodes, the only changes being the ones already described. If the spatial objects are stored external to the R-tree, the primary difference from the general algorithm is in the use of the bounding rectangles stored in the leaf nodes. To exploit that information, a third type of queue element is introduced: object bounding rectangle. The distance of an object bounding rectangle is never greater than the distance of the object, provided the distance functions used are consistent. Informally, the modifications to the algorithm are as follows: When an R-tree leaf is being processed in the main loop of the algorithm, instead of computing the real distances of the objects, the distances of their bounding boxes are computed and inserted into the queue. Only when an object's bounding box is retrieved from the queue is the actual distance computed. If the object is closer to the query object than the next element on the priority queue, it can be reported as the next nearest neighbor. Otherwise, the object is inserted into the queue with its real distance.

Figure 4 shows our algorithm. In lines 1–2, the queue is initialized. In line 9, the next closest object is reported. In line 7, an object $p$ is enqueued with its real distance as the key after it has been determined that there are elements on the queue with a key less than the real distance from $p$ to the query object $q$. If there are no such elements, $p$ is reported as the next nearest object. Line 13 enqueues an object bounding rectangle; brackets around *Object* signal that it is not the object itself but instead the bounding rectangle along with a pointer to the corresponding object. The general incremental nearest neighbor algorithm had an extra test at this point to guard against reporting duplicates, but that is not needed here.

The R-tree variant given above can be used for any spatial data structure method that separates the storage of bounding rectangles and the actual geometric descriptions of objects. For complex objects, for example polygons, one can even conceive of several levels of refinement, e.g., with the use of orthogonal polygons [17].

### 4.4 Example

As an example, suppose that we want to find the three nearest neighbors to query point q in the R-tree given in Figure 1, where the spatial objects are line segments which are stored external to the R-tree. Below, we show the steps of the algorithm and the contents of the priority queue. The algorithm must compute the distances between q and the line segments and bounding rectangles. These distances are given in Table 1 (*BR* means bounding rectangle). They are based on an arbitrary coordinate system and are approximate. When depicting

```
INCNEAREST(QueryObject, R-tree)
 1 Queue ← NEWPRIORITYQUEUE()
 2 ENQUEUE(Queue, R-tree.RootNode, 0)
 3 while not ISEMPTY(Queue) do
 4   Element ← DEQUEUE(Queue)
 5   if Element is an object or its bounding rectangle then
 6     if Element is the bounding rectangle of Object and not ISEMPTY(Queue)
         and DIST(QueryObject, Object) > FIRST(Queue).Key then
 7       ENQUEUE(Queue, Object, DIST(QueryObject, Object))
 8     else
 9       Report Element (or if bounding rectangle, the associated object)
         as the next nearest object
10     endif
11   elseif Element is a leaf node then
12     for each entry (Object, Rect) in leaf node Element do
13       ENQUEUE(Queue, [Object], DIST(QueryObject, Rect))
14     enddo
15   else /* Element is a nonleaf node */
16     for each entry (Node, Rect) in node Element do
17       ENQUEUE(Queue, Node, DIST(QueryObject, Rect))
18     enddo
19   endif
20 enddo
```

Figure 4: Incremental nearest neighbor algorithm for an R-tree where spatial objects are stored external to the R-tree.

| Seg. | Dist. | BR Dist. |
|---|---|---|
| a | 17 | 13 |
| b | 48 | 27 |
| c | 57 | 53 |
| d | 59 | 30 |
| e | 48 | 45 |
| f | 86 | 74 |
| g | 81 | 74 |
| h | 17 | 17 |
| i | 21 | 0 |

| BR | Dist. |
|---|---|
| R0 | 0 |
| R1 | 0 |
| R2 | 0 |
| R3 | 13 |
| R4 | 11 |
| R5 | 0 |
| R6 | 44 |

Table 1: Distances of line segments and bounding rectangles from the query point $q$ in the R-tree of Figure 1.

the contents of the priority queue, the line segments and bounding rectangles are listed with their distances, in increasing order of distance, with ties broken using alphabetical ordering. Bounding rectangles of objects are denoted by the corresponding object names embedded in brackets (e.g., `[h]`). The algorithm starts by enqueueing `R0`, after which it executes the following steps:

1. Dequeue `R0`, enqueue `R1` and `R2`. Queue: $\{$(`R1,0`), (`R2,0`)$\}$.

2. Dequeue R1, enqueue R3 and R4. Queue: {(R2,0), (R4,11), (R3,13)}.

3. Dequeue R2, enqueue R5 and R6. Queue: {(R5,0), (R4,11), (R3,13), (R6,44)}.

4. Dequeue R5, enqueue [c] and [i] (i.e., the bounding rectangles of c and i). Queue: {([i],0), (R4,11), (R3,13), (R6,44), ([c],53)}.

5. Dequeue [i]. The distance of i is 21, which is larger than the distance of R4, so enqueue i. Queue: {(R4,11), (R3,13), (i,21), (R6,44), ([c],53)}.

6. Dequeue R4, and enqueue [d], [g], and [h]. Queue: {(R3,13), ([h],17), (i,21), ([d],30), (R6,44), ([c],53), ([g],74)}.

7. Dequeue R3, enqueue [a] and [b]. Queue: {([a],13), ([h],17), (i,21), ([b],27), ([d],30), (R6,44), ([c],53), ([g],74)}.

8. Dequeue [a]. The distance of a is 17, which is not larger than the distance of [h], so a is reported as nearest neighbor. Queue: {([h],17), (i,21), ([b],27), ([d],30), (R6,44), ([c],53), ([g],74)}.

9. Dequeue [h]. The distance of h is 17, which is not larger than the distance of i, so h is reported as second nearest neighbor. Queue: {(i,21), ([b],27), ([d],30), (R6,44), ([c],53), ([g],74)}.

10. Dequeue i and report it as third nearest neighbor.

Observe that node R6 is left on the priority queue at the end of the execution. This corresponds to the $k$-nearest neighbor algorithm not being invoked on that node (see Section 5.2). For larger examples, the incremental algorithm will generally achieve more pruning than the $k$-nearest neighbor algorithm, but never less.

Also note that the second and third nearest neighbors were obtained with very little additional work once the nearest neighbor was found. This is often the case with the incremental nearest neighbor algorithm regardless of the underlying spatial index. In other words, once the nearest neighbor has been found, the next few nearest neighbors can be retrieved with virtually no additional work.

## 4.5 Variants

With relatively minor modifications, the incremental nearest neighbor algorithm can be used to find the farthest object from the query object. In this case, the queue elements are sorted in decreasing order of their distances. This is not enough, though, since objects or nodes contained in a node $n$ are generally at larger distances from the query object $q$ than $n$ is. This means that elements would be enqueued with larger keys than the node they are contained in, which breaks the condition that elements are dequeued in decreasing order of distance. Instead, the key used for a node $n$ on the queue must be an upper bound on the distance from $q$ to an object in the subtree at $n$, e.g., $d_{\max}(q, n) = \max_{p \in n} d_p(q, p)$. The function implementing $d_{\max}$ must satisfy a consistency condition similar to that defined above for $d_n$; the only difference is that for $d_{\max}$, we replace $\leq$ in the condition by $\geq$.

Another extension to the algorithm is to allow a minimum and a maximum to be imposed on the distances of objects that are reported. However, in order to effectively utilize a minimum, the distance function $d_{\max}$ defined above is needed. Then, a node $n$ is put on the queue only if $d_{\max}(q, n)$ is greater or equal to the minimum desired distance. Notice that in this case, the algorithm performs a spatial selection operation in addition to the ranking.

Figure 5 gives a version of the algorithm with these two extensions added. The arguments *Min* and *Max* specify the minimum and maximum desired distance, and *DoFarthest* is a Boolean variable that is true when the farthest object is desired. In the latter case, negative distances are used as keys for the priority queue, so that elements get sorted in decreasing order of distance. The condition $KeySign(d - e) \geq 0$ in line 19 of Figure 5 encompasses the conditions $d \geq e$ and $d \leq e$, for when *DoFarthest* is false and true, respectively. In line 16, the key of the leaf node is assigned to $e$. This is the minimum or maximum distance of the node, depending on the value of *DoFarthest*. The reason for multiplying the key by *KeySign* in line 16 is to cancel out the effect of multiplying the value of $d$ by *KeySign* in line 33, which makes it negative when looking for the farthest objects.

A powerful way of extending the incremental nearest neighbor algorithm is to combine it with other spatial queries and/or restrictions on the objects or nodes. As an example, the algorithm can be combined with a range query by checking each object and node against the range prior to inserting it onto the priority queue, and rejecting those that do not fall in the range. Many such combined queries can be obtained by manipulating the distance functions so that they return special values for objects and nodes that should be rejected.

The incremental nearest neighbor algorithm can clearly be used to solve the traditional $k$-nearest neighbor problem, i.e., given $k$ and a query object $q$ find the $k$ nearest neighbors of $q$. This is done by simply retrieving $k$ neighbors with the algorithm and terminating once they have all been determined.

## 4.6 Analysis

Performing a comprehensive theoretical analysis of the incremental nearest neighbor algorithm is complicated, especially for high-dimensional spaces. Prior work in this area is limited to the case where both the data objects and the query object are points [8, 26]. A number of simplifying assumptions were made, e.g., that the data objects are uniformly distributed in the data space. In this section, we discuss some of the issues involved, and sketch a rudimentary analysis for two-dimensional points, based on the one in [26].

We wish to analyze the situation after finding the $k$ nearest neighbors. Let $o$ be the $k^{th}$ nearest neighbor of the query object $q$, and let $r$ be the distance of $o$ from $q$. The region within distance $r$ from $q$ is called the search region. Since we assume that $q$ is a point, the search region is a circle (or a hypersphere in higher dimensions) with radius $r$. Figure 2 depicts this scenario. Observe that all objects inside the search region have already been reported by the algorithm (as the next nearest object), while all nodes intersecting the search region have been examined and their contents put on the priority queue. A further insight can be obtained about the contents of the priority queue by noting that if $n$ is a node that is completely inside the search region, all nodes and objects in the subtree rooted at $n$ have already been taken off the queue. Thus all elements on the priority queue are contained in nodes intersecting the boundary of the search region (the dark shaded region in Figure 2).

Before proceeding any further, we point out that the algorithm does not access any nodes or objects that lie entirely outside the search region (i.e., that are farther from $q$ than $o$ is). This follows directly from the queue order and the consistency conditions. In particular, the elements are retrieved from the priority queue in order of distance, and the consistency conditions guarantee that we never insert elements into the queue with smaller distances than that of the element last dequeued. Conversely, any algorithm that uses a spatial index must visit all the nodes that intersect the search region; otherwise, it may miss some objects that are closer to the query object than $o$. Thus we have established that the algorithm visits the minimal number of nodes necessary for finding the $k^{th}$ nearest neighbor. This can be characterized by saying that the algorithm is optimal with respect to the structure of the spatial index. However, this does not mean that the algorithm is optimal with respect to the nearest neighbor problem; how close the algorithm comes to being optimal in this respect depends on the spatial index.

```
INCNEAREST(QueryObject, SpatialIndex, Min, Max, DoFarthest)
Queue ← NEWPRIORITYQUEUE()
ENQUEUE(Queue, SpatialIndex.RootNode, 0)
if DoFarthest then
  KeySign ← $-1$
else
  KeySign ← $1$
endif
while not ISEMPTY(Queue) do
  Element ← DEQUEUE(Queue)
  if Element is a spatial object then
    while Element = FIRST(Queue) do
      DELETEFIRST(Queue)
    enddo
    Report Element
  elseif Element is a leaf node then
    $e$ ← Element.Key*KeySign
    for each Object in leaf node Element do
      $d$ ← DIST(QueryObject, Object)
      if $d \geq$ Min and $d \leq$ Max and KeySign$*(d - e) \geq 0$ then
        ENQUEUE(Queue, Object, $KeySign * d$)
      endif
    enddo
  else /* Element is a nonleaf node */
    for each Child node of node Element in SpatialIndex do
      $d_{\text{min}}$ ← MINDIST(QueryObject,Child)
      $d_{\text{max}}$ ← MAXDIST(QueryObject,Child)
      if $d_{\text{max}} \geq$ Min and $d_{\text{min}} \leq$ Max then
        if DoFarthest then
          $d \leftarrow d_{\text{max}}$
        else
          $d \leftarrow d_{\text{min}}$
        endif
        ENQUEUE(Queue, Child, $KeySign * d$)
      endif
    enddo
  endif
enddo
```

Figure 5: Enhanced incremental nearest neighbor algorithm

Generally, two steps are needed to derive performance measures for the incremental nearest neighbor algorithm. First, the expected area of the search region is determined. Then, based on the expected area of the search region and an assumed distribution of the locations and sizes of the leaf nodes, we can derive such measures as the expected number of leaf nodes accessed by the algorithm (i.e., intersected by the search region) or the expected number of objects in the priority queue. Henrich [26] describes one such approach, which uses a number of simplifying assumptions. In particular, it assumes $N$ uniformly distributed data points in the two-dimensional interval $[0,1] \times [0,1]$, the leaf nodes are assumed to form a grid at the lowest level of the spatial index with average occupancy of $c$ points, and the search region is assumed to be completely contained in the data space. Since we assume uniformly distributed points, the expected area of the search region is $k/N$ and the expected area of the leaf node regions is $c/N$. The area of a circle of radius $r$ is $\pi r^2$, so for the search region we have $\pi r^2 = k/N$, which means that its radius is $r = \sqrt{\frac{k}{\pi N}}$. The leaf node regions are squares, so their side length is $s = \sqrt{c/N}$. Henrich [26] points out that the number of leaf node regions intersected by the boundary of the search region is the same as that intersected by the boundary of its circumscribed square. Each of the four sides of the circumscribed square intersects $\lfloor 2r/s \rfloor \leq 2r/s$ leaf node regions. Since each two adjacent sides intersect the same leaf node region at a corner of the square, the expected number of leaf node regions intersected by the search region is bounded by

$$4(2r/s - 1) = 4\left(\frac{2\sqrt{k/(\pi N)}}{\sqrt{c/N}} - 1\right) = 4\left(2\sqrt{\frac{k}{\pi c}} - 1\right).$$

It is reasonable to assume that, on the average, half of the $c$ points in these leaf nodes are inside the search region, while half are outside. Thus the expected number of points remaining in the priority queue (the points in the dark shaded region in Figure 2) is at most

$$\frac{c}{2}4\left(2\sqrt{\frac{k}{\pi c}} - 1\right) = 2c\left(2\sqrt{\frac{k}{\pi c}} - 1\right) = \frac{4}{\sqrt{\pi}}\sqrt{ck} - 2c \approx 2.26\sqrt{ck} - 2c.$$

The number of points inside the search region (the light shaded region in Figure 2) is $k$. Thus the expected number of points in leaf nodes intersected by the search region is at most $k + 2.26\sqrt{ck} - 2c$. Since each leaf node contains $c$ points, the expected number of leaf nodes that were accessed to get these points is bounded by $k/c + 2.26\sqrt{k/c} - 2$.

To summarize, the expected number of leaf node accesses is $O(k + \sqrt{k})$ and the expected number of objects in the priority queue is $O(\sqrt{k})$. Intuitively, the "extra work" done by the algorithm comes from the boundary of the search region. Roughly speaking, the $k$ term in the expected number of leaf node accesses accounts for the leaf nodes completely inside the search region, while the $\sqrt{k}$ term accounts for the leaf nodes intersected by the boundary of the search region. The points on the priority queue lie outside the search region (since otherwise they would have been taken off the queue) but inside leaf nodes intersected by the boundary of the search region. If the average leaf node occupancy and average node fan-out are fairly high (say 50 or more), the number of leaf node accesses dominates the number of nonleaf node accesses, and the number of objects on the priority queue greatly exceeds the number of nodes on the queue. Thus we can approximate the total number of node accesses and total number of priority queue elements by the number of leaf node accesses and the number of objects on the priority queue. However, the traversal from the root of the spatial index to a leaf node containing the query object will add an $O(\log N)$ term to both of these measures.

If the spatial index is disk-based, the cost of disk accesses is likely to dominate the cost of priority queue operations. However, if the spatial index is memory-based, the priority queue operations are the single largest cost factor for the algorithm. In typical priority queue implementations (e.g., binary heap), the cost of each insertion and deletion operation is $O(\log m)$ where $m$ is the size of the priority queue. The number of objects inserted into the priority queue is $O(k+\sqrt{k})$, each for a cost of $O(\log \sqrt{k})$ (since the expected size is bounded by $O(\sqrt{k})$), for a total cost of $O(k + \sqrt{k}) \cdot O(\log \sqrt{k}) = O(k \log k)$ (again, if we take the nonleaf nodes into account, the formulas become somewhat more complicated).

The analysis that we have outlined is based on assumptions that generally do not hold in practice. In particular, the data is rarely uniformly distributed and the search region often extends beyond the data space. Nevertheless, our analysis allows fairly close predictions of actual behavior for two-dimensional point data even when these assumptions do not hold. For higher dimensions the situation is somewhat more complicated. A detailed analysis in that context is presented in [8].

## 4.7 Correctness

Now let us turn to the correctness of the algorithm in Figure 3. We ignore for the moment the issue of reporting an object more than once. Given a data object $o$, define its *ancestor set*, denoted by $A(o)$, to include $o$ itself, leaf nodes $n$ that contain $o$ for which $d_o(q,o) \geq d_n(q,n)$ (at least one such node is guaranteed to exist by the consistency of the distance functions), and all ancestors $n'$ of $n$. Applied recursively, the consistency property ensures that $d_o(q,o) \geq d_n(q,n')$. The elements in $A(o)$ can be interpreted as representing the object $o$. The following theorem guarantees that an unreported object always has a representative on the queue. This directly implies that every object will eventually be reported, since only bounded numbers of objects and nodes are ever put on the queue.

**Theorem** Let $R$ be the set of objects already reported, and $Q$ the set of elements on the queue. The following is an invariant for the outer **while**-loop of INCNEAREST: For each object $o$ in *SpatialIndex*, we have $A(o) \cap (Q \cup R) \neq \emptyset$ (i.e., at least one element in $A(o)$ is in $Q$ or in $R$).

*Proof:* We prove the theorem for an arbitrary object $o$ by induction. Since we choose $o$ arbitrarily, the proof holds for all objects. The induction is on the number of loop executions. If we can show that the invariant holds before the first execution, and that no loop execution falsifies it (i.e., makes it not hold after the execution of the loop, assuming that it held before the execution), then we have shown that the invariant always holds. Clearly, it holds initially, as the only element on the queue is the root node of *SpatialIndex*, and the root is an ancestor of all nodes and thus is in $A(o)$ for $o$.

Now assume that the invariant holds at the beginning of an execution of the **while**-loop. We will show that it also holds at the end of it. If $o \in R$ (i.e., $o$ has been reported), the invariant trivially holds, as $o$ will not be affected during the loop execution. Otherwise, by the assumption that the invariant holds, there exists some $a \in A(o)$ such that $a \in Q$. The invariant is unaffected if the next element to be dequeued is not $a$, so let us assume that $a$ will be dequeued next.

If $a = o$, then $o$ is subsequently reported, thereby moving from $Q$ to $R$, and the invariant is maintained. If $a$ is a node, we consider the case of a leaf and nonleaf node separately:

1. If $a$ is a leaf node, the **for**-loop at line 11 enqueues all objects with a distance from $q$ of at least $d_n(q,a)$ (i.e., at least DIST($QueryObject, Element$)). Since $o$ is stored in $a$ (recall that $a \in A(o)$) and since $d_o(q,o) \geq d_n(q,a)$ by the construction of $A(o)$, $o$ is indeed put on the queue.
2. If $a$ is a nonleaf node, then all its child nodes are enqueued. Since $a$ is in $A(o)$ (i.e., $a$ is an ancestor of a leaf node $n$ that contains $o$), at least one of the child nodes of $a$ is in $A(o)$, maintaining the invariant.

Thus we see that for both leaf and nonleaf nodes, at least one of the enqueued elements is in $A(o)$. Thus the invariant is maintained for object $o$. Since $o$ was chosen arbitrarily, we have thus shown that the invariant holds for all objects. ■

As mentioned, the theorem guarantees that an unreported object always has a representative on the queue. Since elements are retrieved from the queue in order of distance, and all elements in $A(o)$ are no farther from the query point than $o$, at some point $o$ will be put on the queue and eventually reported. Also, when $o$ is reported, it is indeed the next closest object to $q$. If not, then there exists an unreported object $o'$ closer to $q$. However, since all representatives of $o'$ are also closer to $q$ than $o$ is, at least one of them would be dequeued before $o$, contradicting the assumption that $o$ was most recently dequeued.

The correctness of the duplicate removal (lines 6–8 in Figure 3) follows directly from the ordering imposed on the priority queue. Thus the only way an object can be reported more than once is if it is inserted again into the queue after it has been reported. However, this is avoided by the test in line 12, and the fact that nodes are always processed before objects at the same distance from the query object.

### 4.8 Priority Queue

The cost of priority queue operations plays a role in the performance of the incremental nearest neighbor algorithm. The larger the queue size gets, the more costly each operation becomes. Also, if the queue gets too large to fit in memory, its contents must be stored in a disk-based structure instead of in memory, making each operation even more costly. An example of the worst case of the queue size for the R-tree incremental nearest neighbor algorithm arises when all leaf nodes are within distance $d$ from the query object $q$, while all data objects are farther away from $q$ than $d$. This is shown in Figure 6 where the query object as well as the data objects are points. In this case, all leaf nodes must be processed by the incremental algorithm, and all data objects must be inserted into the priority queue before the nearest neighbor can be determined. Note that any nearest neighbor algorithm that uses this R-tree has to visit all the leaf nodes, since the nearest neighbor is farther away from the query object than all the leaf nodes, and there is no other way to make sure that we have seen the nearest neighbor. Furthermore, note that a worst case like that depicted in Figure 6 is highly unlikely to arise in practice since it depends on a particular configuration of both the data objects and the query object.

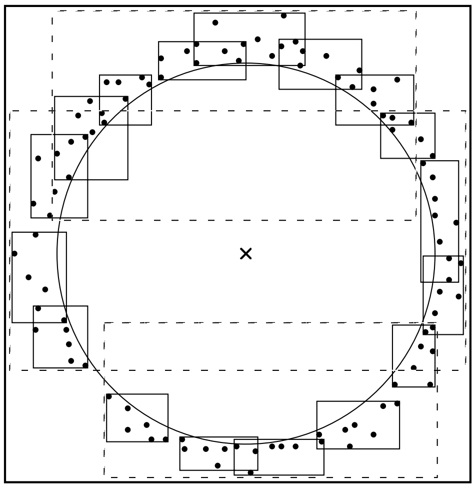

Figure 6: An example of an R-tree of points with node capacity of 8, showing a worst case for nearest neighbor search.

As pointed out in Section 4.6, the objects on the priority queue are contained in leaf nodes intersected by the boundary of the search region. For two-dimensional uniformly distributed data points we mentioned that the expected number of points in the priority queue when finding the $k$ nearest neighbors is $O(\sqrt{k})$. Even if $k$ is as large as several hundred million (of course, the data set has to be even larger than $k$), the size of the priority queue is still manageable for keeping in memory. However, more complex objects than points and very skewed data distributions may cause larger proportions of the objects to be inserted into the priority queue. Moreover, as the number of dimensions grows, the size of the priority queue as a function of $k$ tends to get larger (see Section 7). Thus we must be prepared to deal with a very large priority queue.

In cases where the priority queue exceeds the size of available memory it must be stored in whole or in part in a disk-resident structure. One possibility is to use a B-tree structure to store the entire contents of the priority queue. With proper buffer management, we should be able to arrange that the B-tree nodes that store elements with smaller distances (which will get dequeued early) will be kept in memory. However, we believe that when the priority queue actually fits in memory, using B-trees will be considerably slower than using fast heap-based approaches [20], since the B-tree must expend more work on maintaining the queue elements in fully sorted order. In contrast, heap methods impose a much looser structure on the elements. A hybrid scheme for storing the priority queue, where a portion of the priority queue is kept in memory and a portion is kept on disk, therefore seems more appropriate.

A simple way to implement a hybrid memory/disk-based priority queue is to partition the queue elements based on distance. Below, we outline how this can be done. The contents of the priority queue are split into three tiers. The first tier is kept in a memory-based heap structure, while the second and third tiers are kept in a disk file (the difference is that a little more structure is imposed on the contents of the second tier). Let $D_0, D_1, D_2, \ldots, D_m$ be some monotonically increasing sequence, where $D_0 = 0$ and $D_m$ is an upper bound on the largest possible distance from the query object $q$ to a data object (e.g., the distance from $q$ to the farthest corner of the data space). We use the sequence to define ranges of distances, and associate different ranges with the various tiers. When a new element with a distance of $r$ from the query object is inserted into the priority queue, that element gets added to the tier whose associated distance range matches $r$. Initially, tier 1 is associated with the distance range $[D_0, D_1)$ (i.e., queue elements in this range are stored in the memory-based heap structure), tier 2 with the range $[D_1, D_p)$, and tier 3 with the range $[D_{p+1}, D_m)$. The contents of tier 2 are divided into $p$ ranges, $[D_1, D_2), [D_2, D_3), \ldots, [D_p, D_{p+1})$. The value of $p$ depends on how many ranges it is cost-effective to maintain, but it can be as high as $m$. When tier 1 is exhausted, we move the elements in distance range $[D_1, D_2)$ from tier 2 to tier 1 and associate tier 1 with that distance range. The next time tier 1 is exhausted, we move elements in distance range $[D_2, D_3)$ into tier 1, and so on. If this happens often enough, eventually we will exhaust tier 2. When this happens, we scan the entire contents of tier 3 and rebuild tiers 1 and 2 with new ranges. Note that moving elements from tier 3 to tier 2 only when tier 2 is exhausted rather than each time tier 1 is exhausted reduces the number of scans of tier 3, which may contain a large number of elements.

In general, when the distance of the elements at the head of the priority queue is in the range $[D_i, D_{i+1})$ for some $i = 0, \ldots, m$ (i.e., all neighbors with distances less than $D_i$ from $q$ have already been reported), then tier 1 is associated with the range $[D_i, D_{i+1})$, tier 2 with the range $[D_{i+1}, D_{i+s+1})$, and tier 3 with the range $[D_{i+s+2}, D_m)$, where $s = p - (i \bmod p)$. We keep the elements in tier 2 in a set of linked lists, one for each interval $[D_j, D_{j+1})$ where $j = i + 1, \ldots, i + s$. In order to save on disk I/Os, we can associate a buffer with each of these linked lists and group elements into pages of fixed size. An alternative to using linked lists within the same file is to use a separate file for each range. Also, rather than associating range $[D_{i+1}, D_{i+s+1})$ with tier 2, we can associate with it the entire range $[D_{i+1}, D_{i+p+1})$, so that newly inserted elements in that range get inserted into tier 2 rather than tier 3. However, we still do not want to scan tier 3 each time we exhaust tier 1, so tier 3 will also contain elements in the range $[D_{i+s+1}, D_{i+p+1})$. These elements get moved into tier 2 when tier 3 gets scanned next, which happens when $i \bmod p = 0$.

A variation of this technique is to use an additional tier, between tier 1 and tier 2, in which elements are stored in an unsorted list in memory. The idea is that because we limit the size of the memory-based heap, the insertion and deletion operations on it are less expensive. Keeping the new tier 2 in memory but outside the heap makes it inexpensive to add elements to it (i.e., this does not require disk I/Os). Moreover, if only a small number of neighbors is requested, the elements in tier 2 will never need to be placed on the heap.

The remaining question is how to choose the sequence $D_0, D_1, D_2, ..., D_m$. A naive way is to simply guess some distance threshold $D_T$, and then set $D_i = i \cdot D_T$. Alternatively, we can assume some data distribution and use it to derive an appropriate sequence. For example, recall from Section 4.6 that under the assumptions made there, the expected number of leaf nodes intersected by the boundary of a search region of radius $r$ is bounded by $4(2r/s - 1)$, where $s = \sqrt{c/N}$ is the expected side length of each leaf node region.

Again, assuming that half of the points in these nodes (i.e., $c/2$) are outside the search region, the expected number of points on the priority queue is at most $\frac{c}{2}4(2r/s-1) = 2c(2r/s-1)$. Assuming that we have space in memory for $M$ priority queue elements means that $D_i$ must satisfy the equation $i \cdot M = 2c(2D_i/s - 1)$, so that

$$D_i = \left(\frac{i \cdot M}{2c} + 1\right) /2 \cdot s.$$

Of course, this derivation is based on assumptions that generally do not hold in practice. Nevertheless, it should work fairly well in practice for two-dimensional points. Moreover, it gives an indication of how to obtain such a sequence for other ways of analyzing the size of the priority queue.

## 5 $k$-Nearest Neighbor Search in R-trees

An alternative approach to nearest neighbor search in R-trees was proposed in [45]. This approach is applicable when finding the $k$ nearest neighbors where $k$ is fixed in advance. This is in contrast to the incremental nearest neighbor algorithm, where $k$ does not have to be fixed in advance. The key idea of the $k$-nearest neighbor algorithm is to maintain a global list of the candidate $k$ nearest neighbors as the R-tree is traversed in a depth-first manner. As we will see, the fact that the $k$-nearest neighbor algorithm employs a pure depth-first traversal means that at any step the algorithm can only make local decisions about which node to visit (i.e., the next node to visit must be a child node of the current node), whereas our incremental nearest neighbor algorithm makes global decisions based on the contents of the priority queue (i.e., it can choose among the child nodes of *all* nodes that have already been visited).

In this section, we first describe a somewhat simplified version of the $k$-nearest neighbor algorithm in [45] and show an example of its execution. Next, we prove that our simplified version is in fact equivalent to the algorithm presented in [45]: Both versions visit the same nodes in the R-tree. Finally, we show how the $k$-nearest neighbor algorithm can be transformed in a sequence of steps into an incremental algorithm.

### 5.1 Algorithm Description

In the $k$-nearest neighbor algorithm [45], the R-tree is traversed in a depth-first manner. The complications mentioned in Section 4 that arise in performing nearest neighbor search with a depth-first traversal are overcome by maintaining a list of the candidate $k$ nearest neighbors. In particular, once we reach a leaf node containing the query object, we insert the contents of that node into the candidate list, and unwind the recursive traversal of the tree. Once the candidate list contains $k$ members, the largest distance of any of its members from the query object can be used to prune the search.

Figure 7 shows the $k$-nearest neighbor algorithm. In the figure, *NearestList* denotes the list of the $k$ candidate nearest neighbors, and *NearestList.MaxDist* denotes the largest distance from the query object of any of the members of *NearestList*; if *NearestList* contains fewer than $k$ members, this distance is taken to be $\infty$. When an object is inserted into *NearestList* in line 4 of KNEARESTTRAVERSAL, an existing member is replaced if the list already contains $k$ members. In particular, we replace the member that is farthest from the query object (i.e., the one at distance *NearestList.MaxDist*). Before inserting an object into *NearestList*, we first make sure that its distance from the query object is smaller than *NearestList.MaxDist* (line 3 of KNEARESTTRAVERSAL). Note that *NearestList.MaxDist* decreases monotonically as more objects are inserted into the list, since we always replace objects with objects closer to the query object.

In KNEARESTTRAVERSAL, if *Node* is a nonleaf node, its child nodes are visited in order of distance from the query object. This is done by building the list *ActiveBranchList* of the entries in *Node*, and sorting it by distance from the query object (see Section 5.3 for different ways of defining this order). Next, we iterate through the list (in the sorted order) and recursively invoke KNEARESTTRAVERSAL on the child nodes. Once the distance of *Child* from the query object is larger than *NearestList.MaxDist*, we ignore *Child* and the rest of the entries in *ActiveBranchList*. We can do this because this means that no object in the subtree of *Child* (or the remaining entries in *ActiveBranchList*) will get inserted into *NearestList*.

```
KNEAREST($k$, QueryObject, SpatialIndex)
1  NearestList ← NEWLIST($k$)
2  KNEARESTTRAVERSAL(NearestList, $k$, QueryObject, SpatialIndex.RootNode)
3  return NearestList
```

```
KNEARESTTRAVERSAL(NearestList, $k$, QueryObject, Node)
1  if Node is a leaf node then
2    for each Object in Node do
3      if DIST(QueryObject, Object) < NearestList.MaxDist then
4        INSERT(NearestList, DIST(QueryObject, Object), Object)
5      endif
6    enddo
7  else
8    ActiveBranchList ← entries in Node
9    SORTBRANCHLIST(QueryObject, ActiveBranchList)
10   for each Child node in ActiveBranchList do
11     if DIST(QueryObject, Child) < NearestList.MaxDist then
12       KNEARESTTRAVERSAL(NearestList, $k$, QueryObject, Child)
13     else
14       exit loop
15     endif
16   enddo
17 endif
```

Figure 7: $k$-nearest neighbor algorithm.

The difference between the $k$-nearest neighbor algorithm in Figure 7 and the original presentation in [45] is in the treatment of *ActiveBranchList*. We use only one *pruning strategy* to eliminate entries from consideration, by comparing their distances to *NearestList.MaxDist*, while [45] identifies two other pruning strategies. However, in Section 5.4 we will show that the other two pruning strategies in fact do not allow any more pruning than the one that we use.

If the objects are stored outside the R-tree (i.e., the R-tree leaf nodes contain bounding rectangles and object references), a minor optimization can be made in line 4 of KNEARESTTRAVERSAL. We first compute the distance from the query object to the bounding rectangle. Only if this distance is less than *NearestList.MaxDist* do we compute the real distance from *Object* to the query object. Otherwise, *Object* is not accessed, thereby potentially saving a disk I/O, as in this scenario the objects are stored outside the R-tree. Recall that $d(q, r) \leq d(q, o)$ if $r$ is a bounding rectangle of the object $o$, i.e., the distance of $o$ from $q$ is never less than the distance of $r$ from $q$.

In [45] it is suggested that a sorted buffer be used to store *NearestList*. However, we found that for large values of $k$, the manipulation of *NearestList* started to become a major factor in the execution time of the algorithm. Therefore, we replaced the sorted buffer with a simple priority queue structure, sorted in decreasing order of distance, thereby making it easy to replace the farthest object.

### 5.2 Example

As an example of the algorithm, we describe its use in finding the three nearest neighbors to query point `q` in the R-tree given in Figure 1. Below, we show the steps of the algorithm and the contents of the *ActiveBranchList*s and of *NearestList*. The example makes use of the distances between `q` and the line segments and bounding rectangles given in Table 1. An invocation with node $x$ is denoted by $k$-NN($x$). We start by applying it to the root of the R-tree, `R0`. Next, we describe the subsequent invocations of the algorithm. Each of the line segment elements in *NearestList* is listed along with its distance from `q`. In our specification of *NearestList* we also list the maximum distance used for pruning (i.e., *NearestList.MaxDist*). Initially, *NearestList* is empty and the maximum distance is $\infty$.

1. $k$-NN(`R0`): *ActiveBranchList* for `R0` is (`R1`, `R2`).
   - (a) $k$-NN(`R1`): *ActiveBranchList* for `R1` is (`R4`, `R3`).
     - i. $k$-NN(`R4`): insert `d`, `g`, `h` on *NearestList*: $\{$(`h,17`), (`d,59`), (`g,81`) : `81`$\}$.
     - ii. $k$-NN(`R3`): insert `a`, `b` in *NearestList* (replacing `d`, `g`): $\{$(`h,17`), (`a,17`), (`b,48`) : `48`$\}$.
   - (b) $k$-NN(`R2`): *ActiveBranchList* for `R2` is (`R5`, `R6`).
     - i. $k$-NN(`R5`): `i` replaces `b`, but `c` is too distant: $\{$(`h,17`), (`a,17`), (`i,21`) : `21`$\}$.
     - ii. $k$-NN(`R6`): this invocation does not occur, as the distance of `R6` from `q` is $\geq 21$.

The final contents of *NearestList* is $\{$(`h,17`), (`a,17`), (`i,21`)$\}$ which is returned as the list of the three nearest neighbors of `q`.

### 5.3 Node Ordering and Metrics

The ordering used to sort the elements in *ActiveBranchList* in Figure 7 can be based on various metrics for measuring the distances between *QueryObject* and the elements' bounding rectangles. Two such metrics are considered in [45], MINDIST and MINMAXDIST. For bounding rectangle $r$ of node $n$, MINDIST$(q,r)$ is the minimum possible distance from $q$ to an object in the subtree rooted at $n$, while MINMAXDIST$(q,r)$ is the maximum distance from $q$ at which an object in the subtree rooted at $n$ is guaranteed to be found (i.e., it is the minimum of the maximum distances at which an object can be found). MINDIST and MINMAXDIST are calculated by using the geometry (i.e., position and size) of the bounding rectangle $r$ of node $n$ and do not require examining the actual contents of $n$. A more precise definition is given as follows. MINDIST$(q,r)$ is the distance from $q$ to the closest point on the boundary of $r$ (not necessarily a corner), while MINMAXDIST$(q,r)$ is the distance from $q$ to the closest corner of $r$ that is "adjacent" to the corner farthest from $q$. Figure 8 shows two examples of the calculation of MINDIST and MINMAXDIST, which are shown with a solid and a broken line, respectively. Notice that for the bounding rectangle in Figure 8a the distance from $q$ to $a$ is less than the distance from $q$ to $b$, thereby accounting for the value of MINMAXDIST being equal to the former rather than the latter, while the opposite is true for Figure 8b. In some sense, the two orderings represent the optimistic (MINDIST) and the pessimistic (MINMAXDIST) choice. To see this, observe that if $r_1$ and $r_2$ are minimum bounding rectangles in order of increasing value of MINDIST (i.e., MINDIST$(q,r_1)$ $\leq$MINDIST$(q,r_2)$), then at best, $r_1$ contains an object $o_1$ at a distance close to its MINDIST value, such that DIST$(q,o_1)$ $\leq$MINDIST$(q,r_2)$; but this need not hold, as $r_2$ may contain an object closer to $q$. If $r_1$ and $r_2$ are in order of increasing MINMAXDIST value, on the other hand, then in the worst case, the object in $r_1$ nearest to $q$ is at distance MINMAXDIST$(q,r_1)$, which is no larger than MINMAXDIST$(q,r_2)$.

Experiments reported in [45] showed that ordering *ActiveBranchList* using MINDIST consistently performed better than using MINMAXDIST. This was confirmed in our experiments, although we do not include

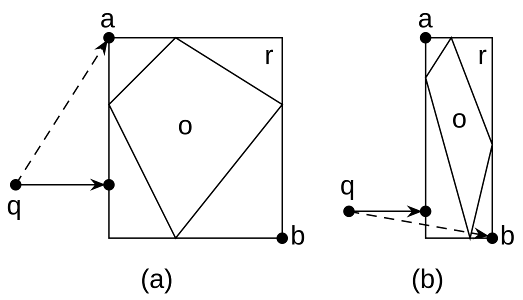

Figure 8: An example of MINDIST (solid line) and MINMAXDIST (broken line) for a bounding rectangle $r$. The distance of the object $o$ from $q$ is bounded from below by MINDIST$(q, r)$ and from above by MINMAXDIST$(q, r)$. Notice that in (b) point $b$ is closer to $q$ than point $a$ while this is not the case in (a).

that result in Section 6 which describes our experimental findings. We suspect that this indicates that the optimism inherent in MINDIST usually provides a better estimate of the distance of the nearest object than the pessimism inherent in MINMAXDIST, so that MINDIST order will in general lead to the nearest object(s) being found earlier in the *ActiveBranchList*. In this paper we therefore assume that *ActiveBranchList* is ordered using MINDIST. In fact, the algorithm in Figure 7 depends on this, as we discuss at the end of Section 5.4.

The metrics have other uses, regardless of which one is used for ordering *ActiveBranchList*. Since MINDIST represents the minimum distance at which an object could be found in a bounding rectangle $r$, it provides a means of pruning nodes from the search, given that a bound on the maximum distance is available. On the other hand, for any bounding rectangle $r$, MINMAXDIST$(q, r)$ is an upper bound on the distance of the object $o$ nearest to $q$. It should be clear that MINMAXDIST by itself does not help in pruning the search, as objects closer to $q$ could be found in elements of $n$ at positions with higher MINMAXDIST values. Moreover, since it only bounds the distance at which the closest element can be found, this property is of limited value, as it is only useful when we are seeking the nearest neighbor (i.e., $k = 1$).

## 5.4 Pruning Strategies

As already mentioned, the algorithm of [45] employs a set of three pruning strategies to prune entries from *ActiveBranchList* as the entries are processed. Two classes of pruning strategies are identified in [45], termed *downward pruning* and *upward pruning*. In downward pruning, entries on *ActiveBranchList* are eliminated prior to processing the nodes (i.e., before entering the **for**-loop in line 10 of KNEARESTTRAVERSAL in Figure 7). In upward pruning, entries on *ActiveBranchList* are eliminated after processing each node (i.e., after returning from the recursive call to KNEARESTTRAVERSAL in line 10 in Figure 7). Of the three pruning strategies discussed in [45], two are said to be applicable to downward pruning and one to upward pruning. Below, we will discuss these three pruning strategies in turn, and show that one of them is sufficient when used in a combination of upward and downward pruning[3].

Strategy 1 is used in downward pruning. It allows pruning an entry from *ActiveBranchList* whose bounding rectangle $r_1$ is such that MINDIST$(q, r_1)$ $>$MINMAXDIST$(q, r_2)$, where $r_2$ is some other bounding rectangle in *ActiveBranchList*. However, as already pointed out, using MINMAXDIST for pruning is of limited value as it is only useful when $k = 1$.

[3]It may appear that we use this pruning strategy only for upward pruning in line 11 of KNEARESTTRAVERSAL in Figure 7. However, since the condition is checked before the recursive call to KNEARESTTRAVERSAL, the **if** statement actually does both upward and downward pruning.

Strategy 2 prunes an object $o$ when $\text{DIST}(q, o) >$ $\text{MINMAXDIST}(q, r)$, where $r$ is some bounding rectangle. Again, this strategy is only applicable to $k = 1$. This strategy is claimed to be of use in downward pruning in [45], but its inclusion is somewhat puzzling, since it does not help in pruning nodes from the search. It is possible that the authors intended strategy 2 to be used to prune objects in leaf nodes. However, this does not appear to be particularly fruitful, since it still requires the objects to be accessed and their distances from $q$ calculated. Another possible explanation for the inclusion of this strategy is that it can be used to discard the nearest object found in a subtree $s$ in *ActiveBranchList* after $s$ has been processed. However, the purpose of this is not clear, since a better candidate will replace this object later on, anyway.

Strategy 3 prunes any node from *ActiveBranchList* whose bounding rectangle $r$ is such that $\text{MINDIST}(q, r) > NearestList.MaxDist$. It is applicable for any value of $k$ and in both downward and upward pruning. Note that although strategy 3 is not explicitly labeled as a downward pruning strategy in [45], its use in downward pruning is noted. In particular, before entering the **for**-loop in line 10 of KNEARESTTRAVERSAL in Figure 7, we can eliminate entries in *ActiveBranchList* with distances larger than *NearestList.MaxDist* (no pruning will occur, though, unless *NearestList* contains at least $k$ entries).

Recalling that strategy 1 is only applicable when $k = 1$, it can be shown that even in this case applying strategy 3 in upward pruning eliminates at least as many bounding rectangles as applying strategy 1 in downward pruning. To see this, let $r$ be the bounding rectangle in *ActiveBranchList* with the smallest MINMAXDIST value. Using strategy 1, we can prune any entry in *ActiveBranchList* with bounding rectangle $r'$ such that $\text{MINDIST}(q, r') >$ $\text{MINMAXDIST}(q, r)$. However, strategy 1 will not prune $r$ or any entry in *ActiveBranchList* preceding it, regardless of the ordering used. If *ActiveBranchList* is ordered based on MINMAXDIST, this clearly holds, since $\text{MINDIST}(q, r) \leq$ $\text{MINMAXDIST}(q, r)$. If *ActiveBranchList* is ordered based on MINDIST, the nodes preceding $r$ have MINDIST values smaller than that of $r$, so their MINDIST values must also be smaller than $\text{MINMAXDIST}(q, r)$. Now, let us see what entries can be pruned from *ActiveBranchList* by strategy 3 after processing the node corresponding to $r$. In particular, at that point, $\text{DIST}(q, o) \leq$ $\text{MINMAXDIST}(q, r)$ where $o$ is the candidate nearest object; this follows directly from the definition of MINMAXDIST. Therefore, when strategy 3 (based on $\text{DIST}(q, o)$) is now applied to *ActiveBranchList*, it will prune at least as many entries as strategy 1 (based on $\text{MINMAXDIST}(q, r)$).

The fact that we have eliminated strategies 1 and 2, and we are interested in finding more than $k$ neighbors, implies that MINMAXDIST is not necessary for pruning as it is not involved in strategy 3. Thus, assuming that MINMAXDIST is not used for node ordering, the CPU cost of the algorithm is reduced, since we do not have to compute the MINMAXDIST value of each bounding rectangle; this is especially important because MINMAXDIST is more expensive to compute than MINDIST. We also observe that there is really no need to distinguish between downward and upward pruning in the sense that there is no need to explicitly remove items from *ActiveBranchList*. Instead, we just test each element on *ActiveBranchList* when its turn comes. If *ActiveBranchList* is ordered according to MINDIST, then once we prune one element, we can terminate all computation at this level, as all remaining elements have larger MINDIST values. This is exactly what we do in the **if** statement in line 11 of KNEARESTTRAVERSAL in Figure 7.

### 5.5 Transformation

In this section we show how the $k$-nearest neighbor algorithm can be transformed into an incremental algorithm, and that the result is identical to our R-tree incremental algorithm. This discussion reveals the main difference between the two algorithms, namely that the control structure of the $k$-nearest neighbor algorithm is fragmented among the nodes on the path from the root to the current node (as specified in the *ActiveBranchList* of each invocation of the algorithm), while the incremental nearest neighbor algorithm employs a unified control structure embodied in its priority queue.

Recall that the R-tree $k$-nearest neighbor algorithm traverses the R-tree in a depth-first manner. It keeps track of the state of the traversal (i.e., which nodes or bounding rectangles it has yet to process) by use of an *ActiveBranchList* for each level (note that at most one node is active at each level at any given time). In addition, in its original formulation (i.e., assuming a sorted buffer implementation) it keeps track of the distances from the query object of the data objects that it has seen by use of *NearestList* sorted in increasing order of distance from the query object. Output of the $k$ nearest neighbors only occurs at the end of the traversal since the R-tree is being traversed in its entirety (subject to the pruning of nodes in *ActiveBranchList*).

If we want to transform the R-tree $k$-nearest neighbor algorithm into an incremental algorithm, we need to also keep track of the nodes in the R-tree that have been encountered (i.e., inserted into an *ActiveBranchList*) but not processed. These are the elements of the various instances of *ActiveBranchList*; let $B$ denote their union. We assume that elements are removed from *NearestList* as they are processed. With the aid of $B$, it is now possible to tell if the first element $o$ in *NearestList* should be reported as the next nearest neighbor to $q$. In particular, this is the case if $o$ is closer to $q$ than the closest node in $B$, as all objects not yet encountered are in subtrees of nodes in $B$. Without the global knowledge that $B$ embodies, it is not possible to report even the nearest neighbor until we have unwound the recursive traversal of the algorithm up to the root node of the R-tree, because before then we do not know what is in the other subtrees of the root.

The $k$-nearest neighbor algorithm can be modified to maintain this global unprocessed node list $B$, thereby enabling it to report nearest neighbors incrementally. This process can be made more efficient by keeping $B$ in sorted order based on distance from $q$. However, this still leaves open the question of how to efficiently add and remove nodes from $B$.

Having made this modification, we can go even further and change the control structure. In particular, instead of keeping to the strict depth-first traversal, the list $B$ can be used to guide the traversal, i.e., the node in $B$ closest to $q$ is taken as the next node to process. As a node is processed, it is deleted from $B$, and as a nonleaf node is processed, all its entries are added to $B$. Note also that as described above, $B$ is sorted in MINDIST order. It could be ordered by MINMAXDIST, but such an ordering has the disadvantage that the node on $B$ nearest to $q$ would not be immediately accessible. Furthermore, we observe that the penalty for choosing to process a wrong node is far less than the penalty for doing so in the $k$-nearest algorithm since all that is done is to inspect the node's entries, rather than traversing its entire subtree (subject to pruning, of course).

Note that with this transformation it is now possible to allow an unbounded $k$, as the last element in *NearestList*, i.e., the one farthest from $q$, no longer plays a role. Of course, this also means that *NearestList* is no longer bounded, except by the total number of objects in the R-tree.

The entire process can be performed most easily by merging $B$ and *NearestList* into one list called *CombinedNearestList*. By ordering *CombinedNearestList* in increasing order of distance we are able to preserve the role of the previous contents of *ActiveBranchList*, in that nodes that would have been pruned will be at greater distances in the *CombinedNearestList* than the $k^{th}$ nearest object. Thus they and their subtrees will not be traversed when outputting the $k$ nearest neighbors. Observe that the transformed algorithm makes use only of the MINDIST distance metric, thereby rendering moot the issue of whether or not to use the MINMAXDIST [45] metric. Also, the transformed algorithm will in general achieve more pruning of nodes than the original $k$-nearest neighbor algorithm.

We conclude our discussion of the $k$-nearest neighbor algorithm by pointing out that the transformation yields an algorithm equivalent to the incremental algorithm presented earlier when *CombinedNearestList* is organized with a priority queue.

## 6 Experimental Results

In order to evaluate the R-tree incremental nearest neighbor algorithm of Figure 4 (denoted by INN), we compared it to the result of using the R-tree $k$-nearest neighbor algorithm of [45] (denoted by $k$-NN) for distance browsing (Section 6.1). We also measured the incremental cost of using INN, i.e., the cost of obtaining the $k + 1^{st}$ neighbor once we have already obtained the $k^{th}$ neighbor (Section 6.2). By varying the number of objects that are browsed, we were able to see the true advantage of our method of computing the nearest neighbors incrementally rather than committing ourselves to a predetermined number of nearest neighbors, as would be the case if we used the $k$-nearest neighbor algorithm. (Remember that we do not know in advance how many objects will be browsed before finding the desired object.) Finally, we compare INN with $k$-NN for computing the result of a $k$-nearest neighbor query (Section 6.3). These studies were performed for small numbers of neighbors (i.e., less than 25), as this is the most common situation in which distance browsing is useful. Nevertheless, we also treat the case of a large number of neighbors in Section 6.3.

In the experiments mentioned above we measured the execution time, the disk I/O behavior and the number of distance computations for two representative maps. In order to discern whether the size of the maps was a factor, we performed experiments in which the size was varied (Section 6.4). In addition, for an extreme case, we experimented with a very large data set (Section 6.5). Finally, in Section 6.6 we report the maximum size of the priority queue for the experiments in Sections 6.3 and 6.4.

The data sets used in the experiments consisted of line segments, both real-world data and randomly generated data. The real-world data consisted of four data sets from the TIGER/Line File [43] (see Figure 9):

1. Howard County: 17,421 line segments.
2. Water in the Washington DC metro area: 37,495 line segments.
3. Prince George's County: 59,551 line segments.
4. Roads in the Washington DC metro area: 200,482 line segments.

The randomly generated line segment maps were constructed by generating random infinite lines in a manner independent of translation and scaling of the coordinate system [38]. These lines were clipped to the map area to obtain line segments, and then subdivided further at their intersection points with other line segments so that at the end, line segments meet only at endpoints. Note that the random maps do not necessarily model real-world maps perfectly. In particular, by their construction, random maps cover an entire square area, whereas this is not the case for most real maps (e.g., TIGER/Line File county maps). Furthermore, the random maps tend to be rather uniform, while real maps tend to have dense clusters of small line segments mixed with more sparsely covered areas. Nevertheless, these randomly generated maps do capture some important features of real maps (e.g., there is a low probability of more than four line segments meeting at a point), and they enabled us to run the experiments on a wide range of map sizes for maps with similar characteristics.

Our experiments differ from those in [45], which used a Hilbert-packed R-tree [31, 46], whereas we used an R$^*$-tree. The Hilbert-packed R-tree is a static structure, constructed by applying a Peano-Hilbert space ordering (e.g., [47]) to spatial objects on the basis of their centroids. The leaf nodes of the R-tree are then built by filling them with the objects, and the nonleaf nodes are built on top, with bounding rectangles computed for the nodes. Notice that the conventional R-tree node splitting rules were not applied in the construction of the Hilbert-packed R-tree since each node is filled to capacity by the Hilbert-packed R-tree construction algorithm. As we are interested in dynamic environments we chose to use the R$^*$-tree rather than the Hilbert-packed R-tree for our experiments except where noted.

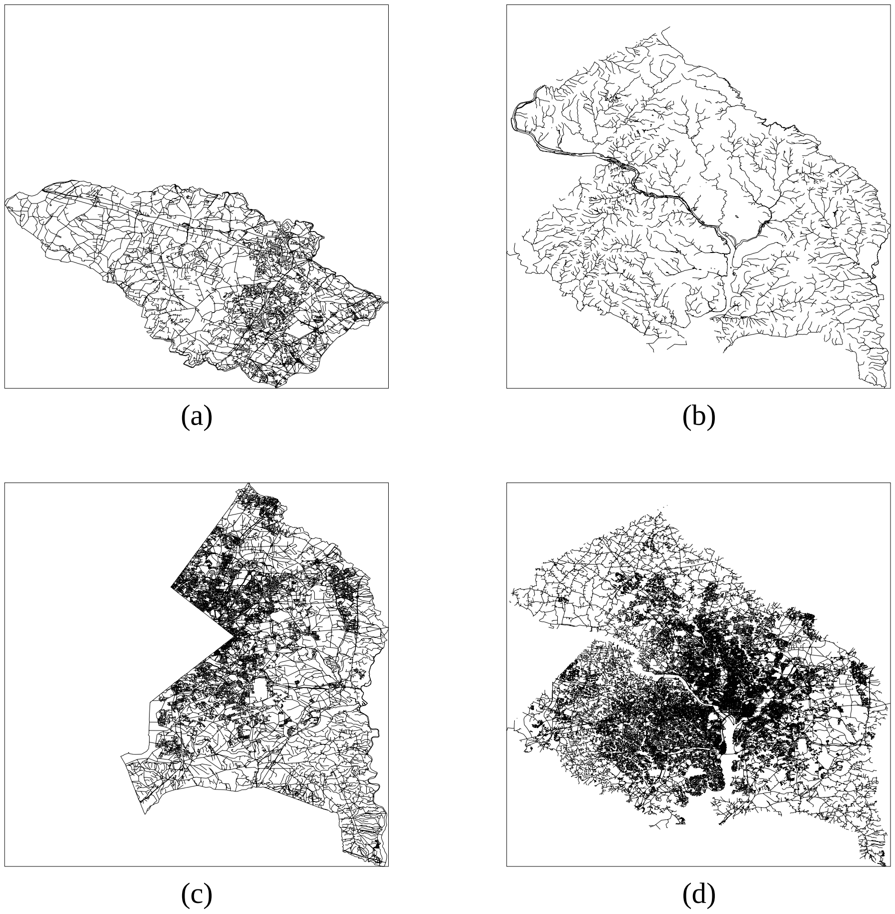

Figure 9: The four real-world data sets from the TIGER/Line File: (a) Howard, (b) Water, (c) PG, and (d) Roads.

Most of the data sets that we used were small enough to fit in main memory of many modern computers (except in the experiments reported in Section 6.5). Nevertheless, we used a disk-based R-tree structure, and employed buffers to store a limited number of recently used R-tree nodes (128). We therefore believe that our results will scale well to large data sets. The fact that we employ buffered I/O, with the added possibility of a requested disk block being in a disk cache or in operating system buffers, complicates the comparison between the two algorithms. There are two extremes: for each I/O, the requested disk block is found in memory, or every I/O leads to disk activity. Given a query for a fixed number of neighbors, the incremental nearest neighbor (INN) algorithm shows less improvement over the $k$-nearest neighbor algorithm ($k$-NN) in the former case (i.e., if the entire data sets resides in memory), and may even be slower, as will be seen for the small random data sets. This is mainly due to the overhead incurred by priority queue operations. However, for the other extreme the INN algorithm would show even more advantage than we found, as it always requests fewer R-tree nodes and objects than the $k$-NN algorithm.

For each experiment, we ran multiple queries on the same data set for the same number of neighbors. This was done so that more than one query point could be tested, as well as to make sure that the timing results were meaningful (given the timing granularity of the system we used). Since our R-tree implementation utilizes buffered I/O, this means that a query may access disk blocks that have already been loaded into the buffer by earlier queries in the same sequence. We feel that this was a reasonable choice to make, since the buffers were small compared to the data size, and clearing them prior to each query would have affected the timing results. Also, in a real world scenario, it is likely that a user will execute more than one query for a given map.

We use three measures for comparing the algorithms: execution time, R-tree node I/O (frequently referred to as *disk I/O* [6, 32]), and object distance calculations. The R-tree node I/O is reported as the number of accesses, and may not correspond to actual disk I/O if nodes can be found in database or system buffers. However, we have found that the number of accesses predicts the relative performance of actual disk I/O reasonably well. Furthermore, any saving due to buffering will show up in reduced execution time. Thus we used the disk I/O characterization.

In all the experiments that we conducted, the maps were embedded in a 16K by 16K grid, and the capacity of each R-tree node was 50. In order to simplify the analysis of the execution time results, we chose to store the actual line segments in the R-tree leaf nodes instead of just their bounding boxes. Also, the organization of the external object storage has a large effect on the performance, and thus introduces an extra variable into the comparison of the two algorithms. Query points were uniformly distributed over the space covered by the map data, and the distance functions used to measure the distances of lines and bounding rectangles from the query points were based on the squared Euclidean metric (in order to avoid computing square roots). The experiments were run sufficiently often to obtain consistent results with a different query point each time. Execution times are reported in milliseconds per query; they include the CPU time consumed by the algorithm and its system calls. We used a SPARCstation 5 Model 70 rated at 60 SPECint92 and 47 SPECfp92, and a GNU C++ compiler set for maximum optimization (–O3).

## 6.1 Cumulative Cost of Distance Browsing

In this section we focus on the distance browsing query when we do not know in advance how many neighbors will be needed before the query terminates. In this case, we need to reapply the $k$-nearest neighbor algorithm as the value of $k$ changes. In contrast, in the case of the incremental nearest neighbor algorithm, we need to reinvoke the algorithm to obtain just one neighbor (i.e., the next nearest one). For these experiments we used the map of Prince George's County (denoted by *PG* in the figures) as well as a randomly generated line map of a similar size, containing 64,000 lines (denoted by *R64K*). We included the random line map to see if the performance was affected by some unknown characteristics of the PG map.

Figures 10 through 12 show each measure's cumulative cost for distance browsing through the database by finding the neighbors incrementally. There are a number of ways of using a $k$-nearest neighbor algorithm to perform distance browsing. In our tests (shown in the figures) we use two such methods: (1) Execute $k$-NN each time we need a new neighbor. (2) Invoke $k$-NN for every five neighbors. Thus, for example, in case (2) the cost of computing the $11^{th}$ through $14^{th}$ neighbors is the same as the cost of computing the $15^{th}$ neighbor (which requires invoking the $k$-NN algorithm for $k$ = 5, 10, and 15). From the figures, it is clear that using the incremental nearest neighbor (INN) algorithm for distance browsing significantly outperforms simulating incremental access with the $k$-NN algorithm. In fact, the difference quickly becomes an order of magnitude. The figures use a logarithmic scale for the $y$-axis in order to bring out relative scale. Since the differences were so great, in order to simplify the presentation, we include results only for the $k$-NN algorithm for the PG map, as the results for the random data were similar.

The method that we used above for choosing the value of $k$ when performing distance browsing with the $k$-NN algorithm is not the best that we can do for larger values of $k$. For example, it would be better to multiply $k$ by 2 each time the algorithm must be re-invoked. In addition, the $k$-NN algorithm can be adapted to make it more suitable for use in distance browsing. In particular, after finding the $m$ nearest neighbors and determining that we must find the $m' > m$ nearest neighbors, we can use the distance of the $m^{th}$ nearest

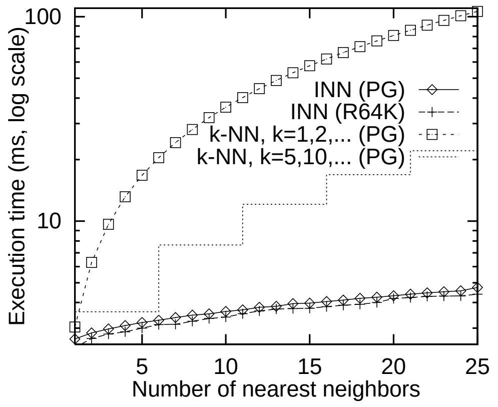

Figure 10: Cumulative execution time for distance browsing.

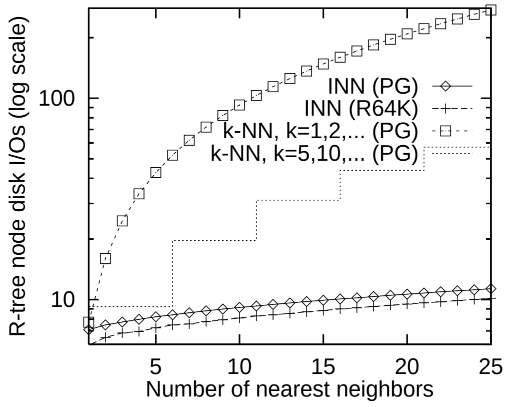

Figure 11: Cumulative R-tree node disk I/O for distance browsing.

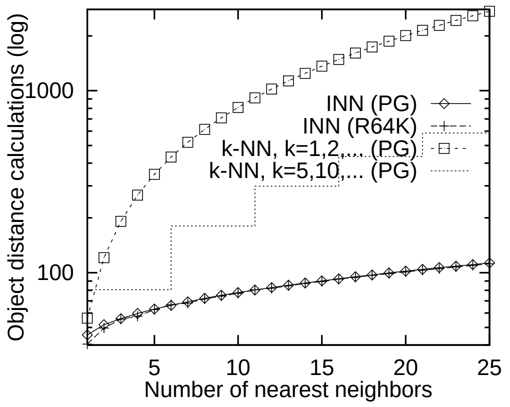

Figure 12: Cumulative object distance calculations for distance browsing.

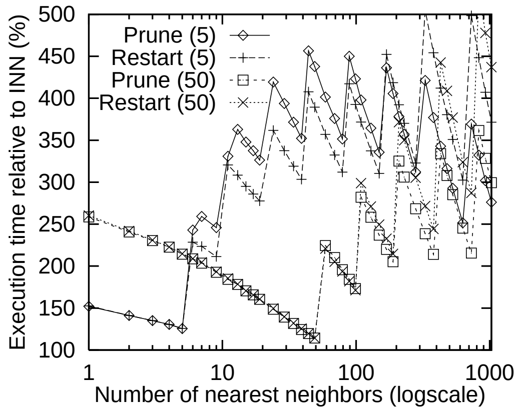

Figure 13: Execution time of $k$-NN relative to that of INN when used for distance browsing when the $k$-NN approach is made as good as possible.

neighbor as a minimum distance when the $k$-NN algorithm is re-invoked with $k = m'$ (actually, $k$ is set to $m' - m$, since the $m$ nearest neighbors would be excluded from the search). This minimum distance can be used to prune the search in much the same way as we described using minimum distance in the INN algorithm in Section 4.5. Some complications arise if other objects have the same distance from $q$ as the $m^{th}$ nearest neighbor. The best way to resolve this is to return all neighbors with that distance, which means that sometimes we obtain more neighbors than we requested. In Figure 13 we compare the execution time when using such an adapted $k$-NN algorithm (labelled "Prune") for distance browsing to the execution time when using the INN algorithm. Also, we show the result for the unmodified algorithm, where we must restart the search from scratch when the $k$-NN algorithm must be re-invoked (labelled "Restart"). We show the results only for the real-world data set (PG), as they were almost identical when using the random data set.

We use two different starting values for $k$ in Figure 13, namely 5 and 50 (shown in parentheses). Each time the $k$-NN algorithm is re-invoked, $k$ is doubled. The figure shows that if the $k$-NN algorithm must be re-invoked at least once, it usually takes more than twice (and up to nearly five times) as long as the INN algorithm. Using the "Prune" variant of the $k$-NN algorithm does not pay off unless a rather large number of neighbors is needed (over 100 or 200 in these experiments). The reason why this variant takes longer for a smaller number of neighbors is that not enough nodes get pruned to offset the cost of more node distance computations (for each node we must compute two distances, a minimum and a maximum, instead of just the minimum). Another observation is that the $k$-NN approach is highly sensitive to the initial value of $k$, and which initial value is better depends on how many neighbors we need (which we do not know in advance in distance browsing). The spikes on the curves occur where the $k$-NN algorithm is re-invoked an additional time for higher values of $k$, and between a spike and the next low point, no more neighbors are computed[4]. The reason the slope of the curve decreases after each spike is that in the range from a spike to the next low point, the cost of the $k$-NN approach remains constant (since no more neighbors are computed) while the cost of the INN approach increases gradually as we must compute additional neighbors. Note that the absolute low point on the two curves corresponds to the case where the number of neighbors needed happens to be equal to the initial value of $k$ (5 and 50, respectively). For those values of $k$, the $k$-NN algorithm is not much slower than the INN algorithm (about 25% slower for $k = 5$ and 14% slower for $k = 50$).

## 6.2 Incremental Cost of Distance Browsing

The results of the experiments conducted in Section 6.1 show the total cost of distance browsing after retrieving the $k^{th}$ neighbor. Using INN to implement each browsing step requires us to examine just one neighbor regardless of how many browsing steps we have already executed. In contrast, use of $k$-NN for distance browsing requires us to examine $k + 1$ neighbors when $k$ browsing steps have already been executed. In this section, we compare the two algorithms in terms of the cost of each browsing step (i.e., the incremental cost). This is shown in Figures 14 through 16. For the INN algorithm, the incremental cost can be seen to fluctuate somewhat, but it is always at least one order of magnitude less than the cost of the $k$-NN algorithm once the first neighbor has been obtained. Although not shown here, we found that this holds for all values of $k$. Again, we use a logarithmic scale for the $y$-axis so that the fluctuation in the cost of the incremental algorithm can be seen more clearly.

We evaluated the incremental execution time for up to 1000 neighbors in the PG map. Interestingly, we found that the incremental execution time clusters around an average of about .04 ms after the first 100 neighbors or so. This is in agreement with the results that we will discuss in Section 6.3, where we find that the average execution time per neighbor is around .04 ms when retrieving a few thousand neighbors or more in the PG and R64K maps. Thus we see that for a given map, the incremental execution time is remarkably close to constant after a small fraction of the objects have been retrieved (for the PG map this was around 100 neighbors or less than 0.2% of the map size).

For the R-tree node disk I/Os (Figure 15), the incremental algorithm (INN) was at least an order of magnitude better than $k$-NN after the first neighbor had been found. INN appears to be decreasing (i.e., between .1 and .2 after 25 neighbors), but levels off after a few hundred neighbors have been found. (The graph is not a step function because the number of node accesses is averaged over many queries.)

For the object distance calculations (Figure 16), the incremental algorithm (INN) was at least an order of magnitude better than $k$-NN after the first few neighbors had been found. The improvement approaches two orders of magnitude when 25 neighbors have been found, and continues in this manner for larger values of $k$ (not shown here). The average number of distance calculations performed for each incremental invocation is seen to be decreasing. This continues as more neighbors are retrieved and is below 1.2 after 300 neighbors. Thus INN quickly reaches a stage of accessing only about one object per reported neighbor.

[4]There should be a spike at 5 neighbors for "Prune (5)", but instead it occurs at 6 neighbors. The reason for this is that occasionally when requesting the nearest five neighbors, the sixth nearest neighbor has the same distance as the fifth one, so the $k$-NN algorithm does not need to be re-invoked when we want to obtain the sixth neighbor (the same is true for the second spike at 10 neighbors).

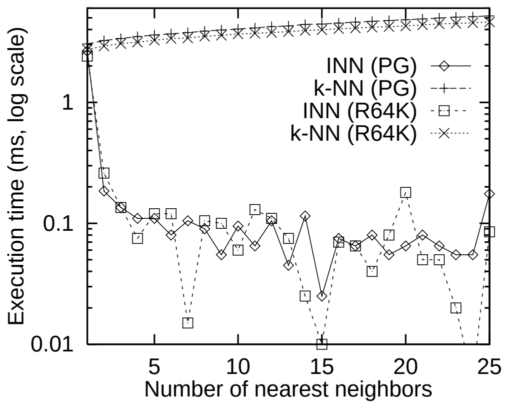

Figure 14: Incremental execution times for distance browsing.

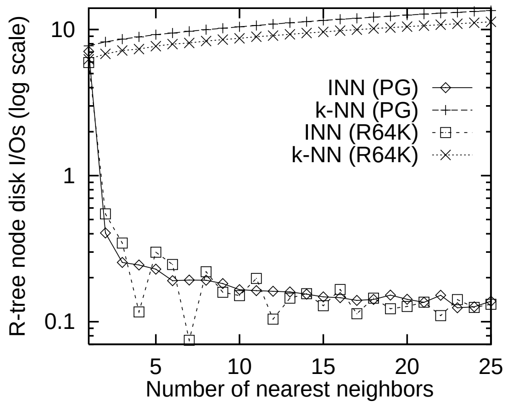

Figure 15: Incremental R-tree node disk I/O for distance browsing.

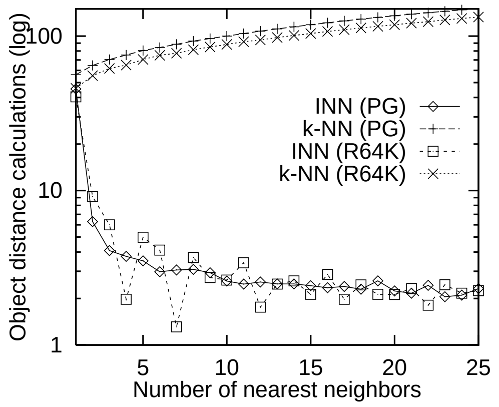

Figure 16: Incremental object distance calculations for distance browsing.

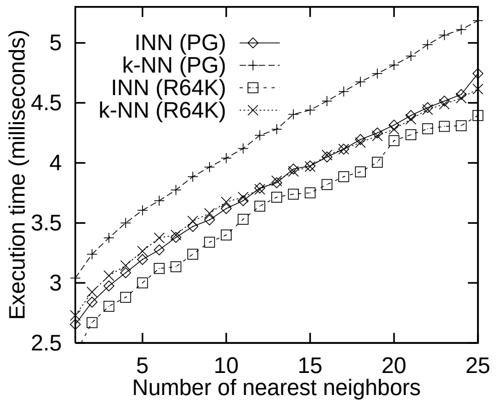

Figure 17: Execution time for $k$-nearest neighbor query.

## 6.3 $k$-Nearest Neighbor Queries

We now consider what the cost would be if we used the incremental nearest neighbor algorithm to solve the $k$-nearest neighbor problem. In other words, instead of browsing the database on the basis of distance, obtaining one neighbor at a time, we address the related problem of finding all $k$ neighbors at once, as we would do if we knew in advance how many neighbors we need. It is interesting to see if a performance penalty is incurred in solving this classical problem by using our incremental algorithm, rather than using approaches such as the $k$-NN algorithm which obtain all $k$ neighbors at once. We ran a sequence of tests in the same manner as those reported in Sections 6.1 and 6.2; the results are shown in Figures 17 through 19. From these figures we observe that using the INN algorithm leads to no sacrifice of performance. In fact, the incremental algorithm outperforms the $k$-nearest neighbor algorithm for the two maps for all values of $k$.

In addition to the experiments mentioned above, we ran $k$-nearest neighbor queries for values of $k$ from 1 up to the size of the data set. The results of these experiments are reported in Figures 20 through 22, where the cost measures are divided by the number $k$ of nearest neighbors, so that we are reporting the cost per neighbor. For the incremental nearest neighbor algorithm, this value is close to the average incremental cost for all but the smallest values of $k$ (for small $k$, the cost of retrieving the first neighbor dominates the cost). Dividing the cost measures by $k$ makes it possible to distinguish the cost measures for large values of $k$, which is difficult otherwise. In Figures 20–22, the $y$ axis uses a logarithmic scale.

For the execution time (Figure 17), we see that the two algorithms have similar growth patterns, with $k$-NN being somewhat slower than INN (about 11-14% for PG and 4-10% for R64K). While the improvement of INN over $k$-NN is modest for values of $k$ up to 25, Figure 20 reveals that the difference widens as $k$ grows larger, up to 75% for PG and 87% for R64K (for $k = 2^{15} = 32,768$). Even for values of $k$ as small as several hundred, the improvement of INN over $k$-NN is 20-30%. Note how the performance of INN for the two maps is very similar, whereas the performance of $k$-NN is worse for the PG map than for the R64K map. This observation holds for the other two cost measures as well. This suggests that INN is much less sensitive than $k$-NN to the distribution of data objects.

For very large values of $k$, we may ask whether it is not better to simply calculate distances for the entire database and then sort on the distance. If all the objects are ranked with the INN algorithm (or the $k$-NN algorithm), we must also compute the distances for all the objects in the database. The question then reduces to whether the overhead of the INN algorithm (for computing distances of nodes and manipulating the priority queue) exceeds the cost of sorting all the distance values once they have been computed. Interestingly, we found that for the PG map, using the INN algorithm to rank all the objects was faster than computing all the distances and sorting them, whereas the $k$-NN algorithm was a little slower than the sorting approach. Of course, this result cannot be generalized, as it depends on numerous factors, such as size of the data set, the spatial index being used, and whether the spatial objects are stored directly in the leaf nodes of the R-tree or in an external object table.

For the R-tree node disk I/Os (Figure 18) we see that INN is always better than $k$-NN, while the rate of growth is similar for both and appears to be linear in $k$ for low values of $k$. In fact, we found that this same pattern held for all values of $k$, as we see in Figure 21. The figures show that for each value of $k$, INN achieves more pruning of the input tree than $k$-NN. This partially explains its better execution time performance. For values of $k$ ranging between $2^6$ and $2^{15}$, INN accesses 20-53% fewer nodes for PG and 12-35% for R64K, with the largest difference occurring at $k = 2^9$ for both maps.

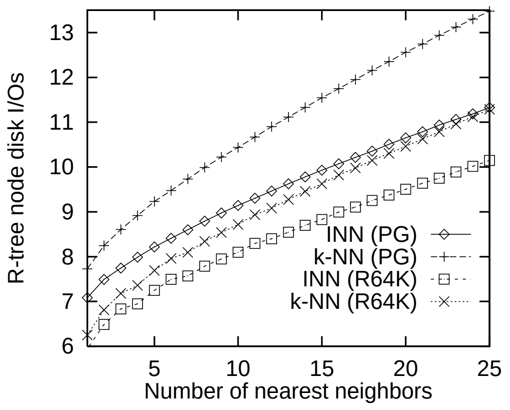

Figure 18: R-tree node disk I/O for $k$-nearest neighbor query.

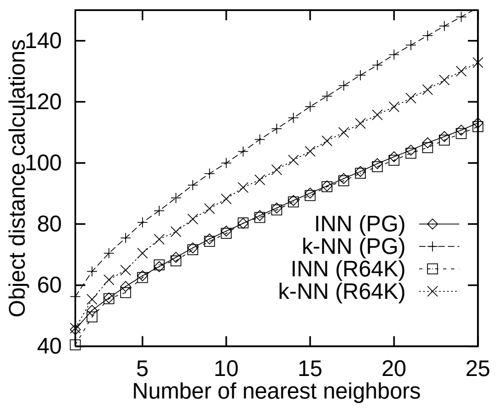

Figure 19: Object distance calculations for $k$-nearest neighbor query.

For the object distance calculations (Figure 19), we see that the INN algorithm again outperforms the $k$-NN algorithm. Figure 22 shows that this holds for all values of $k$, except when ranking all the map objects (in which case the number of distance calculations equals the number of map objects in both cases, as no pruning of objects or nodes is possible). The shapes of the curves in Figure 22 can be seen to be very similar to those in Figure 21. This is not surprising when we realize that the number of distance calculations is proportional to the number of R-tree leaf nodes that are accessed, and the leaf nodes in an R-tree greatly outnumber the nonleaf nodes.

Figure 23 shows the fraction of total execution time that is attributed to disk I/O operations in the above

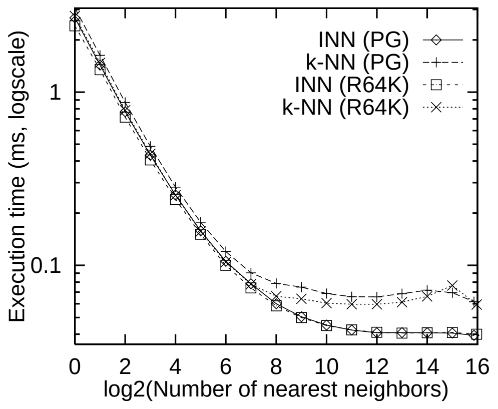

Figure 20: Execution time per neighbor for $k$-nearest neighbor query.

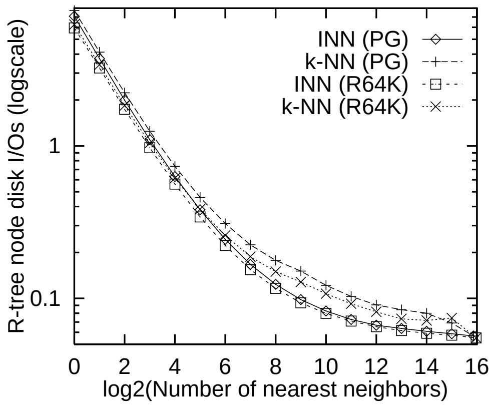

Figure 21: R-tree node disk I/O per neighbor for $k$-nearest neighbor query.

experiments. We compute this by recording the node accesses performed during the execution of the algorithms, and measuring the time needed to do nothing but access those nodes. The figure shows that the fraction of time spent by the INN algorithm in doing I/O is relatively constant, but starts to decrease for a large number of neighbors. In contrast, the fraction of time spent by the $k$-NN algorithm in doing I/O has a much larger variation, initially increasing rapidly, and decreasing significantly as the number of neighbors needed increases. In fact, eventually the fraction of time spent in doing I/O by the $k$-NN algorithm is considerably less than that spent by the INN algorithm as the number of neighbors increases; thus the INN algorithm becomes more efficient from a CPU cost perspective. (This may be due, in part, to the fact that for a large number of neighbors, the priority queue for the INN algorithm is considerably smaller than the *NearestList* maintained by the $k$-NN algorithm, as discussed in Section 6.6 and seen in Figure 31.)

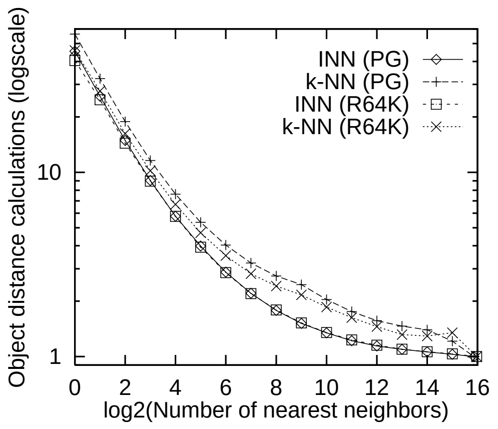

Figure 22: Object distance calculations per neighbor for $k$-nearest neighbor query.

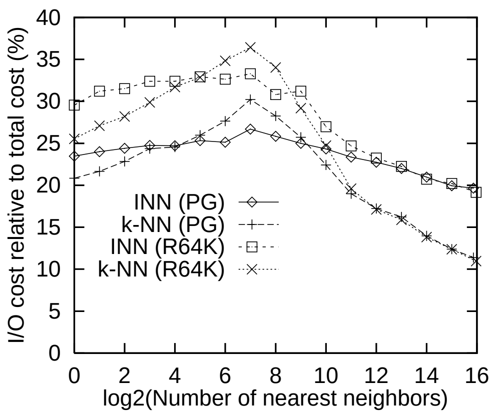

Figure 23: Fraction of total execution time taken by disk I/Os in computing $k$-nearest neighbor query.

## 6.4 Results for Varying Data Size

In the previous sections we investigated the performance of the two algorithms by varying the number of neighbors for both distance browsing and computing the $k$ nearest neighbors for similarly sized data sets. It is important that the performance of the algorithms remain reasonable even when the size of the data set is increased. To verify that this is indeed the case, we tested the performance of INN and $k$-NN on both random and real-world map data. Our experiments showed the same relationships for the two algorithms between the cumulative and incremental costs of distance browsing, as well as computing the $k$ nearest neighbors, that we found in the experiments reported in Sections 6.1–6.3 (provided the maps are nontrivial in size). In particular, they confirmed the superiority of INN over $k$-NN. In the interest of saving space we do not show these results here.

In the rest of this section, we focus on the relative behavior of the algorithms when finding the nearest neighbor (i.e., $k = 1$). This operation is important as it is the first step in distance browsing, and as we saw in Section 6.2 its execution time dominates the cost of distance browsing for small values of $k$.

Figures 24 through 26 show the performance of the two algorithms when finding the nearest neighbor. The $x$-axis in the figure is $\log_2 N$, where $N$ is the number of line segments. The real-world maps appear in the same order in which they were described above (from left to right: Howard County, Water, Prince George's County, and Roads). The random maps that we tested contained 1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000 and 256000 line segments.

For the execution time (Figure 24), we see that the INN algorithm is faster for most of the maps; $k$-NN took from 10-19% more time for the real-world maps, and up to 14% more time for the randomly generated maps. The exceptions are the three smallest randomly generated maps. This can be explained partly by the fact that these maps were small enough to fit in the R-tree node buffer, and partly by the fact that their small sizes gave less room for improvement (see Figures 25 and 26). Even so, for larger values of $k$, INN became better than $k$-NN for these data sets. For all the randomly generated maps, which have similar characteristics, the rate of growth of the execution time can be seen to be nearly identical for the two algorithms. In fact, the rate of growth appears to be very nearly logarithmic in the number of line segments (recall that the $x$-axis uses a log scale). The execution times for the real-world maps correlate remarkably well with the execution times for the random maps of comparable size.

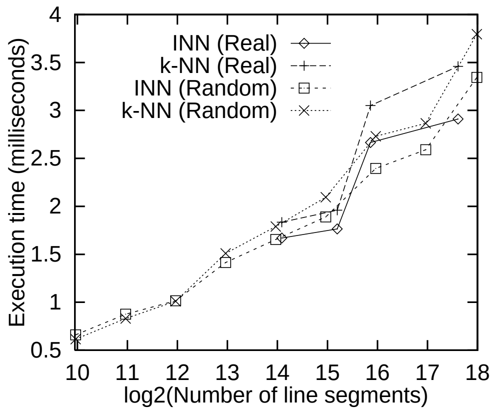

Figure 24: Execution time for finding one neighbor.

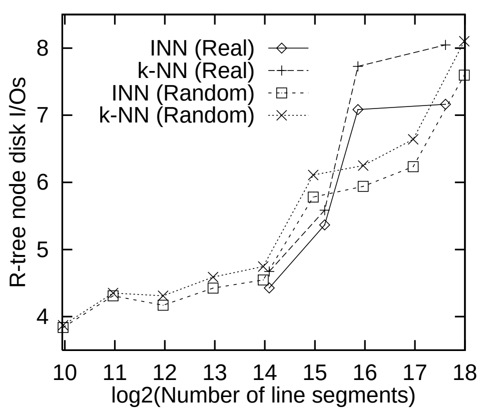

Figure 25: R-tree node disk I/O for finding one neighbor.

For the R-tree node disk I/Os (Figure 25) we find the same relative behavior of the algorithms, with INN being always better than $k$-NN, while the rate of growth is similar for both. The rate of growth appears to be logarithmic in the number of line segments. This compares with the results reported in [45] for $k$-NN, where it was observed that the number of R-tree node accesses grew linearly with the height of the tree. Our experiments are not in exact agreement with that observation, but asymptotically, the two observations are equivalent, since in R-trees the height of the tree grows logarithmically with the number of objects.

For the object distance calculations (Figure 26), again, INN performs better than $k$-NN.

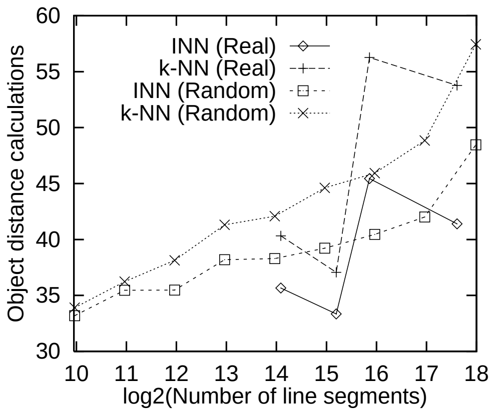

Figure 26: Object distance calculations for finding one neighbor.

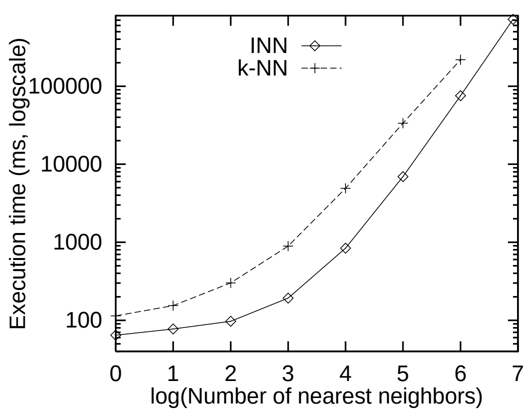

Figure 27: Execution time for large data set.

## 6.5 Results for Large Data Sets

Admittedly, the data sets that we used in the experiments reported above were moderate in size. For the largest data set that we used, the spatial index occupies approximately 9 MB of disk space, which is small enough to fit into the main memory of most modern computers. Even so, in our experiments, we only used a small amount of main memory for buffers (128 nodes), and the size of the priority queue remained small compared to the data size (100 KB in the worst case for the experiments in Section 6.3, or about 3% of the size of the map files). Thus we believe that our results will also hold for larger data sets, i.e., data sets much larger than the size of main memory.

In order to verify this claim, we conducted an experiment with a randomly generated data set of 8 million lines. As it was prohibitively slow to build an R$^*$-tree for such a large data set, we built instead a Hilbert-packed R-tree [31], which occupied almost 300 MB. We used the same level of fan-out (50) and the same amount of buffering (128 nodes) as in our previous experiments (though it might have been better to use a larger fan-out and buffer sizes for such a large data set). Incidentally, we found that both algorithms performed more poorly with a Hilbert-packed R-tree than with an R$^*$-tree for the same data set. This appears to be due to the greater amount of node overlap in the Hilbert-packed R-tree. The incremental nearest neighbor algorithm proved to be much less sensitive to the level of node overlap, due to its superior pruning of the R-tree nodes.

Figures 27–29 show the results of our experiments on this large map, which consisted of $k$-nearest neighbor queries for values of $k$ from 1 through the size of the data set (8 million). Unfortunately, we were not able to run the $k$-NN algorithm for $k = 8$ million, as there was not enough memory to hold the neighbor list for 8 million neighbors. This is in contrast to the INN algorithm, where the priority queue contained at most about 83,000 elements, or about 1% of the number of neighbors. The speedup in execution time for INN over $k$-NN ranged from 1.8 to 5.8. $k$-NN accessed from 1.8 to 5.3 times as many nodes and performed up to 6 times as many distance calculations as INN.

## 6.6 Priority Queue Size

In Section 4.8 we showed that in the worst case, all the data objects must be inserted into the priority queue when using the incremental nearest neighbor algorithm. In our experiments, however, we found that the priority queue remained modest in size. The size of the priority queue affects the performance of queue operations during the algorithm's execution. Also, a very large queue requires a disk-based implementation,

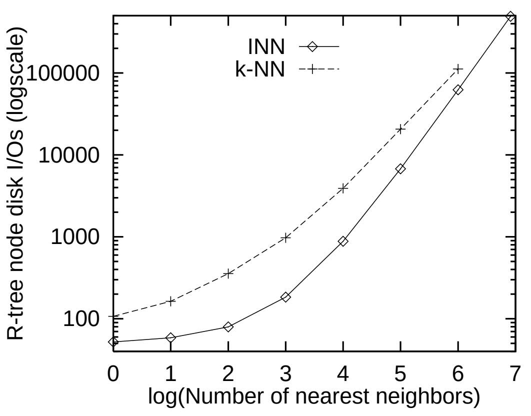

Figure 28: Node disk I/Os for large data set.

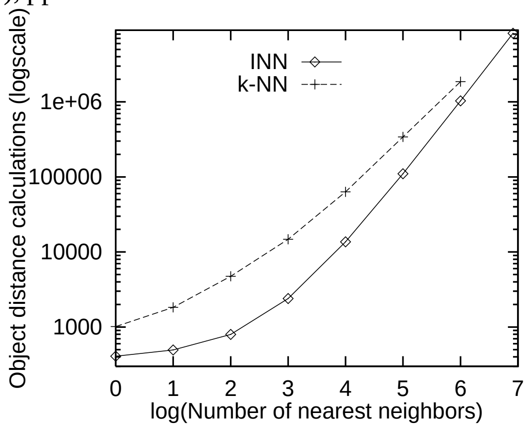

Figure 29: Object distance calculations for large data set.

thereby slowing the algorithm down. However, in most applications the maximum queue size remains relatively modest, which permits using a memory-based data structure for the queue. For example, consider Figure 30, which shows the maximum size of the queue when computing the nearest neighbor (i.e., $k = 1$) using the same data sets as in Section 6.4. Notice that for the worst case situation described above, in this first step of distance browsing for the given query object, all objects must be inserted into the queue before determining the nearest neighbor. From the figure it is evident that the maximum queue size grows remarkably slowly as the number of line segments increases. The results for the random maps suggest that this growth is logarithmic in the number of line segments.

Figure 31 shows the maximum size of the priority queue when using the incremental nearest neighbor algorithm after $k$ distance browsing operations for the maps used in Section 6.1 ($k$ ranged from 1 up to the size of the map). In the figure, the $y$-axis is logarithmic. We see that the maximum queue size $M$ grows extremely slowly. Note also that $M$ is relatively small (less than 5% in the worse case) in comparison with the sum of the number of data objects and R-tree nodes for the two comparably-sized maps, which is $M$'s theoretical maximum. When $k$ reaches a value of $2^{10} \approx 1000$, the priority queue needed by the incremental nearest neighbor algorithm is smaller than the priority queue needed to store the sorted buffer for the $k$-NN algorithm. A similar picture emerged for the large map used in Section 6.5, where the size of the priority queue was an even smaller fraction of the map size (1% in the worst case).

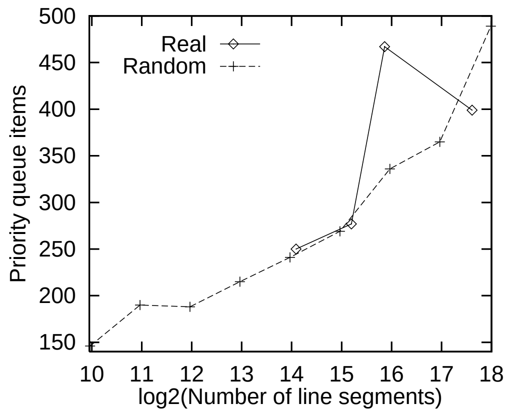

Figure 30: Maximum queue size for finding the nearest neighbor (i.e., $k = 1$).

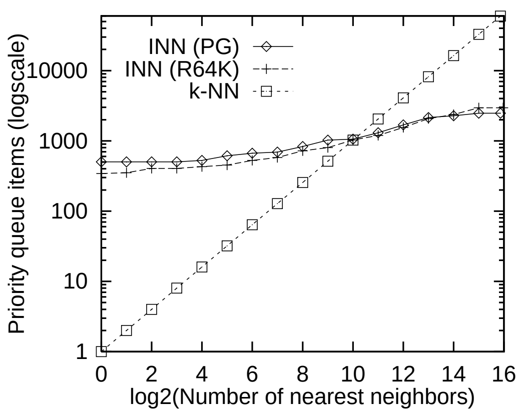

Figure 31: Maximum queue size for a wide range of $k$.

## 7 High-Dimensional Space

As already pointed out, the incremental nearest neighbor algorithm is independent of the dimensionality of the data objects and is equally applicable to data embedded in low-dimensional and high-dimensional spaces. Unfortunately, it is difficult to effectively index high-dimensional data and nearest neighbor search also becomes more costly. In this section we address some of the issues that arise. As it is hard to reason about arbitrary data distributions, some of the conclusions we draw are based on uniformly-distributed data.

High-dimensional data arises in a number of current applications, including multimedia databases, data warehouses, and information retrieval. Usually, such data is limited to points, but more general objects also arise [8]. As an example of an application that leads to high-dimensional data, color histograms have been used in image databases to allow searching for images with a specific color or with a combination of colors similar to some query image. The colors in an image are described by $d$-dimensional vectors, in which each element encodes the intensity of a particular range of colors (e.g., by using RGB values). To compare the closeness of the sets of colors in two images, a complex distance function is used, involving matrix multiplication. Using that distance function, we can use a nearest neighbor search on an image database to find the image closest in color to some query image. The number of dimensions, $d$, for color histograms is typically 64, 100 or 256. In other applications, the number of dimensions can be even higher (as much as several thousand).

Most spatial indexing structures do not work very well for high dimensions. The R-tree, for example, has been found to degenerate for dimensions higher than 7 or so [9]. Specifically, what happens is that even for range queries with small query windows, so many of the index pages must be read that reading them is more expensive than sequential scan of the data. Several indexing structures have been proposed to address this issue; for example the X-tree [9] and LSD$^{\mathrm{h}}$-tree [27], based on the R-tree and LSD-tree, respectively. However, even these often do not provide much speedup compared to sequential scan for dimensions above 20 or so. An approach often taken to speed up access to point data of very high dimension is to map the points into a space of lower dimension [18, 34], in which case we can use the incremental nearest neighbor algorithm on the lower-dimensional space. In order to guarantee the accuracy of the result, the output of the algorithm can be filtered based on the distances of the corresponding higher-dimensional points [48]. Another approach is to abandon the goal of indexing the data points based on space occupancy and instead use properties of the distance metric employed (see the discussion of the metric space model in Section 2). If a hierarchical index method based on distance (e.g., [11, 14, 52]) is employed, our algorithm is still applicable. In fact, the $k$-nearest neighbor algorithm presented in [14] is similar to our algorithm in that it uses a priority queue for nodes to guide the traversal of the index.

If we use the Euclidean distance metric, the nearest neighbor search region (Section 4.6) is spherical. On the other hand, the node regions for most types of spatial index structures are hyper-rectangular in shape. This has the effect of making nearest neighbor search more expensive, as more points are accessed than necessary. To see why this is true, consider that in two dimensions the areas of a square and a circle, both with radius $r$, are $4r^2$ and $\pi r^2$, respectively. Thus the ratio of the area of the circle to the area of the square is $\pi/4 \approx 79\%$. In three dimensions the ratio of the volume of a sphere to the volume of a cube is about 52%, and in four dimensions the corresponding ratio for a hypersphere and hypercube is 10%. In general, the ratio between the volume of a hypersphere and its circumscribed hypercube decreases exponentially with the number of dimensions. Intuitively, the reason for this is that the number of "corners" of the hypercube grows exponentially with dimension. This effect has a direct consequence for nearest neighbor search using the Euclidean distance metric. To see why, let us assume that we have uniformly-distributed data points inside a hypercube of radius $r$ and a search region of radius $r$ centered inside the hypercube; the hypercube represents the smallest bounding box of the set of hyper-rectangular leaf node regions that intersect the search region. Then the proportion of the data points inside the search region decreases exponentially with the number of dimensions; e.g., for four dimensions, only about 10% are inside the search region. The large number of data points inside the hypercube but outside the search region represent wasted effort for a nearest neighbor search. In order to alleviate this effect, spatial index structures that use hyperspheres as node regions [55] have been proposed for use in nearest neighbor applications for higher dimensions. However, since this can lead to a much higher level of overlap between nodes than using hyperrectangles, a compromise is to use shapes formed by intersections of hyperspheres and hyperrectangles [35], essentially smoothing out the corners of the hyperrectangles.

In Section 4.6 we pointed out that the objects on the priority queue are contained in the leaf nodes intersected by the boundary of the search region (and similarly for the nodes on the priority queue). As the number of dimensions grows, the ratio of the number leaf nodes intersected by the boundary of the search region to the number of leaf nodes intersected by the interior of the search region tends to grow. Thus the size of the priority queue also tends to grow with the number of dimensions. For uniformly-distributed points spread evenly among the leaf nodes, where each leaf node covers about the same amount of space, it can be shown that this ratio grows exponentially with the number of dimensions. This is true even if both the search region and leaf node regions are hypercubes (i.e., if we use the Chessboard metric $L_\infty$). Of course, this is only of major significance when the number of desired neighbors is large, since the volume of the search region depends on the number of neighbors.

Some of the problems arising from operating in high-dimensional spaces can be alleviated by relaxing the requirement that the nearest neighbors be computed exactly. Our goal is to report neighbors as quickly as possible. In the incremental nearest neighbor algorithm, when an object $o$ is slightly farther from the query object $q$ than a node $n$, the algorithm must process $n$ before reporting $o$. In a high-dimensional space, as we have seen, this may cause a lot of extra work. Instead, what we can do is to report $o$ as the next nearest neighbor if its distance from $q$ is not "much" larger than that of $n$. In particular, suppose $o$ is the object on the priority queue closest to $q$, and $n$ is the node on the queue closest to $q$. We propose to report $o$ as the next (approximate) nearest neighbor if $d_o(q,o) \leq (1+\epsilon)d_n(q,n)$, where $\epsilon$ is some nonnegative constant. This leads to a definition of approximate nearest neighbor that conforms to that in [4]: if $r$ is the distance of the $k^{th}$ nearest neighbor, then the distances of the objects returned by an approximate $k$-nearest neighbor search must be no larger than $(1+\epsilon)r$. Obviously, for $\epsilon = 0$ we get the exact result, and the larger $\epsilon$ is, the less exact the result is. The only change required to the incremental nearest neighbor algorithm to make it approximate in this sense is in the key used for nodes on the priority queue. Specifically, for a node $n$ we use $(1+\epsilon)d_n(q,n)$ as a key instead of $d_n(q,n)$. In [4][5], it was found that a significant reduction in node accesses results from finding the $k$ approximate nearest neighbors as opposed to the $k$ exact nearest neighbors. Moreover, with relatively high probability, the result is the same in the exact and approximate cases. For example, for approximate nearest neighbor search in 16 dimensions using $\epsilon = 3$ (meaning that a 300% relative error in distance is allowed), it was found [4] that the speedup in execution time was on the order of 10 to 50 over exact nearest neighbor search, while the average relative error was only 10% and the true nearest neighbor was found almost half the time.

## 8 Concluding Remarks

A detailed comparison of two approaches to browsing spatial objects in an R-tree on the basis of their distances from an arbitrary spatial query object was presented. It was shown that an incremental algorithm (INN) significantly outperforms (in terms of execution time, R-tree node disk I/O, and object distance calculations) a solution based on a $k$-nearest neighbor algorithm ($k$-NN). This was true even when the $k$-NN approach was optimized for this application by carefully choosing the increments for $k$ and using previous search re-

[5]The algorithm described in [4] is not incremental, but it accesses the same set of nodes as the incremental nearest neighbor algorithm modified as described above

sults for pruning when the $k$-NN algorithm must be re-invoked. The incremental approach was also found to have superior performance when applied to the problem of computing the $k$ nearest neighbors of a given query object. Our experiments confirm that the INN algorithm achieves a higher level of pruning than the $k$-NN algorithm. This is important as it reduces the amount of R-tree node disk I/O as well as the number of distance calculations, which, when combined, account for a major portion of the execution time. Moreover, as the data sets became larger, the superiority of INN algorithm became more pronounced.

The experimental results were in reasonably close agreement with our rudimentary analysis of the INN algorithm, which predicts that the number of node accesses is $O(k + \sqrt{k} + \log N)$, where $k$ is the number of neighbors and $N$ the size of the data set. The superior performance of our algorithm in the experimental study was perhaps not surprising, as we prove informally that, at any step in its execution, the incremental nearest neighbor algorithm is optimal with respect to the spatial data structure that is employed. From a practical standpoint, this means that a minimum number of nodes is visited in order to report each object. In other words, upon reporting the $k^{th}$ neighbor $o_k$ of the query object $q$, the algorithm has only accessed nodes that lie within a distance of $d(q, o_k)$ of $q$. Our adaptation of the algorithm to the R-tree has the added benefit that a minimum number of objects is accessed, i.e., only objects whose minimum bounding rectangles lie within a distance of $d(q, o_k)$ of $q$.

In the experiments reported in Section 6, we used an R-tree variant in which the spatial objects were stored directly in the leaf nodes of the R-tree. This is not always practical, especially for complex and variable-size objects such as polygons. The other alternative is to store the objects in an external file, in which case the leaf nodes store the bounding boxes of the spatial objects and pointers to the objects. We performed additional experiments where the maps used in Section 6 were stored in such an R-tree, and we used the INN variant given in Figure 4[6]. These experiments revealed an even larger advantage for the incremental nearest neighbor algorithm over the $k$-nearest neighbor algorithm (typically over 50%). This is primarily because the INN algorithm accessed many fewer data objects (for the purpose of calculating their distances from the query object) than the $k$-NN algorithm. The $k$-NN algorithm typically accessed 4-6 times as many objects as the INN algorithm for low values of $k$, and up to twice as many for values of $k$ as high as 5% of the map size. Reducing the number of object accesses and object distance calculations when using the incremental algorithm has an even greater effect in terms of reducing the execution time for more complex spatial objects (e.g., polygons).

In a worst-case scenario, all the leaf nodes in the spatial data structure must be accessed (see Figure 4.8 and the discussion in Section 4.8). In contrast to the incremental algorithm presented in Figure 3, the variant presented in Figure 4 for the R-tree implementation where the spatial objects are stored external to the R-tree alleviates the worst case described above by making use of bounding rectangles in leaf nodes, thereby enabling it to avoid accessing many data objects from disk[7]. In particular, in the original version of the algorithm, the spatial index was not assumed to have bounding rectangles, which meant that for this worst case all data objects had to be accessed from disk in order to measure their distances from the query object. The use of bounding rectangles stored in the tree leads to a considerably more efficient (and conceptually different) incremental algorithm for R-trees in that the bounding boxes can be used as pruning devices to reduce disk I/O for accessing spatial descriptions of objects.

Future work involves comparing the behavior of the incremental nearest neighbor algorithm on different spatial data structures such as PMR quadtrees, R-trees, and R$^+$-trees, as well as adapting the algorithm to other classes of index structures, such as distance-based indexes [11, 14, 52]. We also wish to investigate

[6]We decided to report only the results of experiments where the spatial objects are stored in the leaf nodes rather than external to the R-tree. This was done, in part, because the organization of the external object storage has a large effect on the performance, and thus introduces an extra variable into the comparison of the algorithms.

[7]Recall from footnote 6 that we decided to report only the experiments in which the spatial objects are stored in the leaf nodes rather than external to the R-tree.

further the use of the algorithm with very large data sets and in high-dimensional spaces, where the priority queue may have to be stored on disk.

## References

[1] P. M. Aoki. Generalizing "search" in generalized search trees. In *Proceedings of the 14th International Conference on Data Engineering*, pages 380–389, Orlando, FL, Feb 1998.

[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, SC, August 1992.

[3] W. G. Aref and H. Samet. Estimating selectivity factors of spatial operations. In *Optimization in Databases — Fifth International Workshop on Foundations of Models and Languages for Data and Objects*, pages 31–40, Aigen, Austria, September 1993. (Also Technical Report: Informatik-Bericht 93/9, Technische Univerität Clausthal, Clausthal–Zellerfeld.)

[4] 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. Revised version: `http://www.cs.umd.edu/~mount/`.

[5] L. Becker and R. H. Güting. Rule-based optimization and query processing in an extensible geometric database system. *ACM Transactions on Database Systems*, 17(2):247–303, June 1992.

[6] 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.

[7] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[8] 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.

[9] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 28–39, Mumbai, India, September 1996.

[10] M. Bern. Approximate closest-point queries in high dimensions. *Information Processing Letters*, 45(2):95–99, February 1993.

[11] S. Brin. Near neighbor search in large metric space. In U. Dayal, P. M. D. Gray, and S. Nishio, editors, *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 574–584, Zurich, Switzerland, September 1995.

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

[13] W. A. Burkhard and R. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, April 1973.

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

[15] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[16] C. M. Eastman and M. Zemankova. Partially specified nearest neighbor searches using k-d–trees. *Information Processing Letters*, 15(2):53–56, September 1982.

[17] C. Esperança and H. Samet. Orthogonal polygons as bounding structures in filter-refine query processing strategies. In M. Scholl and A. Voisard, editors, *Advances in Spatial Databases — Fifth International Symposium*, pages 197–220, Berlin, Germany, July 1997. (Also Springer-Verlag Lecture Notes in Computer Science 1262.)

[18] C. Faloutsos and K. Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of the ACM SIGMOD Conference*, pages 163–174, San Jose, CA, May 1995.

[19] A. U. Frank and R. Barrera. The Fieldtree: A data structure for geographic information systems. In A. Buchmann, O. Günther, T. R. Smith, and Y. F. Wang, editors, *Design and Implementation of Large Spatial Databases — First Symposium*, pages 29–44, Santa Barbara, CA, July 1989. (Also Springer-Verlag Lecture Notes in Computer Science 409.)

[20] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[21] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

[22] 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.

[23] O. Günther and H. Noltemeier. Spatial database indices for large extended objects. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 520–526, Kobe, Japan, April 1991.

[24] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference*, pages 47–57, Boston, MA, June 1984.

[25] J. Hafner, H.S. Sawhney, W. Equitz, M. Flickner, and W. Niblack. Efficient color histogram indexing for quadratic form distance functions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(7):729–736, July 1995.

[26] A. Henrich. A distance-scan algorithm for spatial access structures. In *Proceedings of the Second ACM Workshop on Geographic Information Systems*, pages 136–143, Gaithersburg, MD, December 1994.

[27] A. Henrich. The LSD$^h$-tree: An access structure for feature vectors. In *Proceedings of the 14th IEEE International Conference on Data Engineering*, pages 362–369, Orlando, FL, February 1998.

[28] A. Henrich, H. W. Six, and P. Widmayer. The LSD tree: Spatial access to multidimensional point and non-point data. In P. M. G. Apers and G. Wiederhold, editors, *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 45–53, Amsterdam, The Netherlands, August 1989.

[29] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In M. J. Egenhofer and J. R. Herring, editors, *Advances in Spatial Databases — Fourth International Symposium*, pages 83–95, Portland, ME, August 1995. (Also Springer-Verlag Lecture Notes in Computer Science 951.)

[30] 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 — Second Symposium*, pages 237–256, Zurich, Switzerland, August 1991. (Also Springer-Verlag Lecture Notes in Computer Science 525.)

[31] I. Kamel and C. Faloutsos. On packing R-trees. In *Proceedings of the Second International Conference on Information and Knowledge Management*, pages 490–499, Washington, DC, November 1993.

[32] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 500–509, Santiago, Chile, September 1994.

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

[34] K. V. R. Kanth, D. Agrawal, and A. Singh. Dimensionality reduction for similarity searching in dynamic databases. In *Proceedings of the ACM SIGMOD Conference*, pages 237–248, Seattle, WA, June 1998.

[35] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In J. Peckham, editor, *Proceedings of the ACM SIGMOD Conference*, pages 369–380, Tucson, AZ, May 1997.

[36] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast nearest neighbor search in medical image databases. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 215–226, Mumbai, India, September 1996.

[37] H.-P. Kriegel, T. Schmidt, and T. Seidl. 3D similarity search by shape approximation. In M. Scholl and A. Voisard, editors, *Advances in Spatial Databases — Fifth International Symposium*, pages 11–28, Berlin, Germany, July 1997. (Also Springer-Verlag Lecture Notes in Computer Science 1262.)

[38] M. Lindenbaum and H. Samet. A probabilistic analysis of trie-based sorting of large collections of line segments. Computer Science Department TR-3455, University of Maryland, College Park, MD, April 1995.

[39] D. Lomet and B. Salzberg. A robust multi-attribute search structure. In *Proceedings of the Fifth IEEE International Conference on Data Engineering*, pages 296–304, Los Angeles, CA, February 1989.

[40] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proceedings of the ACM SIGMOD Conference*, pages 28–36, Chicago, IL, June 1988.

[41] O. J. Murphy and S. M. Selkow. The efficiency of using *k-d*–trees for finding nearest neighbors in discrete space. *Information Processing Letters*, 23(4):215–218, November 1986.

[42] R. C. Nelson and H. Samet. A consistent hierarchical representation for vector data. *Computer Graphics*, 20(4):197–206, August 1986. (Also *Proceedings of the SIGGRAPH'86 Conference*, Dallas, TX, August 1986.)

[43] Bureau of the Census. *Tiger/Line precensus files*. Washington, DC, 1989.

[44] J. T. Robinson. The *k–d–b*–tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD Conference*, pages 10–18, Ann Arbor, MI, April 1981.

[45] 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.

[46] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proceedings of the ACM SIGMOD Conference*, pages 17–31, Austin, TX, May 1985.

[47] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[48] T. Seidl and H.-P. Kriegel. Optimal multi-step $k$-nearest neighbor search. In *Proceedings of the ACM SIGMOD Conference*, pages 154–165, Seattle, WA, June 1998.

[49] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD Conference*, pages 23–34, Boston, MA, June 1979.

[50] T. Sellis, N. Roussopoulos, and C. Faloutsos. The $R^+$–tree: a dynamic index for multi–dimensional objects. In P. M. Stocker and W. Kent, editors, *Proceedings of the 13th International Conference on Very Large Databases*, pages 71–79, Brighton, England, September 1987. (Also Computer Science Department, University of Maryland, College Park, MD, TR–1795.)

[51] R. F. Sproull. Refinements to nearest-neighbor searching in *k*-dimensional trees. *Algorithmica*, 6(4):579–589, 1991.

[52] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, November 1991.

[53] T. L. Wang and D. Shasha. Query processing for distance metrics. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *Proceedings of the 16th International Conference on Very Large Databases*, pages 602–613, Brisbane, Australia, August 1990.

[54] D. A. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Computing Laboratory, University of California, San Diego, CA, 1996. http://vision.ucsd.edu/papers/simret.

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