CAR-TR-956 IRI-97-12715
CS-TR-4199 November 2000

Incremental Similarity Search in Multimedia Databases

Gísli R. Hjaltason and Hanan Samet

Computer Science Department
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742-3275
grh@cs.umd.edu and hjs@cs.umd.edu

Abstract

Similarity search is a very important operation in multimedia databases and other database applications involving complex objects, and involves finding objects in a data set S similar to a query object q, based on some distance measure d, usually a distance metric. Existing methods for handling similarity search in this setting fall into one of two classes. The first is based on mapping to a low-dimensional vector space (making use of data structures such as the R-tree), while the second directly indexes the objects based on distances (making use of data structures such as the M-tree). We introduce a general framework for performing search based on distances, and present an incremental nearest neighbor algorithm that operates on an arbitrary "search hierarchy". We show how this framework can be applied in both classes of similarity search methods, by defining a suitable search hierarchy for a number of different indexing structures. Armed with an appropriate search hierarchy, our algorithm thus performs incremental similarity search, wherein the result objects are reported one by one in order of similarity to a query object, with as little effort as possible expended to produce each new result object. This is especially important in interactive database applications, as it makes it possible to display partial query results early. The incremental aspect also provides significant benefits in situations when the number of desired neighbors is unknown in advance. Furthermore, our algorithm is at least as efficient as existing k-nearest neighbor algorithms, in terms of the number of distance computations and index node accesses. In fact, provided that the search hierarchy is properly defined, our algorithm can be shown to be optimal in the sense of performing as few distance computations and node accesses as possible, given the available index structure. An experimental study confirms our reasoning, and suggests that the overhead due to the incremental aspect is modest, especially if distance computations are expensive and/or the indexing structure or data objects are stored on disk.

This work was supported in part by the National Science Foundation under Grant IRI-97-12715.

1 Introduction

Multimedia data are becoming increasingly important in modern database applications. Examples of such data include images, video, and text documents, and even such exotic data as protein and DNA sequences. One of the most common types of queries on multimedia data is similarity searching, also termed *content-based* or *similarity* retrieval. This commonly involves finding the nearest neighbor to a sample object, based on some similarity measure. Usually, what is desired is to retrieve several of the nearest neighbors. In some applications, the number of desired neighbors is fixed in advance, in which case we can use *k*-nearest neighbor algorithms. In other applications, the number of desired neighbors is unknown in advance. This is where incremental nearest neighbor algorithms prove useful. By incremental, we mean that such an algorithm computes the neighbors one by one, attempting to report the next neighbor with as little effort as possible. An important area where incremental nearest neighbor algorithms are beneficial is in computing complex queries where one of the conditions is a "nearest" predicate. In this situation, we do not know how many neighbors must be retrieved before one is found that satisfies all the conditions. Another advantage to incremental solutions is that in interactive user interfaces they make it possible to quickly present some answer to the user, before the entire query result has been computed.

A common feature of most kinds of multimedia data is that evaluating the similarity between two objects is complicated, and thus time-consuming. Usually, some sort of distance metric is defined that indicates the degree of similarity between two objects. Thus, we can view the objects as residing in a metric space (S,d), where S is the finite set of objects and d is the distance metric. When evaluating proximity queries, we can only rely on properties of distance metrics (i.e., non-negativity, symmetry, and the triangle inequality). Contrast this with data residing in a coordinate space, where more information is available (the locations and extents of the objects). Multimedia data objects are actually frequently represented as points in a vector space (often termed *feature vectors*). For example, color and shape histograms are commonly used to characterize images [23, 30]. However, indexing and search methods for coordinate spaces usually work well only for low-dimensional spaces. Even indexing methods specifically designed to handle a higher number of dimensions (e.g., the X-tree [6], LSD$^{\text{h}}$-tree [33], and hybrid tree [13]) typically end up accessing nearly all index pages for any non-trivial query when the number of dimensions exceeds 20 and often even when it is lower. In contrast, feature vectors for multimedia objects are often of much higher dimensionality (e.g., color histograms often lead to 64, or even as high as 256-dimensional vectors).

Two strategies have been proposed for handling proximity queries in metric spaces. The first is to work directly within the metric space, often by building hierarchical *distance-based* index structures [9, 11, 19, 66]. Nearest neighbor and *k*-nearest neighbor algorithms have been proposed for many of these structures. The second strategy maps the database objects into a low- to medium-dimensional vector space and then makes use of efficient spatial indexing methods available there, such as the R-tree [29]. The index on the mapped objects serves as a filter in proximity queries, but the actual objects must be consulted to compute the final answer; thus, algorithms based on this *filter and refine* [56] approach are sometimes said to be *multistep*. A *k*-nearest neighbor algorithm for such a scenario was proposed in [41], and later improved in [60].

In [35, 36] we proposed a general incremental nearest neighbor algorithm that works for virtually all hierarchical spatial index structures. A similar algorithm was presented in [32], targeted for the LSD tree. In this paper, we show how our earlier algorithm can be adapted to handle incremental nearest neighbor search in metric spaces, whether the data set is represented in a distance-based index or has been mapped into a vector space (and is represented in a spatial index). A key step is a formulation of the incremental nearest neighbor algorithm in terms of an arbitrary *search hierarchy*, which is derived from the particular data structure used to organize a data set. The search hierarchy represents the decomposition of a

proximity-related search problem on the data set. In most cases, the structure of the search hierarchy is identical to (or easily derived from) the hierarchical nature of the data structure[1], so the main challenge is to determine lower bounds on distances between a query object and the objects in subtrees of the hierarchy. The incremental nearest neighbor algorithm is shown to be correct when this lower-bound criterion is fulfilled. Also, we describe various useful extensions of the algorithm, such as finding the farthest neighbor first, and we show how better performance can be gained when complete accuracy is not crucial. We present search hierarchies both for the mapping-based approach and for various distance-based indexes, and prove that correctness of the algorithm is ensured. During this discussion, we survey a number of different mapping methods and distance-based indexing methods. When compared to existing nearest neighbor and k-nearest neighbor algorithms for these methods, our algorithm has the distinct advantage of being incremental. This gives it much better performance for *distance browsing* queries, wherein we browse through a database based on distance and may terminate the browsing at any time — that is, the number of desired objects is unknown in advance. Moreover, even for a fixed number of neighbors, our algorithm can be shown to be at least as efficient as existing k-nearest neighbor algorithms, in terms of the number of distance computations (and I/O operations, if needed). Furthermore, the overhead due to the incremental aspect of the algorithm is usually relatively modest, and is usually overwhelmed by savings due to the better search pruning exhibited by our algorithm. An experimental study confirms these conclusions, where we perform experiments on various data sets using both the mapping-based approach (with the R*-tree spatial index) and a distance-based index (with the M-tree).

The rest of the paper is organized as follows. In Section 2 we present a generalized version of our incremental nearest neighbor algorithm. In Section 3 we give an overview of different types of mapping methods, and show how the general incremental nearest neighbor algorithm can be applied to the mapping-based approach. In Section 4 we describe the main principles behind most distance-based indexes, and show how to apply our algorithm to a number of representative indexing methods. In Section 5, we compare our algorithm to existing k-nearest neighbor algorithms, in terms of the level of pruning that is achieved. In Section 6 we report the results of our experiments, while conclusions are drawn in Section 7.

2 Generalized Incremental Nearest Neighbor Algorithm

In this paper, we will show how to perform incremental nearest neighbor (abbreviated INN) search based on a variety of different data structures for similarity search. The nature of such search is to report the objects in a data set $S \subset \mathbb{U}$, one by one, in order of distance from a query object $q \in \mathbb{U}$ based on a distance function d, where $\mathbb{U}$ is the domain (usually infinite) from which the objects are drawn. In all cases, the same basic algorithm is adapted to the specific nature of each data structure. Thus, in this section we describe a generalized version of the incremental nearest neighbor algorithm of [35, 36], that encompasses all the specific instances presented in later sections. In order to make the discussion more concrete, we use the R-tree, a commonly used spatial data structure, as an example data structure.

The rest of this section is organized as follows: In Section 2.1, we briefly describe the R-tree. In Section 2.2, we introduce the basic framework for performing search, in the form of a *search hierarchy*. In Section 2.3, we present a general incremental nearest neighbor algorithm, based on the abstract concept of a search hierarchy. This algorithm can be adapted to virtually any data structure, by a suitable definition of a search hierarchy, and we illustrate this in the case of the R-tree. In Section 2.4, we show the algorithm can be extended in a variety of ways, including setting limits on the distances of the result objects or on the number of result objects, and reporting the objects in reverse order of distance (i.e., finding the *farthest* neighbor first). Furthermore, in situations where complete accuracy is not critical, we show how the performance of the algorithm can be improved at the risk of having the objects reported

[1] However, as we note in Section 4.6, a search hierarchy can be formalized even when the data structure is not hierarchical.

somewhat out of order.

2.1 R-Tree

The R-tree [29] (see Figure 1) is an object hierarchy in the form of a balanced tree inspired by the B^+-tree [21]. 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. If the geometric description of the objects is simple (e.g., for points or line segments) it is also possible to store the objects directly in the leaf nodes, instead of their bounding rectangles. 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).

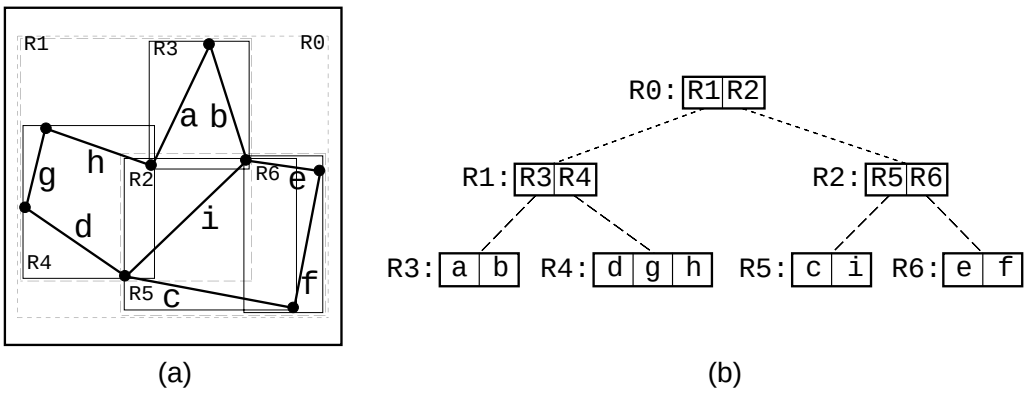

Figure 1: An R-tree index for a set of nine line segments. (a) Spatial rendering of the line segments and bounding rectangles, and (b) a tree access structure for (a). The bounding rectangles for the individual line segments are omitted from (a) in the interest of clarity.

2.2 Search Hierarchy

The incremental nearest neighbor algorithm is applicable whenever the search space can be represented in a hierarchical manner, provided that certain conditions hold. In this section, we define what we mean by "search hierarchy" and use the R-tree as a specific example.

The algorithm operates on a finite set $S \subset \mathbb{U}$ of objects, a query object $q \in \mathbb{U}$, and a data structure T that organizes the set S or provides information about it. The search hierarchy is composed of elements e_t of several different types t, with $t = 0, \ldots, t_{\max}$. Figure 2 depicts the search hierarchy for a set of 14 objects, A_0 through N_0, where there are three types of elements. As depicted in the figure, the search hierarchy forms a tree, with the objects as leaves. Each element represents a subset of S, with an element e_0 of type 0 representing a single object in S. An element e_t of type t can give rise to one or more "child" elements of types 0 through $t_{\max}$. Thus, the search problem for e_t is decomposed into several smaller subproblems. Often, all child elements are of the same type, but this is not a requirement. In this presentation, we assume that an element has only one "parent" in the search hierarchy and that each object is represented only once in the hierarchy, but the algorithm can be adapted to handle multiple parents and duplicate object instances as well.

At this point, it may seem superfluous to use different types of elements, when they all merely represent subsets of S. However, for any particular search problem, the various element types represent differ-

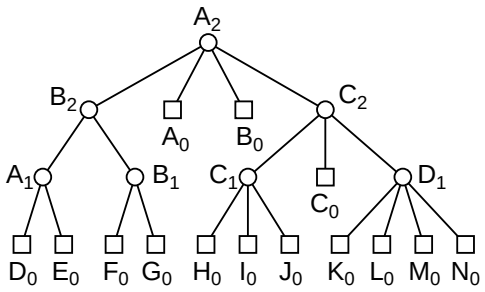

Figure 2: A sample search hierarchy for objects A_0 though N_0.

ent kinds of entities, and therefore have different pieces of information attached to them. Furthermore, each specific search hierarchy will have a different definition of what types of elements can be produced from each element type. In the example in Figure 2, for instance, elements of type 2 produce elements of types 2, 1, or 0, while elements of type 1 produce only elements of type 0. Elements of each type t have an associated distance function $d_t(q,e_t)$ for measuring the distance from a query object q to the elements of that type. Naturally, computation of $d_t(q,e_t)$ is based only on the specific information attached to the parent element of e_t (since the computation occurs when the parent element is processed). Furthermore, for the algorithm to be practical, this computation must be substantially less expensive than the total cost of computing $d_0(q,e_0)$ for all the objects e_0 represented by e_t. As we shall see, it is sufficient for correctness that $d_t(q,e_t) \leq d_0(q,e_0)$ for any object e_0 in the subset represented by e_t. Some of the variants of the algorithm (e.g., farthest neighbors; see Section 2.4) make use of another set of distance functions, $\hat{d}_t(q,e_t)$, that bound from above the distances from q of the objects in a subtree (of course, $\hat{d}_0 = d_0$, so $\hat{d}_0$ is not really needed). In other words, $\hat{d}_t(q,e_t) \geq d_0(q,e_0)$ for any object e_0 in the subset represented by e_t.

In most cases, the search hierarchy arises naturally from the hierarchical structure of T. In the search hierarchy for the R-tree, for instance, we have three types of elements. Elements of type 0 are the spatial objects themselves, elements of type 1 are minimum bounding rectangles for single objects, while elements of type 2 represent R-tree nodes. The reason for distinguishing between the spatial objects and their bounding rectangles will become clear later. The distance functions are typically based on a distance metric $d_p(p_1,p_2)$ defined on points in the space, such as the Euclidean metric. Thus, for arbitrary spatial objects o_1 and o_2, we define $d(o_1,o_2) = \min_{p_1 \in o_1, p_2 \in o_2}\{d_p(p_1,p_2)\}$[2]. This definition serves as a basis for the distance functions for all three types of elements, the only difference being the types of the arguments. Given an object o and any rectangle r that bounds o (i.e., r can be the minimum bounding rectangle of o or the rectangle associated with any ancestor of the leaf node containing o), this definition guarantees that $d(q,r) \leq d(q,o)$, thus ensuring correctness. In fact, according to this definition, the distance functions obey an even stricter condition, namely that $d_t(q,e_t) \leq d_{t'}(q,e_{t'})$ for any element $e_{t'}$ that is a descendant of e_t in the search hierarchy.

2.3 Algorithm

The key to our incremental nearest neighbor algorithm [36] is that it traverses the search hierarchy in a "best-first" manner, like the classic A*-algorithm [59], instead of the more traditional depth-first or breadth-first traversals. In other words, at any step of the algorithm, the algorithm visits the element with the smallest distance from the query object among all unvisited elements in the search hierarchy (or,

[2] In other words, if o_1 and o_2 have at least a point in common, then $d(o_1,o_2) = 0$. Otherwise, $d(o_1,o_2)$ equals the smallest distance between the boundaries of o_1 and o_2.

more accurately, all unvisited elements whose parents have been visited). This is done by maintaining a global list of elements, organized by their distance from the query object. A data structure that supports the necessary operations (i.e., insert and delete minimum) is an instance of an abstract data type termed a *priority queue*. In order to break ties among elements having the same distance, we give priority to elements with lower type numbers, and among elements of the same type, priority is given to elements deeper in the search hierarchy. This allows the algorithm to report neighbors as quickly as possible.

The general incremental nearest neighbor algorithm is shown in Figure 3, where q is the query object, S the set of objects, and T a data structure that organizes S. Since we make no assumptions about the objects or the data structure, the algorithm is presented in terms of the search hierarchy induced by them. The algorithm starts off by initializing the priority queue with the root of the search space (lines 1–3). In the main loop, the element e_t closest to q is taken off the queue. If it is an object, we report it as the next nearest object (line 7). Otherwise, the child elements of e_t in the search hierarchy are inserted into the priority queue (line 10). Letting o_k be the k^{th} object reported by the algorithm, observe that the non-object elements on the priority queue (i.e., elements e_t with $t > 0$) essentially represent portions of the search hierarchy that did not have to be explored to establish o_k as the k^{th} nearest neighbor. Thus, if o_k is the last neighbor requested, those portions are said to be *pruned* by the algorithm, and all distance computations for objects descended from the non-object elements on the priority queue are avoided. As we show in Section 2.5, the algorithm achieves maximal possible pruning with respect to the given search hierarchy. Of course, if more neighbors are requested, some of those elements may be processed later. Nevertheless, the fact that their processing is deferred means that o_k is reported as quickly as possible.

```
INCNEAREST(q, S, T)
 1  Queue ← NEWPRIORITYQUEUE()
 2  e_t ← root of the search hierarchy induced by q, S and T
 3  ENQUEUE(Queue, e_t, 0)
 4  while not ISEMPTY(Queue) do
 5    e_t ← DEQUEUE(Queue)
 6    if t = 0 then /* e_t is an object */
 7      Report e_t as the next nearest object
 8    else
 9      for each child element e_t' of e_t do
10        ENQUEUE(Queue, e_t', d_t'(q, e_t'))
11      enddo
12    endif
13  enddo
```

Figure 3: Generalized incremental nearest neighbor algorithm.

Interestingly, in the spatial domain, the algorithm can be viewed as having an initial phase that corresponds to an intersect query (or, if q is a point, a point-location query). In particular, the algorithm first processes all elements e in the search hierarchy with $d(q,e) = 0$, namely those elements that q intersects (fully or partially). Thus, at the conclusion of this initial phase, all leaf nodes in the spatial index intersecting q have been processed and their contents inserted into the priority queue, and all objects intersecting q have been reported. A similar observation also holds for many of the search hierarchies for similarity search that we present in the remaining sections.

To get a better idea of how the algorithm functions, let us examine how it behaves when applied to the R-tree for spatial data objects. Initially, the root of the R-tree (representing all the data objects) is placed on the priority queue. At subsequent steps, the element at the front 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 4). 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* (for query objects other than points, the search region will have a more complex shape). Each time the circle hits the boundary of a node region, the contents of that node are put on the queue (and similarly for the bounding rectangle of an object), and each time the circle hits an object, we have found the object next nearest to q. We can characterize the contents of the priority queue based on their relationships with the search region. The elements still on the queue are outside the search region (but they all have parents that intersect the search region), while all elements that have been dequeued lie at least partially inside. Also, nodes whose bounding rectangles are fully contained in the search region will have had their entire subtrees already taken off the priority queue. Therefore, all the queue elements are contained in nodes whose bounding rectangles intersect the boundary of the search region. Similarly, for all objects on the priority queue, their corresponding bounding rectangles intersect the boundary of the search region.

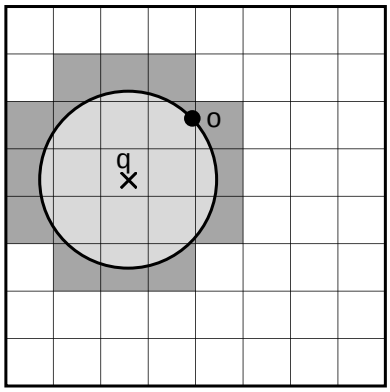

Figure 4: The circle around query object q depicts the search region after reporting o as next nearest object. For simplicity, the leaf nodes are represented by a grid; in most spatial indexes, the shapes of the leaf nodes are more irregular. Only the shaded leaf nodes are accessed by the incremental nearest neighbor algorithm. The region with darker shading is where we find the objects in the priority queue.

It should be clear now why we choose to distinguish between the spatial objects and their bounding rectangles in the R-tree implementation of the incremental nearest neighbor algorithm. Usually the precise geometry of the objects is stored in a separate file on disk, whereas the bounding rectangles are stored directly in the leaf nodes of the R-tree. Thus, computing the distance from the query object to the bounding rectangle of an object is much less expensive than computing the distance to the object itself. When a leaf node in the R-tree is visited by the algorithm, the bounding rectangles for the objects in the leaf node are inserted on the priority queue. The actual distance from the query object to a data object is computed only when its bounding rectangle reaches the front of the priority queue.

In fact, at times, we can report an object as a nearest neighbor without ever having to compute its distance from the query object. Such a situation can arise when using an R-tree, where we recall that elements of type 1 represent minimum bounding rectangles of objects — that is, each element e_1 represents only one object. In particular, when processing such an element e_1, we can sometimes use the distance $\hat{d}_1(q,e_1)$ to immediately report the object o represented by e_1, without computing the actual distance $d(q,o)$. This can be done provided that $\hat{d}_1(q,e_1) \leq D$, where D is the distance of the element at the front of the queue (i.e., after e_1 has been removed), since we then know that $d(q,o)$ is also no larger than D. This principle is generally applicable any time that one (or more) of the element types $t > 0$ represents

single objects and the upper-bound distance function $\hat{d}_t$ can be defined.

Deferring the computation of the actual distances usually leads to many fewer object distance computations, except when a large number of neighbors must be reported (i.e., k neighbors are required, where k is a significant fraction of N, the size of the data set S). Hence, when the algorithm is terminated (i.e., enough neighbors have been obtained), any bounding rectangles present on the priority queue represent distance computations that were avoided by their use. Of course, these benefits come at the cost of more priority queue operations. Similar considerations apply to other search structures.

As a practical matter, we have observed that a careful implementation of the priority queue is often crucial for achieving good performance; how important this is depends on the relative cost of priority queue operations and other operations in the algorithm, such as distance computations. Typically, the algorithm performs more enqueue operations than dequeue operations, and this should guide the selection of what priority queue algorithm to use. Also, we have found that careful attention to memory allocation is an important aspect. In particular, rather than relying on dynamic allocation for each individual queue element, the implementation should manage the allocation from a pool of large memory blocks (this is made easier if each type of queue element occupies the same amount of space; alternatively, separate pools can be kept for each type). This way, all the elements remaining on the queue when the algorithm is terminated can be deallocated inexpensively, since the pool consists only of a few blocks. In our experience [36], the size of the priority queue usually remains very modest compared to the size of the data set, so it is unlikely that an external (i.e., disk-based) implementation is needed.

2.4 Algorithm Extensions

The algorithm is easily adapted to take advantage of imposed distance bounds (as in a range query) as well as maximum result size (as in a k-nearest neighbor query). In particular, given a maximum distance bound $D_{\max}$, we only enqueue elements having distances from q less than or equal to $D_{\max}$. A minimum distance bound $D_{\min}$ can be exploited in a similar way, but doing so requires the additional distance functions $\hat{d}_t(q, e_t)$ for each element type that provide an upper bound on the distances from q of the objects in the subtree represented by e_t. Taking advantage of such distance bounds reduces the size of the priority queue (since fewer items will be enqueued) and, consequently, the average cost of priority queue operations. However, the number of distance computations or the number of search hierarchy elements visited is not decreased when using $D_{\max}$, and possibly not even when using $D_{\min}$, depending on its value (in fact, utilizing $D_{\min}$ may even lead to worse performance as it requires computing the $\hat{d}$ distances).

The simplest way of exploiting a maximum result size, say k, is to simply terminate the algorithm once k neighbors have been determined. Alternatively, the algorithm can be modified in such a way that the distance of the k^{th} candidate nearest neighbor is used to reduce the number of priority queue operations. However, such a modification carries the cost of additional complexity in the algorithm. In particular, we must use two priority queues, one for the objects and the other for the remaining types of elements, and we must be able to identify and remove the element in the object priority queue with the largest distance. In this way, we can keep track of o_k, the current candidate k^{th} nearest neighbor, whose distance is used to prune the search in the same way we used $D_{\max}$ above.

A useful extension of the algorithm is to finding the farthest neighbor of a query object, i.e., the object in S that is farthest from q. This simply requires replacing $d_t(q, e_t)$ as a key for any element e_t on the priority queue with $-\hat{d}_t(q, e_t)$ (or, alternatively, ordering the elements on the priority queue in decreasing order of key values, and using $\hat{d}_t(q, e_t)$ as a key value). Thus, once an object o has reached the front of the priority queue, we know that no unreported object has a greater distance from q. Observe that as in the discussion of the nearest neighbor algorithm in Section 2.3 (by reversing the roles of d and $\hat{d}$), at times, we can even report an object as a farthest neighbor without ever having to compute its distance from the query object.

In many applications, obtaining exact results is not critical. Therefore, users are willing to trade off accuracy for improved performance. This is another direction in which the algorithm can be easily extended. For approximate nearest neighbor search [2], a common criterion is that the distance between q and the resulting candidate nearest neighbor o' is within a factor of $1+\varepsilon$ of the distance to the actual nearest neighbor o, i.e., $d(q,o') \leq (1+\varepsilon)d(q,o)$. The incremental nearest neighbor algorithm can be made approximate in this sense by multiplying the key values for non-object elements on the priority queue by $1+\varepsilon$. In other words, for an element e_t, $t > 0$, we use $(1+\varepsilon)d_t(q,e_t)$ as a key. With this extension, for the object o'_k returned as the k^{th} nearest neighbor by the algorithm and the actual k^{th} nearest neighbor o_k, we have $d(q,o'_k) \leq (1+\varepsilon)d(q,o_k)$. However, notice that in this case, the result objects are not necessarily reported by the algorithm in strictly increasing order of distance from q. Other methods for performing approximate nearest neighbor search can often be integrated into the algorithm by similarly modifying the distances used in the priority queue (e.g., the method of Chen et al. [16], which involves shrinking the radii of bounding spheres when computing their distances). For approximate farthest neighbor search in high-dimensional point data, Duncan et al. [22] suggest the criterion that o' is an approximate farthest neighbor of q if $d(q,o') \geq d(q,o) - \varepsilon D$, where o is the actual farthest neighbor and D is the *diameter* of the point set (i.e., the distance between the two farthest points)[3]. This criterion can also be incorporated into our incremental algorithm in a straightforward manner.

The version of the incremental nearest neighbor algorithm presented in this paper is actually slightly less general than our earlier presentation [35, 36] in that duplicate instances of objects in the search hierarchy are not eliminated. Such duplicate detection is necessary for spatial indexing methods that represent each spatial object in all leaf nodes whose mutually non-intersecting regions intersect the object (e.g., the PMR quadtree [51] and the R^+-tree [61]). Adding this feature to the algorithm in Figure 3 is straightforward and only requires a few extra lines [36]. However, this is not needed for the present purposes, since none of the methods for similarity search presented below result in duplicate instances of objects in the search hierarchy.

2.5 Correctness and Optimality of the Algorithm

As alluded to in Section 2.2, the correctness of the algorithm depends on the distance functions used on the elements of the search hierarchy:

Lemma 1 *Let e_t be an arbitrary element in the search hierarchy. The following* correctness criterion *guarantees that the algorithm in Figure 3 reports neighbors in order of non-decreasing distance: for any object e_0 in the subtree represented by e_t, we have $d_t(q,e_t) \leq d_0(q,e_0)$.*

Proof We prove the lemma by contradiction. Assume that the condition holds for all elements of the search hierarchy and assume that e_0 and e'_0 represent two objects, with $d_0(q,e_0) < d_0(q,e'_0)$, that are reported out of order (i.e., e'_0 first). This implies that at some point, e'_0 is the element in the queue with the smallest distance from q, while an element e_t that represents some subset that contains e_0 is also on the queue. In other words, $d_t(q,e_t) \geq d(q,e'_0)$. However, since $d_t(q,e_t) \leq d_0(q,e_0) < d(q,e'_0)$, we have $d_t(q,e_t) < d(q,e'_0)$, contradicting the assumption that e'_0 is reported before e_0. Thus, the lemma holds. ■

Some of the variants of our algorithm outlined in Section 2.4 make use of the upper-bound distance functions $\hat{d}_t$. For these variants to be correct, these functions must indeed upper-bound the distances of

[3] The motivation for basing the criterion on an *absolute error bound* based on D rather than a *relative error bound* based on $d(q,o_k)$ is that the absolute error bound gives a tighter bound in the farthest neighbor case. For example, all points in S would tend to be the approximate farthest neighbor according to the relative error bound if the points in S were relatively close while q is very far from these points [22].

objects in a subtree — that is, for any object e_0 in the subset represented by e_t, we must have $\hat{d}_t(q,e_t) \geq d_0(q,e_0)$.

Visiting an element in the search hierarchy and computing distances can both be expensive operations. For example, for the R-tree version, visiting an R-tree node involves node I/O, and computing the distances of objects may also involve I/O (if stored outside the R-tree) in addition to potentially expensive geometric computation (e.g., for polygons). Thus, our goal is to visit as few elements in the search hierarchy as possible. This notion is captured in the notion of r-optimality (range optimality):

Definition 1 *Let o_k be the k^{th} nearest neighbor to a query object q reported by a nearest neighbor algorithm, which operates on a search hierarchy induced by a set of objects S, a data structure T, and the query object q, using distance functions d_t. The algorithm is r-optimal if for all visited elements e_t in the search hierarchy, $d_t(q,e_t) \leq d(q,o_k)$.*

According to this definition, a nearest neighbor algorithm is r-optimal if it visits the same search hierarchy elements as would a range query with query radius $d(q,o_k)$, implemented with a top-down traversal of the search hierarchy. The incremental nearest neighbor algorithm is r-optimal, as shown below:

Lemma 2 *If the distance functions satisfy the correctness criterion, the algorithm in Figure 3 is r-optimal.*

Proof Again, we proceed by contradiction. Let o_k be the k^{th} nearest neighbor to q, and assume that e_t is an element that was visited before o_k was reported, with $d_t(q,e_t) > d(q,o_k)$. This assumption means that when e_t was visited, some ancestor $e_{t'}$ of the element e_0 representing o_k in the search hierarchy existed in the priority queue, with $d_t(q,e_t) \leq d_{t'}(q,e_{t'})$. By the correctness criterion, we have $d_{t'}(q,e_{t'}) \leq d_0(q,e_0) = d(q,o_k)$ which implies that $d_t(q,e_t) > d_{t'}(q,e_{t'})$. This contradicts the initial assumption that e_t was visited before o_k was reported, so the lemma holds. ■

Underlying both lemmas is the premise that for any object e_0 that has not been reported, there is a representative e_t on the priority queue (i.e., e_t is an ancestor of e_0 in the search hierarchy). This is clearly true initially, since the root of the search hierarchy represents all the objects in S. The only actions of the algorithm are to remove objects from the priority queue and to replace search hierarchy elements with all their children. Thus, the algorithm never violates the premise.

Observe that the correctness criterion (i.e., $d_t(q,e_t) \leq d_0(q,e_0)$) does not actually guarantee that all search hierarchy elements are obtained from the priority queue in strict order of distance. In particular, the criterion does not prevent the possibility that we may obtain the elements e_t and $e_{t'}$ in succession, where $d_t(q,e_t) > d_{t'}(q,e_{t'})$ and neither element is an object[4]. The implication is that the algorithm has not necessarily visited all elements e_t with $d_t(q,e_t) \leq d(q,o_k)$ when the k^{th} neighbor o_k of q is reported. However, this does not change the fact that object elements are obtained from the priority queue in order of distance. Often, however, the distance functions satisfy the stricter criterion that $d_t(q,e_t) \leq d_{t'}(q,e_{t'})$ for any element $e_{t'}$ that is a descendant of e_t in the search hierarchy. This was the case for the distance functions defined for the R-tree in Section 2.2. This stricter criterion guarantees that elements are obtained from the priority queue in order of increasing distance from the query object, regardless of their type.

It is important to note that r-optimality does not imply that the incremental nearest neighbor algorithm for the given search hierarchy is necessarily optimal in some absolute sense (whether we consider only the number of distance computations required or the overall work). First, actual search performance is highly dependent on the index structure that the search hierarchy is based on. Performance can suffer

[4] In other words, $e_{t'}$ is a child of e_t that is closer to the query object than its parent is. Such a situation could arise, for example, in the vp-tree (see Section 4.2.1).

if the index structure poorly captures the inherent structure of the search space. Second, it is often possible to define more than one search hierarchy for a given index structure, each with different performance characteristics. In some cases, different choices of a search hierarchy even represent a tradeoff between the amount of work required to maintain the priority queue operations and to compute the various distance functions (e.g., see Section 4.6). Moreover, it is sometimes possible to tailor the algorithm more closely to the given index structure than is achieved with our general notion of search hierarchy. Finally, even for a specific search hierarchy, it is possible that a k-nearest neighbor algorithm is more efficient for any fixed value of k. Thus, some sacrifice in performance can be expected for the flexibility offered by the incremental aspect. However, when the number of desired neighbors is unknown prior to initiating the search, an incremental algorithm provides much better performance on the average than algorithms that require fixing the number of neighbors in advance (we observe this experimentally in Section 6).

3 Incremental Similarity Search with Mapping-Based Approaches

In mapping-based approaches, objects are mapped into multidimensional points (also known as *feature vectors*) in such a way that distances in the mapped space approximate the distances among the original objects. Formally, let $S \subset \mathbb{U}$ be a finite set of data objects taken from a universe $\mathbb{U}$, and let $d : \mathbb{U} \times \mathbb{U} \to \mathbb{R}^+$ be a distance function on $\mathbb{U}$. The key to such methods is a mapping $F : \mathbb{U} \to \mathbb{R}^m$ and an accompanying distance function $\delta : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}^+$. Typically, δ is some Minkowski metric L_p, the most common being the Euclidean (L_2), the Cityblock (L_1), and the Chessboard (L_∞) distance metrics. F is often termed an *embedding* of (S,d) into $(\mathbb{R}^m,\delta)$, and a *Euclidean embedding* when δ is the Euclidean metric.

By applying F to all the elements of S, the result of a query on S with a query object q can be approximated by also mapping the query object q and then applying a corresponding query to $F(S)$ with $F(q)$ as a query object. The advantage of doing so is that it allows replacing the expensive computations of $d(q,o)$, for $o \in S$, by the much less expensive computations of $\delta(q',o')$, for $q' = F(q)$ and $o' \in F(S)$. Most importantly, by using a spatial index to represent $F(S)$, many points o' in $F(S)$ can be pruned from the search without having to evaluate $\delta(q',o')$. As we shall see in Section 3.2, the property of F being *contractive* is important for such queries, i.e., that $\delta(F(o_1),F(o_2)) \leq d(o_1,o_2)$ for all $o_1,o_2 \in S$.

In principle, any spatial indexing method can be used to implement search with the mapping-based approach. However, when the mapping results in m, the dimensionality of the mapped space, being high, indexing methods specifically designed for high-dimensional spaces should be employed. Examples of such methods include hierarchical spatial indexes such as the X-tree [6], LSD$^{\text{h}}$-tree [33], and hybrid tree [13] (typically applicable up to 20–30 dimensions). These methods break down for very high dimensions in which case a linear scan is often faster. In such a case, attention has been focused on speeding up the linear scan process by reducing the size of the data that is examined during the scan (via the use of quantization), thereby, hopefully, pruning most irrelevant objects from further consideration. This is the basis of the VA-file method [72, 25], and the hybrid IQ-tree method [5].

The remainder of this section is organized as follows: In Section 3.1, we present a brief overview of mapping-based approaches, which are shown to fall into three classes. In Section 3.2, we go into the background of how search is performed by using the mapping. In Section 3.3, we show how to perform incremental nearest neighbor search with mapping-based approaches. In Section 3.4, we demonstrate that the result is a correct algorithm when F is contractive, and discuss the implications of non-contractiveness.

3.1 Mapping Methods

Numerous methods have been proposed for mapping objects into multidimensional vectors. Such methods fall into three general categories:

- *Domain-specific mapping methods* are derived using some special properties of the objects.
- *Dimensionality-reduction methods* apply when the data objects are already represented by multi-dimensional points, but of too high a dimensionality for practical use of spatial indexes.
- *General embedding methods* perform the mapping based solely on the distance function d and its properties.

Below, we briefly describe a few examples of methods from each category. Each method makes different assumptions about S and d and their properties. In some cases of domain-specific mapping methods, no explicit distance function d may exist, beyond the subjective judgments of human domain experts. However, the vectors resulting from domain-specific mapping methods often have a much larger dimensionality than is practical for spatial indexing, so it may be necessary to apply dimensionality reduction.

3.1.1 Domain-Specific Methods

Domain-specific methods exist for a wide variety of data types and applications, such as images, strings, and time series. Such methods are often referred to as *feature extraction*, and their results as *feature vectors*.

For images, a number of similarity measures have been proposed, based on different aspects of images, such as color [30], texture [46], and shape [1, 41]. The *color histogram* [30] is a common feature extraction method that allows matching images by color. In its simplest form, such histograms are constructed by dividing the color spectrum into a number of intervals, and then counting the number of pixels in the image whose colors fall into each interval. More accuracy is obtained by dividing the image into several equal-sized parts, each of which has its own color histogram. Thus, the dimensionality of color histogram feature vectors can be quite large, often ranging from 64 to 256. A feature extraction method for shape matching in collections of 2D medical images, specifically tumor images, was presented by Korn et al. [41]. Their method is based on applying a series of morphological operations to each image, resulting in a series of derived images. The values of the components of the feature vector of an image are then simply the numbers of black pixels in each of the derived images (including the original image). The Chessboard distance (L_∞) between the feature vectors of two images a and b is then a lower bound on the distance $d(a,b)$, where d is defined in terms of pixel-wise comparison between the derived images of a and b.

3.1.2 Dimensionality-Reduction Methods

Dimensionality-reduction methods are applicable when $S \subset \mathbb{U} = \mathbb{R}^n$. The rationale for applying dimensionality reduction is that the points in S often lie in a subspace of lower dimensionality than n. Thus, the implicit goal of dimensionality-reduction methods is to identify a subspace V of $\mathbb{R}^n$ that contains S, allowing all distances to be preserved. Of course, the dimensionality of V may still be high, so it may be impossible to completely preserve the distances when reducing the dimensionality down to the target dimensionality m (which is usually significantly smaller than n). The reason why it is advantageous to work with lower-dimensional point sets is that working with high-dimensional point sets can be very expensive. Partly, this is due to the fact that the distance computations are more expensive. More important, however, are various geometric effects that arise in high-dimensional spaces. For example, spatial indexing becomes increasingly ineffective as the dimensionality increases, so that even simple queries may result in examining all the points in S. Recall that the dimensionality of the vectors that result from domain-specific methods is often very high, making it necessary to reduce their dimensionality.

The most common dimensionality-reduction methods are based on linear transformations of the vector set S. Examples of such methods are the essentially equivalent Karhunen-Loève transform (KLT),

singular value decomposition (SVD), and principal component analysis (PCA) methods. Briefly, these methods apply a linear transformation to S, yielding S', that involves rotation and translation. Furthermore, the coordinate axes in S' are ordered based on "importance", as measured by variance. In particular, the linear transformation is such that the first axis in S' is the axis that yields the greatest possible variance for the vectors in S, etc. This enables us to keep only the first few dimensions of the vectors in S' while retaining as much information as is possible with a linear transformation. The drawback of these methods is that they are data-dependent, in that the mapping F is constructed based on all the objects in S. Thus, they are most appropriate in situations where the database is static (e.g., stored on CD-ROM) or changes little (at least in terms of data distribution). Nevertheless, some work has been done on adapting them to situations where the database is dynamic [39].

Other dimensionality-reduction methods exist where the mapping F is independent of S, thus making them more appropriate for dynamic databases. Examples of these methods are the discrete Fourier transform (DFT), the discrete cosine transform (DCT), and wavelet transforms.

Most dimensionality-reduction methods result in a contractive mapping only for certain types of distance functions d. For example, the linear transformation-based methods require d (and δ) to be the Euclidean metric, as it is the only metric that is invariant to rotation. For other choices of d, it is often possible to derive a suitable δ that ensures contractiveness, but this may be at the cost of lower precision of queries on $F(S)$ (see Section 3.2).

3.1.3 General Embedding Methods

In many applications, evaluating distances according to the distance function d is very expensive, and deriving a suitable domain-specific mapping is difficult or impossible. This is where general embedding methods come to the rescue, as the mapping F that they produce is derived based purely on inter-object distances as measured by d, rather than on any knowledge about the properties of the objects in S. Most such methods require d to be a distance metric, thus satisfying the symmetry, non-negativity, and triangle inequality properties. Clearly, such methods can also be used for dimensionality reduction, and may in some cases be preferable to pure dimensionality-reduction methods. Examples of general embedding methods are multidimensional scaling (MDS) [42, 75], FastMap [24], MetricMap [71], and Lipschitz embeddings [43].

In MDS, the mapping F is obtained by minimizing some cost function that measures the quality of the embedding. A common cost function is *stress*, defined as

$$\frac{\sum_{o_1,o_2 \in S}(\delta(F(o_1),F(o_2)) - d(o_1,o_2))^2)}{\sum_{o_1,o_2 \in S} d(o_1,o_2)^2}.$$

Minimizing stress is essentially a non-linear optimization problem, where the variables are the $N \cdot m$ coordinate values corresponding to the embedding (i.e., m coordinate values for each of the N objects). Unfortunately, MDS is virtually useless for similarity search applications, since deriving $F(q)$ — for the purpose of searching in $F(S)$ — requires computing the distances of all objects in S from a query object q. Thus, using MDS for similarity searching is just as expensive as applying brute force search, as both require $O(N)$ distance computations.

In contrast, other methods typically require $O(m)$ distance computations (where m is the target dimensionality) to derive $F(q)$. One such method is FastMap, which is based on the idea that we can imagine that the objects represent points in some Euclidean space of unknown dimensionality. The coordinate axes of this (imaginary) space are constructed one by one, in each iteration choosing two objects, imagining a line passing through them and projecting the rest of the objects onto this line. This process is continued until some fixed number of coordinate axes has been obtained. The operations performed by FastMap are essentially equivalent to rotations and translations [37], so FastMap can be seen as a

heuristic version of KLT/SVD/PCA when S is drawn from a Euclidean space. Unfortunately, when S is not drawn from a Euclidean space (i.e., d is not the Euclidean distance metric), the mapping produced by FastMap is not contractive [37]. This drawback is shared by MetricMap [71], which is also based on concepts from linear algebra.

Lipschitz embeddings define a coordinate space where each axis corresponds to a reference set which is a subset of S. Each coordinate of an object $o \in S$ is defined as the minimum distance from o to an object in the corresponding reference set (if δ is not the Chessboard metric, a suitable scaling factor must also be applied [43] to guarantee contractiveness). SparseMap [38] is a variant of the Lipschitz embeddings that attempts to reduce the computational cost of the embedding, thus making it more suitable for similarity search applications. The heuristics involved in SparseMap mean that the mapping F that is produced is not contractive, but this drawback can be alleviated by a suitable modification of the method [37].

3.2 Object Mapping and Distance Functions

The success of mapping-based approaches clearly depends on how well the distances among the mapped objects correspond to the distances among the original objects. In other words, $\delta(F(o_1),F(o_2))$ should be a reasonable approximation of $d(o_1,o_2)$ for o_1,o_2 in S, at least on the average. At best, we can achieve complete distance preservation with F, such that $\delta(F(o_1),F(o_2)) = d(o_1,o_2)$ for all $o_1,o_2 \in S$.[5] However, this is rare, so we must be satisfied with the distance correspondence being approximate. Thus, the set $R' \subset S$ resulting from a query performed on $F(S)$ (or more accurately, $R' = F^{-1}(R_F)$, where $R_F \subset F(S)$ is the result of the query on $F(S)$) may not match exactly the result R of the corresponding query on S. Two measures, termed *precision* and *recall*, are often used to characterize the correspondence between the two result sets. Precision measures the proportion of the objects in R' that are also in R (i.e., $|R \cap R'|/|R'|$), while recall measures the proportion of the objects in R that are also in R' (i.e., $|R \cap R'|/|R|$). Clearly, they can both be 100% only when $R = R'$, and both are zero if $R \cap R' = \emptyset$. Reduced precision reflects more "garbage" in R' (often termed *false positives*) while reduced recall reflects more "good" objects missing from R' (often termed *false dismissals*).

As an example, consider a range query, where we wish to determine all objects within distance ε from a query object q. Making use of $F(S)$ and its spatial index, we perform a range query on $F(S)$ using $F(q)$ as a query object and the same query radius ε (in some cases, a more appropriate choice is to use a query radius of $f(\varepsilon)$, where f depends on F and δ). In the case of such a range query on $F(S)$, low precision indicates that the distances $\delta(F(q),F(o))$ are often much smaller than $d(q,o)$, $o \in S$. In the worst case, $\delta(F(q),F(o)) \leq \varepsilon = d(q,o_n)$ for all objects $o \in S$, where o_n is the nearest neighbor of q in S, causing a precision of nearly zero. On the other hand, low recall indicates that the distances $\delta(F(q),F(o))$ are typically much larger than $d(q,o)$, $o \in S$. To see why, observe that an object o that is a false positive (i.e., $o \in R'$ but $o \notin R$, where R and R' are defined as above) means that $\delta(F(q),F(o)) \leq \varepsilon < d(q,o)$, while o suffering false dismissal (i.e., $o \in R$ but $o \notin R'$) means that $d(q,o) \leq \varepsilon < \delta(F(q),F(o))$. Thus, the quality of the mapping F with respect to d depends on the extent to which it achieves a balancing act between high precision and high recall. In other words, for any objects $o_1,o_2 \in S$, the distance $\delta(F(o_1),F(o_2))$ between them should ideally be as large as possible, while not exceeding $d(o_1,o_2)$ by a wide margin. The definitions of precision and recall are clearly symmetric with respect to the result sets R and R'. For range queries this symmetry is reflected in the fact that by using a smaller query radius than ε for $F(S)$, precision increases but recall decreases, whereas a larger query radius causes precision to decrease but recall to increase. However, while increasing the query radius is the only way to increase recall (assuming that it is not already 100%), precision can be increased to 100% by simply eliminating from R' those objects that are not within a distance of ε from q (of course, it is still possible that R' does not contain all

[5] When this occurs and d and δ are distance metrics (i.e., satisfy the symmetry, non-negativity, and triangle inequality properties; see Section 4.1), F is said to be an *isometry* and (S,d) and $(\mathbb{R}^m,\delta)$ are said to be *isometric*.

elements of R). This gives rise to a two-step *filter and refine* process, where we use the spatial index on $F(S)$ as a filter in the first step (i.e., finding all objects o that satisfy $\delta(F(q),F(o)) \leq \varepsilon$), and then refine the resulting candidate set in the second step by using the original distance function d. Thus, we gain enhanced precision at the cost of having to perform more work. Of course, use of the filter and refine process (and the spatial index) does not necessarily lead to 100% recall. To ensure that it does, we adjust the query radius for $F(S)$ to $c\varepsilon$, where $\delta(F(q),F(o)) \leq cd(q,o)$ for all $o \in S$. Thus, if $\delta(F(q),F(o)) > c\varepsilon$ for some $o \in S$, then $d(q,o) > \varepsilon$, so $F(o)$ can be safely rejected in the range query on $F(S)$. Unfortunately, for a given query object $q \in \mathbb{U}$ and data set $S \subset \mathbb{U}$, we cannot efficiently derive such a value c on a per query basis. Therefore, the only feasible alternative is to derive a value of c that holds over all possible $q \in S$ and $S \subset \mathbb{U}$:

$$c = \max_{o_1,o_2 \in \mathbb{U}} \left\{ \frac{\delta(F(o_1),F(o_2))}{d(o_1,o_2)} \right\}. \tag{1}$$

The value of c in Equation 1 clearly depends primarily on F and δ, and is termed the *expansion* of F with respect to d and δ.

For some choices of F and δ, the value of c may be ∞, or so large that $\delta(F(q),F(o)) \leq c\varepsilon$ for most $o \in S$, thus leading to a precision of nearly zero (since nearly all objects in S would be reported as a result of the query). Thus, in such cases, 100% recall can be realized only at the cost of poor performance. In many applications, achieving 100% recall is not crucial, in which case performance can be improved at the cost of reduced recall. For the special case when the expansion of F as determined by Equation 1 is no more than 1, i.e., $\delta(F(o_1),F(o_2)) \leq d(o_1,o_2)$ for all $o_1,o_2 \in \mathbb{U}$, F is said to be *contractive* with respect to d and δ. Clearly, when F is contractive, a range query on $F(S)$ with query radius ε always achieves a recall of 100%. Of course, the expansion c may be less than 1, in which case a query radius of $c\varepsilon$ for a range query on $F(S)$ provides improved precision compared to a query radius of ε.

3.3 Incremental Nearest Neighbor Search

In Section 3.2 we saw how we could increase the precision of range queries to 100% through a two-step filter and refine strategy that uses a spatial index T on $F(S)$, the result of mapping all the objects in S. Furthermore, we saw that if F is contractive (or if we adjust the query radius based on the expansion of F), it is also possible to achieve 100% recall. In other words, when F is contractive, the filter and refine range query strategy obtains the exact result that would have been obtained by a linear scan of S (i.e., by checking the range condition against each object in turn, thereby computing N distances, where $N = |S|$). In this section, we first present a filter and refine strategy for performing incremental nearest neighbor search that is applicable when F is contractive and that also guarantees that the result is always correct (i.e., 100% precision and recall). In Section 3.4, we discuss the implications of F not being contractive, and describe possible variants of the strategy for that case.

Incremental nearest search for the mapping-based approach fits readily into the framework presented in Section 2. In fact, if T, the spatial index on $F(S)$, is an R-tree, the search hierarchy is a straightforward extension of the search hierarchy for performing incremental nearest neighbor search using R-trees (see Section 2.2). In general, regardless of what data structure is used for T, the search hierarchy for the mapping-based approach is obtained by extending the search hierarchy for T. In the search hierarchy for T, the elements of type 0 are points in $F(S)$, while the other elements represent nodes in T (and/or possibly some other aspects of the data structure). When extending the search hierarchy for T to form the new search hierarchy, we increment the type numbers for all the element types of the search hierarchy of T, and add the objects in S as the new elements of type 0. Thus, an element e_1 of type 1, representing a point $F(o)$, has as a child the element e_0 of type 0 that represents the corresponding object $o \in S$. For the specific case of the R-tree, recall that the elements of type 1 were minimum bounding rectangles of the objects. However, each object $F(o)$ in $F(S)$ is actually a point, so its minimum bounding box

is degenerate, and thus equivalent to $F(o)$. Therefore, the elements of type 1 in the (original) search hierarchy for the R-tree, representing minimum bounding rectangles, always have identical distances as the corresponding elements of type 0 (i.e., in the original search hierarchy), which represent the points in $F(S)$ (recall that elements of type 2 represent R-tree nodes; see Section 2.2). Thus, the elements of type 1 can be thought of as representing the points in $F(S)$[6], leaving us free to let the elements of type 0 represent the objects in S (replacing the original type 0) when extending the search hierarchy for R-trees.

The distance functions of the elements in the hierarchy are based on d and δ, the distance functions for S and $F(S)$, respectively. In particular, for an element e_0 of type 0, $d_0(q,e_0) = d(q,o)$ where $o \in S$ is the object that corresponds to e_0. The distance function for elements of type 1, corresponding to points in $F(S)$, is defined as

$$d_1(q,e_1) = \delta(F(q),F(o)), \tag{2}$$

where $F(o)$ is the point represented by e_1. An element of type $t \geq 2$ essentially represents a region in $\mathbb{R}^m$ that contains all the points in the corresponding subtree. If T is an R-tree, for example, elements of type 2 represent nodes, which cover regions that are rectilinear hyperrectangles. Thus, for $t \geq 2$, we define the distance function

$$d_t(q,e_t) = \min_{p \in R}\{\delta(F(q),p)\}, \tag{3}$$

where R is the region covered by element e_t. Usually, Equation 3 can be evaluated with fairly simple geometric calculations (e.g., if δ is some Minkowski metric and T is an R-tree, the distance is equal to the distance between $F(q)$ and one of the faces of the hyperrectangle R, or zero if $F(q)$ is inside R). The upper-bound distance functions $\hat{d}_t$ (see Section 2.4 for their usage) can be defined in a similar way, except that max would be used in the equivalent of Equation 3. However, since $\delta(F(q),F(o))$ cannot simultaneously be an upper and lower bound on $d(q,o)$ unless $\delta(F(q),F(o)) = d(q,o)$, we cannot define $\hat{d}_1$ exactly like d_1 in Equation 2. In Section 3.4 we discuss how a suitable definition can be arrived at, but this may not always be practical.

Clearly, a range query on the search hierarchy defined above is equivalent to the two-step range query algorithm described in Section 3.2, using the same query radius ε. Assuming that T is an R-tree, incremental nearest neighbor search using the hierarchy would proceed as follows (see Figure 3). Initially, the root of the R-tree would be inserted on the queue as an element of type 2. If the element e_t taken off the queue is of type 2, representing a node n, we insert elements of type 1 (if n is a leaf node) or 2 (if n is a nonleaf node) into the priority queue based on the entries in n. If the element e_t that comes off the priority queue is of type 1, we insert the corresponding object as an element of type 0. Finally, if the element e_t is of type 0, we report the corresponding object as the next nearest neighbor. Figure 5 depicts how the algorithm would proceed on a small example consisting of three objects and their mapped versions. Both are shown here as two-dimensional points; we can also think of the figure as merely showing the relative distances of the objects from q and the mapped objects from $\mathsf{F(q)}$. Observe that the distance $d(\mathsf{q},\mathsf{c})$ need not be computed by the algorithm, since $\delta(\mathsf{F(q)},\mathsf{F(c)}) > d(\mathsf{q},\mathsf{b})$.

3.4 Correctness and Performance

The distance functions defined above guarantee the correctness of the algorithm when F is contractive, as they satisfy the correctness criterion outlined in Section 2.5. In particular, for any elements e_0 and e_1, representing o and $F(o)$, respectively, we have $d_1(q,e_1) = \delta(F(q),F(o)) \leq d(q,o) = d_0(q,e_0)$. Furthermore, for any ancestor e_t, $t \geq 1$, of an element e_0, we know that $d_t(q,e_t) \leq d_1(q,e_1) \leq d_0(q,e_0)$, where e_1 is the parent of e_0, since the region represented by e_t must contain the mapped point $F(o)$ represented by e_1. Thus, if o_k is the k^{th} nearest neighbor of q, the algorithm will perform as many distance

[6] Since points occupy half as much space as rectangles, we could even replace the minimum bounding rectangles in the leaf nodes of the R-tree by the points themselves, thus doubling the capacity of the leaf nodes.

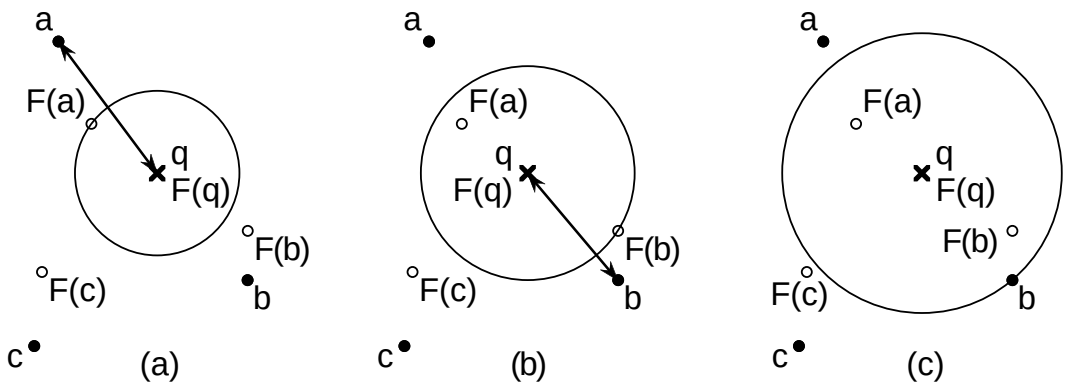

Figure 5: Progress of the incremental nearest neighbor algorithm on a small example involving three objects. Both the objects and their mapped versions are depicted as two-dimensional points. (a) The search region first reaches F(a), and $d(\mathrm{q},\mathrm{a})$ is computed. (b) F(b) is reached next, causing $d(\mathrm{q},\mathrm{b})$ to be computed. (c) The object b is determined to be the nearest neighbor of q.

computations (i.e., using d) as a range query on the same search hierarchy with $\varepsilon = d(q,o_k)$ (since both would visit exactly the same elements of type 1, i.e., those whose distance from q is less than or equal to $d(q,o_k)$). Observe that the precision of the range query on $F(S)$ with $\varepsilon = d(q,o_k)$ determines the performance of the algorithm, in terms of the number of distance computations. In particular, the precision P of the range query equals $|R \cap R'|/|R'| = |R|/|R'|$ (the equality is due to the contractiveness of F), where $R = \{o \mid d(q,o) \leq d(q,o_k)\}$ and $R' = \{F(o) \mid \delta(F(q),F(o)) \leq d(q,o_k)\}$. On the other hand, the proportion of excess distance computations, $|R'|/|R| - 1 = 1/P - 1$, is an appropriate measure of the performance of the incremental nearest neighbor algorithm. In the best case, the precision is 100% and the proportion of excess distance computation is 0%.

If F is not contractive (i.e., $c > 1$, as determined by Equation 1), then we may have $\delta(F(q),F(o)) > d(q,o)$ in which case the correctness criterion doesn't hold, implying that the algorithm may not return the correct result — that is, the object o'_k reported as the k^{th} nearest neighbor of q may not be the actual k^{th} nearest neighbor o_k. In other words, the result of the algorithm would only be approximate in that the objects in S may not be reported in strictly increasing order of distance from q. For example, in Figure 5, if $\delta(\mathrm{F(q)},\mathrm{F(c)}) > d(\mathrm{q},\mathrm{b}) > d(\mathrm{q},\mathrm{c})$, then c would be the actual nearest neighbor rather than b, but the algorithm would still report b as the nearest neighbor. Nevertheless, we can still characterize the performance of the incremental nearest neighbor algorithm in terms of a corresponding range query. In particular, obtaining the first k "approximate" neighbors with the algorithm requires the same number of distance computations as would be performed with a range query with $\varepsilon = \max_{i=1}^{k}\{d(q,o'_i)\}$, where o'_i is the i^{th} object returned by the algorithm.

By suitable modification of the distance functions, it is possible to achieve 100% recall even if the expansion c of F is greater than 1 (as determined by Equation 1). In particular, this is done by adding $\frac{1}{c}$ as a multiplicative factor to the distance functions in Equations 2 and 3 (e.g., by defining $d_1(q,e_1) = \delta(F(q),F(o))/c$). In this case, to obtain k neighbors with the algorithm, the set R' of objects whose distances are computed corresponds to the result of a range query on $F(S)$ with query radius $\varepsilon = cd(q,o_k)$, where o_k is the k^{th} nearest neighbor of q. However, if the precision of this range query with respect to the range query on S with query radius $d(q,o_k)$ tends to be low, the proportion of excess distance computations (i.e., $|R'|/k - 1$) tends to be high, so modifying the distance functions in this way may cause an unacceptable degradation in performance. Thus, in order to get good performance, we may need to live with the distance functions in Equations 2 and 3, thereby sacrificing accuracy in the result of the incre-

mental nearest neighbor algorithm. On the other hand, if c is substantially smaller than 1, modifying the distance functions should improve the performance measurably, since the precision of a range query on $F(S)$ would be better with $\varepsilon = cd(q,o_k)$ than with $\varepsilon = d(q,o_k)$ (i.e., with respect to a range query on S with query radius $d(q,o_k)$). In general, if a function f can be derived such that $f(\delta(F(q),F(o))) \leq d(q,o)$, for all $o \in S$, then we can achieve correctness by defining the distance functions d_1 and d_t, $t \geq 2$, as follows:

$$\begin{aligned} d_1(q,e_1) &= f(\delta(F(q),F(o))), \text{ and} \\ d_t(q,e_t) &= \min_{p \in R}\{f(\delta(F(q),p))\}, t \geq 2, \end{aligned}$$

where o and R are defined as in Equations 2 and 3. The definitions above for the case of an expansion of c are special cases of these definitions, where $f(x) = x/c$.

The upper-bound distance functions $\hat{d}_t$ can be properly defined in a similar way. In particular, assume that a function g can be defined such that $g(\delta(F(q),F(o))) \geq d(q,o)$ for all $o \in S$, and let

$$\begin{aligned} \hat{d}_1(q,e_1) &= g(\delta(F(q),F(o))), \text{ and} \\ \hat{d}_t(q,e_t) &= \max_{p \in R}\{g(\delta(F(q),p))\}, t \geq 2, \end{aligned}$$

where o and R are, again, defined as in Equations 2 and 3. For example, if $c'd(q,o) \leq \delta(F(q),F(o))$, then $g(x) = x/c'$. Incidentally, the quantity c/c' is termed the *distortion* of F with respect to d and δ, where $c'd(o_1,o_2) \leq \delta(F(o_1),F(o_2) \leq cd(o_1,o_2)$ for all $o_1, o_2 \in \mathbb{U}$. Recall that the $\hat{d}_t$ functions are needed only when a minimum bound $D_{\min}$ is set on the distances of the desired objects, or when finding farthest neighbors (see Section 2.4). The performance of the algorithm in these applications using the above definition of $\hat{d}_t$ depends on the precision of a "reverse" range query on $F(S)$, i.e., a query that seeks the objects o such that $d(q,o) \geq \lambda$. If many objects $o \in S$ existed such that $g(\delta(F(q),F(o))) \geq \lambda$ while $d(q,o) < \lambda$, the performance would clearly suffer as many false positives would occur.

4 Incremental Similarity Search with Distance-Based Indexing

In Section 3 we described a solution to similarity search that involved mapping into a low-dimensional vector space in order to take advantage of spatial indexing structures. An alternative is to construct index structures that are based solely on distances between objects. A number of such methods have been proposed over the past few decades, some of the earliest being due to Burkhard and Keller [11]. These methods generally assume that (S,d) forms a finite metric space (see Section 4.1). Typical of distance--based indexing structures are *metric trees* [65, 66], which are binary trees that result in recursively partitioning a data set into two subsets at each node. Uhlmann [66] identified two basic partitioning schemes, *ball partitioning* and *generalized hyperplane partitioning*.

In ball partitioning, the data set is partitioned based on distances from one distinguished object, sometimes called a *vantage point* [74], i.e., into the subset that is inside and the subset that is outside a ball around the object (e.g., see Figure 6a). In generalized hyperplane partitioning, two distinguished objects a and b are chosen and the data set is partitioned based on which of the two distinguished objects is the closest, i.e., all the objects in subset A are closer to a than to b, while the objects in subset B are closer to b (e.g., see Figure 6b).

Below, we collectively use the term *pivot* to refer to any type of distinguished object that can be used during search to achieve pruning of other objects, following the convention of Chávez et al. [15]. In other words, a pivot $p \in S$ is an object for which we have some information about its distance from some or all objects in S, e.g., for all objects $o \in S' \subseteq S$ we know

1. the exact value of $d(p,o)$,

2. that $d(p,o)$ lies within some range $[r_{\text{lo}}, r_{\text{hi}}]$ of values, or

3. that o is closer to p than to some other object $p' \in S$.

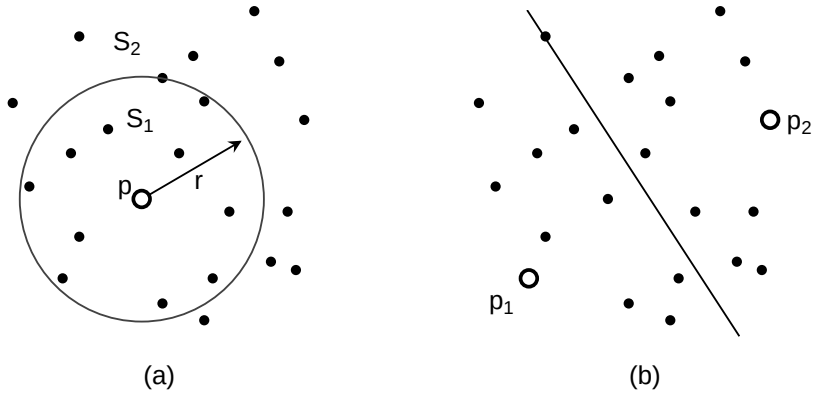

Figure 6: Possible top-level partitionings of a set of objects (depicted as two-dimensional points) in a metric tree using (a) ball partitioning and (b) generalized hyperplane partitioning.

While most distance-based indexing structures are variations on and/or extensions of metric trees, a few radically different approaches exist. Several methods based on distance matrices have been designed [67, 49, 70]. In these methods, all or some of the distances between the objects in the data set are precomputed. Then, when evaluating queries, once we have computed the actual distances of some of the objects from the query object, the distances of the other objects can be estimated based on the precomputed distances. Clearly, these distance matrix methods do not form a hierarchical partitioning of the data set, but combinations of such methods and metric tree-like structures have been proposed [47]. The *sa-tree* [50] is another departure from metric trees, inspired by the Voronoi diagram. In essence, the sa-tree records a portion of the Delaunay graph of the data set, a graph whose vertices are the Voronoi cells, with edges between adjacent cells.

In this section, we describe a number of distance-based indexing methods and show how to perform incremental nearest neighbor search on them. The focus of the discussion is on how to construct a search hierarchy for each structure such that incremental nearest neighbor search can be implemented. Thus, we do not always provide much detail about construction algorithms or devote much effort to comparing and/or contrasting different structures. Furthermore, with a few exceptions, we do not describe other nearest neighbor algorithms in this section (however, see Section 5).

This section is organized as follows. In Section 4.1 we discuss properties of the distance metric useful for search pruning. In Section 4.2 we describe the *vp-tree* [74] and other variants of metric trees that employ ball partitioning. In Section 4.3 we present the *gh-tree* [66] and other variants of metric trees that employ generalized hyperplane partitioning. In Section 4.4 we describe the *M-tree* [19], a dynamic and balanced metric tree variant, suitable for disk-based implementation. In Section 4.5 we introduce the sa-tree [50]. Finally, in Section 4.6 we describe AESA [67] and LAESA [49], methods that rely on distance matrices.

4.1 Distance Metric and Search Pruning

As mentioned earlier, the indexed objects must reside in a finite metric space (S,d). This means that the distance function d must satisfy the following three properties, where $o_1, o_2, o_3 \in S$:

1. $d(o_1, o_2) = d(o_2, o_1)$ (symmetry)

2. $d(o_1, o_2) \geq 0$, $d(o_1, o_2) = 0$ iff $o_1 = o_2$ (non-negativity)

3. $d(o_1, o_3) \leq d(o_1, o_2) + d(o_2, o_3)$ (triangle inequality)

The indexing methods discussed in this section are often applicable even when these three properties are relaxed. For example, it rarely matters if $d(o_1, o_2) = 0$ for some pairs of distinct objects o_1 and o_2 (in this case, d is often termed a *pseudo-metric*). Furthermore, adequate performance can often be attained even if the triangle inequality is occasionally violated[7], but this leads to approximate results (i.e., we cannot guarantee that the nearest neighbors are obtained in strictly non-decreasing order of distance).

Of the distance metric properties, the triangle inequality is the key property for pruning the search space when processing queries. However, in order to make use of the triangle inequality, we often find ourselves applying the symmetry property. Furthermore, the non-negativity property allows discarding negative values in formulas. Below, we enumerate a number of results that can be derived based on the metric properties. Our goal is to provide ammunition for use in later sections when constructing distance functions that lower-bound or upper-bound the distances between a query object and the objects in a subtree of some search hierarchy (as defined in Section 2.2). Thus, we provide lower and upper bounds on the distance $d(q, o)$ between a query object q and some object o, given some information about distances between q and o and some other object(s). The reader may wish to skim over this section on first reading and refer back to it as needed.

Recall that $S \in \mathbb{U}$, where $\mathbb{U}$ is some underlying set, usually infinite, and we assume that $(\mathbb{U}, d)$ is also a metric space (i.e., that d also satisfies the above properties on $\mathbb{U}$). For generality, we present our results in terms of $(\mathbb{U}, d)$, since a query object is generally not in S. In the first lemma, we explore the situation where we know the distances from an object p to both q and o (while the distance between q and o is unknown).

Lemma 3 *Given any three objects $q, p, o \in \mathbb{U}$, we have*

$$|d(q, p) - d(p, o)| \leq d(q, o) \leq d(q, p) + d(p, o). \tag{4}$$

Thus, knowing $d(q, p)$ and $d(p, o)$, we can bound the distance of $d(q, o)$ from both below and above.

Proof The upper bound is a direct consequence of the triangle inequality. For the lower bound, notice that $d(p, o) \leq d(p, q) + d(q, o)$ and $d(q, p) \leq d(q, o) + d(o, p)$ according to the triangle inequality. The first inequality implies $d(p, o) - d(p, q) \leq d(q, o)$, while the second implies $d(q, p) - d(o, p) \leq d(q, o)$. Therefore, combining these inequalities and making use of symmetry, we obtain $|d(q, p) - d(p, o)| \leq d(q, o)$, as desired. ■

Figure 7a illustrates the situation where the lower bound $|d(q, p) - d(p, o)|$ established in Lemma 3 is nearly attained. Clearly, in the figure, $d(q, o)$ is nearly as small as $d(q, p) - d(p, o)$. The opposite relationship (i.e., $d(q, o)$ being nearly as small as $d(p, o) - d(q, p)$) is obtained by exchanging q and o in the figure. Similarly, Figure 7b illustrates the situation where the upper bound $d(q, p) + d(p, o)$ is nearly attained.

In the next lemma, we assume that we know the distance between q and p, but that the distance between p and o is only known to be within some range. This is illustrated in Figure 7c, where we show three different positions of the query object q. The lower bounds on the distances $d(q_1, o)$ and $d(q_2, o)$ are indicated with gray arrows, and the upper bound on $d(q_2, o)$ is indicated with a gray broken arrow.

[7] Distance functions for DNA data that are based on edit distance are usually of this nature [63].

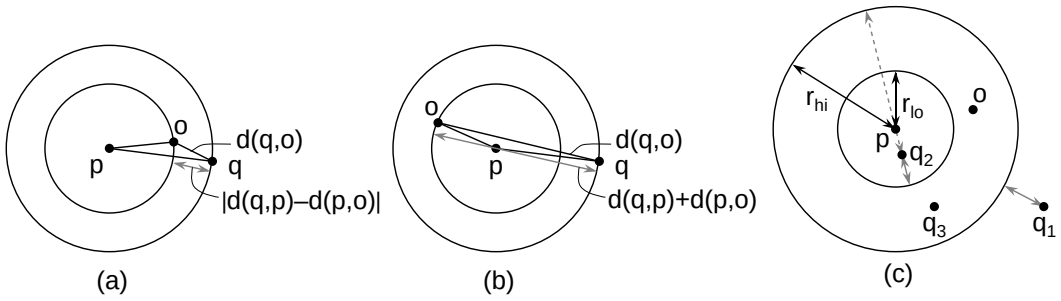

Figure 7: Illustration of distance bounds. Given $d(q,p)$ and $d(p,o)$, a lower bound (a) and an upper bound (b) can be established for $d(q,o)$. (c) Given that $r_{\text{lo}} \leq d(p,o) \leq r_{\text{hi}}$, we can establish lower and upper bounds on $d(q,o)$. Three positions are shown for q, demonstrating three cases that can arise.

Lemma 4 *Let o and p be objects in $\mathbb{U}$ such that $r_{\text{lo}} \leq d(o,p) \leq r_{\text{hi}}$. The distance $d(q,o)$ from $q \in \mathbb{U}$ to o can be bounded as follows, given $d(q,p)$:*

$$\max\{d(q,p) - r_{\text{hi}}, r_{\text{lo}} - d(q,p), 0\} \leq d(q,o) \leq d(q,p) + r_{\text{hi}}. \tag{5}$$

Proof Again, we use the triangle inequality to prove these bounds. In particular, from the inequality $d(q,p) \leq d(q,o) + d(o,p)$ and the upper bound $d(o,p) \leq r_{\text{hi}}$, we obtain $d(q,p) - d(q,o) \leq d(o,p) \leq r_{\text{hi}}$, which implies that $d(q,p) - r_{\text{hi}} \leq d(q,o)$ (e.g., see q_1 in Figure 7c). Similarly, we can combine the triangle inequality and the lower bound on $d(o,p)$ to obtain $r_{\text{lo}} \leq d(o,p) \leq d(q,o) + d(q,p)$, which implies that $r_{\text{lo}} - d(q,p) \leq d(q,o)$ (e.g., see q_2 in Figure 7c). Either or both of these lower bounds can be negative (e.g., for q_3 in Figure 7c), whereas distance values are required to be non-negative. Thus, the overall lower bound in Equation 5 is obtained by taking the maximum of zero and these two lower bounds. The upper bound in Equation 5 is obtained by a straightforward application of the triangle inequality and the upper bound on $d(o,p)$, i.e., $d(q,o) \leq d(q,p) + d(o,p) \leq d(q,p) + r_{\text{hi}}$ (e.g., see the broken arrow from q_2 through p to the outer boundary in Figure 7c). ■

In some situations, the distance $d(q,p)$ in Lemma 4 may not be known exactly. The next lemma establishes bounds on the distance from q to o in such circumstances:

Lemma 5 *Let o, p, and q be objects in $\mathbb{U}$ for which $d(o,p)$ is known to be in the range $[r_{\text{lo}}, r_{\text{hi}}]$ and $d(q,p)$ is known to be in the range $[s_{\text{lo}}, s_{\text{hi}}]$. The distance $d(q,o)$ can be bounded as follows:*

$$\max\{s_{\text{lo}} - r_{\text{hi}}, r_{\text{lo}} - s_{\text{hi}}, 0\} \leq d(q,o) \leq r_{\text{hi}} + s_{\text{hi}}. \tag{6}$$

Proof Substituting s_{lo} for the first instance of $d(q,p)$ in Equation 5 can only reduce the lower bound. Thus, we find that $s_{\text{lo}} - r_{\text{hi}} \leq d(q,o)$. The same is true when substituting s_{hi} for the second instance of $d(q,p)$ in the equation, as this instance is subtractive, which shows that $r_{\text{lo}} - s_{\text{hi}} \leq d(q,o)$. Similarly, substituting s_{hi} for the last instance of $d(q,p)$ in the equation increases the upper bound, so we obtain $d(q,o) \leq r_{\text{hi}} + s_{\text{hi}}$. ■

Clearly, the roles of the two ranges in Lemma 5 are symmetric. For an intuitive understanding of the lower bound, imagine two shells around p, one with radius range $[r_{\text{lo}}, r_{\text{hi}}]$ (where o is allowed to reside) and the other with radius range $[s_{\text{lo}}, s_{\text{hi}}]$ (where q is allowed to reside). As illustrated by the shaded arrow in Figure 8a, the minimum distance between q and o is equal to the space between the shells, if any.

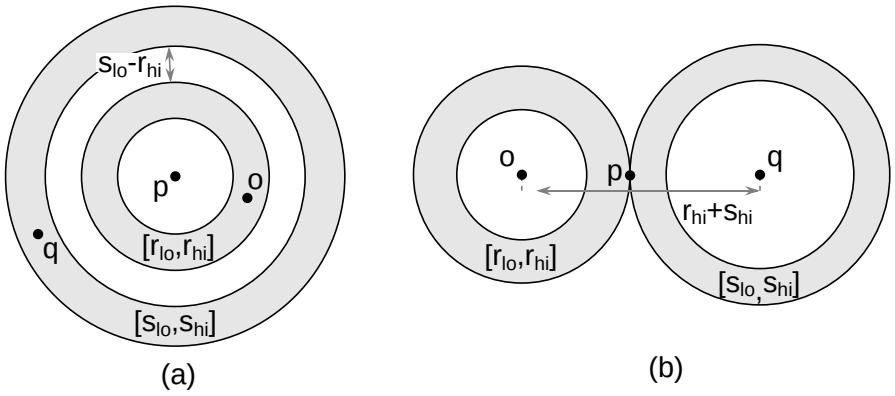

Figure 8: (a) The lower bound on $d(q,o)$ is illustrated for the case when $d(p,o)$ is in the range $[r_{\text{lo}}, r_{\text{hi}}]$ and $d(p,q)$ is in the range $[s_{\text{lo}}, s_{\text{hi}}]$. (b) Illustration of how the upper bound on $d(q,o)$ can be attained when $d(p,o)$ and $d(p,q)$ are in these ranges.

Similarly, the upper bound can be understood by visualizing shells around q and o, with p at the outer edge of each shell, as illustrated in Figure 8b.

In some distance-based indexes, objects are partitioned based on relative closeness to two or more objects. The following lemma provides a result that we can use in such situations:

Lemma 6 *Let $o \in \mathbb{U}$ be an object that is closer to p_1 than to p_2, or equidistant from both (i.e., $d(p_1,o) \leq d(p_2,o)$). Given $d(q,p_1)$ and $d(q,p_2)$, we can establish a lower bound on $d(q,o)$:*

$$\max\left\{\frac{d(q,p_1)-d(q,p_2)}{2}, 0\right\} \leq d(q,o). \tag{7}$$

Proof From the triangle inequality, we have $d(q,p_1) \leq d(q,o)+d(p_1,o)$, which yields $d(q,p_1)-d(q,o) \leq d(p_1,o)$. When combined with $d(p_2,o) \leq d(q,p_2)+d(q,o)$ (from the triangle inequality) and $d(p_1,o) \leq d(p_2,o)$, we obtain $d(q,p_1)-d(q,o) \leq d(q,p_2)+d(q,o)$. Rearranging yields $d(q,p_1)-d(q,p_2) \leq 2d(q,o)$, which yields the first component of the lower bound in Equation 7, the second component being furnished by non-negativity. ■

One way to get some intuition about this result is to consider the situation shown in Figure 9a where q lies on the line between p_1 and p_2 in a two-dimensional Euclidean space, closer to p_2. If o is closer to p_1, it is to the left of the horizontal line midway between p_1 and p_2 which separates the regions in which objects are closer to p_1 or to p_2. Thus, $d(q,o)$ is lower-bounded by the distance from q to the dividing line, which equals $(d(q,p_1)-d(q,p_2))/2$ for the particular position of q in the figure. If we move q parallel to the dividing line (i.e., up or down in Figure 9a), the distance from q to the line is clearly unchanged. However, the difference between $d(q,p_1)$ and $d(q,p_2)$ can be shown to decrease as both increase[8], so the value of $(d(q,p_1)-d(q,p_2))/2$ will also decrease. In other words, we see that $(d(q,p_1)-d(q,p_2))/2$ is exactly the distance from q to the dividing line in the figure, while $(d(q,p_1)-d(q,p_2))/2$ decreases as q is moved while keeping the distance from q to the dividing line constant. Therefore, the value $(d(q,p_1)-d(q,p_2))/2$ is indeed a lower bound on the distance from q to the dividing line, and thus also a lower

[8]Figure 9b depicts the relative distances for a query point q' that is above q. From $\alpha^2 = a^2 + c^2$ we obtain $\alpha^2 - a^2 = (\alpha-a)(\alpha+a) = c^2$ or $\alpha - a = \frac{c^2}{\alpha+a}$. In the same manner, we can show that $\beta - b = \frac{c^2}{\beta+b}$. Since q is closer to p_2, we have $a > b$ and $\alpha > \beta$, and therefore $\alpha + a > \beta + b$. Thus, $\alpha - a = \frac{c^2}{\alpha+a} < \frac{c^2}{\beta+b} = \beta - b$, implying that $\alpha - \beta < a - b$, and thus $(d(q',p_1)-d(q',p_2))/2 < (d(q,p_1)-d(q,p_2))/2$.

bound on the distance between q and o. Note that this argument holds for all positions of q that are closer to p_2 than to p_1, as the initial position of q can be anywhere on the line between p_1 and p_2. Observe that without additional information, an upper bound on $d(q,o)$ cannot be established, as o may be arbitrarily far away from p_1 or p_2.

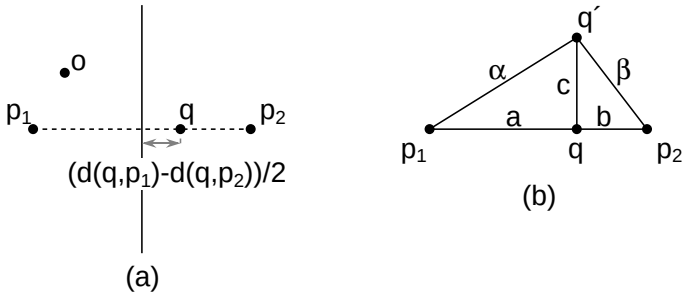

Figure 9: (a) Lower bound on $d(q,o)$, illustrated in a two-dimensional Euclidean space when q is on the line between p_1 and p_2, closer to p_1, while o is closer to p_2. (b) The lower bound can be shown to decrease when q is moved off the line (e.g., to q').

4.2 Ball Partitioning Methods

4.2.1 The VP-Tree

The vp-tree (Vantage Point Tree) [74] is an example of an indexing method that uses ball partitioning (and thus is a variant of the metric tree [65, 66]). In this method, we pick a pivot p from S (termed a vantage point in [74]), compute the median r of the distances of the other objects to p, and then divide the remaining objects into roughly equal-sized subsets S_1 and S_2 as follows:

$$S_1 = \{o \in S \setminus \{p\} \mid d(p,o) < r\}$$
$$S_2 = \{o \in S \setminus \{p\} \mid d(p,o) \geq r\}$$

Thus, the objects in S_1 are *inside* the ball of radius r around p, while the objects in S_2 are *outside* this ball. Applying this rule recursively leads to a binary tree, where a pivot object is stored in each internal node, with the left and right subtrees corresponding to the subsets inside and outside the corresponding ball, respectively. In the leaf nodes of the tree we would store one or more objects, depending on the desired capacity. An example of such a partition is shown in Figure 6a. Note that the ball regions play a role somewhat similar to the bounding rectangles in the R-tree In fact, we can define *bounding values* for each subset S_1 and S_2. In particular, for $o \in S_i$ we have $d(p,o) \in [r_{i,\mathrm{lo}}, r_{i,\mathrm{hi}}]$, for some bounding values $r_{i,\mathrm{lo}}$ and $r_{i,\mathrm{hi}}$. Given only the radius r, the known bounds are $[r_{1,\mathrm{lo}}, r_{1,\mathrm{hi}}] = [0,r]$ and $[r_{2,\mathrm{lo}}, r_{2,\mathrm{hi}}] = [r,\infty]$ (or, more accurately, $[0, r-\delta]$ and $[r,M]$, respectively, where $\delta \leq d(o,o_1) - d(o,o_2)$ and $M \geq d(o_1,o_2)$ for all o, o_1, o_2 in S). For the tightest bounds possible, all four bounding values can be stored in the node corresponding to p. However, this may yield improved search performance, but perhaps at the price of excessive storage cost. Below, we use r_{lo} and r_{hi} to denote bounding values of S_1 or S_2 for statements or equations that apply to either subset.

The simplest method of picking pivots is to simply select at random. Yianilos [74] argues that a more careful selection procedure can yield better search performance (but at the price of a higher construction cost). The method he proposes is to take a random sample from S, and choose the object among the sample objects that has the best spread (defined in terms of the variance) of distances from a subset of S, also chosen at random. For a data set drawn from a Euclidean space for which the data points are

relatively uniformly spread over a hypercube c, this would tend to pick points near corners as pivots (the observation that such pivots are preferable was first made by Shapiro [62]). Choosing such points as pivots can be shown to minimize the boundary of the ball that is inside c (e.g., the length of the boundary in Figure 10a is greater than that in Figure 10b), which Yianilos [74] argues increases search efficiency. Some intuitive insight into the argument that the boundary is reduced as the pivot is moved farther from the center of c can be gained by considering that if we are allowed to pick points outside c as pivots, the resulting partitioning of the hypercube increasingly resembles a partitioning by a hyperplane (e.g., see Figure 10c). Notice that the areas of the two regions inside c formed by the partitioning tend to be about the same when the points are uniformly distributed, and the length of the partitioning arc inside c is inversely proportional to the distance between the pivot point and the center of c (see Figure 10). Observe also that the length l of the partitioning arc decreases even more as the pivot is moved further away from c (e.g., see Figure 10c).

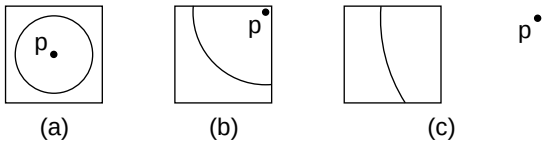

Figure 10: Depiction of partitions of a set of points in a two-dimensional Euclidean space, assumed to be uniformly distributed in a cube c, for pivot points (a) in the center of c, (b) in a corner of c, and (c) outside c.

In the vp-tree, the ball radius is always chosen as the median, so that the two subsets are roughly equal in size. Another possibility would be to split at the mid-point between the distances of the objects in $S \setminus \{p\}$ that are closest and farthest from p, as proposed by Chávez et al. [15] (and inspired by Burkhard and Keller [11]). This yields a partition into equal-width "shells" around p. Chávez et al. [15] argue that splitting at the mid-point yields better partitions for data sets whose "inherent dimensionality" is high, as the objects outside the ball may reside in a thin "shell" when always splitting at the median [15]. However, the disadvantage of splitting at the mid-point is that the resulting tree is not balanced, as is the case when splitting at the median.

Clearly, search algorithms are fundamentally the same regardless of how the pivot and ball radius are determined, since the basic structure is the same. First, let us examine how a range query would proceed for a query object q and query radius ε — that is, we wish to determine all objects o such that $d(q,o) \leq \varepsilon$. Such a range query is most easily implemented with a depth-first traversal of the tree. When visiting a node n with pivot p and ball radius r, we must decide whether to visit the left and/or right child of n. Lemma 4 enables us to establish lower bounds on the distances from q to objects in the left and right subtrees. If the query radius is less than the lower bound for a subtree, there is no need to visit that subtree. For example, in Figure 11a the left subtree (for the objects inside the ball) need not be visited, while in Figure 11b, the left subtree must be visited. Formally, from Equation 5, in Lemma 4, with $r_{\text{lo}} = 0$ and $r_{\text{hi}} = r$, we know that the distance from q to an object in the left subtree of n is at least $\max\{d(q,p) - r, 0\}$. Similarly, by applying the equation with $r_{\text{lo}} = r$ and $r_{\text{hi}} = \infty$, we know that the distance from q to an object in the right subtree of n is at least $\max\{r - d(q,p), 0\}$. Thus, we visit the left child if and only if $\max\{d(q,p) - r, 0\} \leq \varepsilon$ and the right child if and only if $\max\{r - d(q,p), 0\} \leq \varepsilon$.

We can also use a similar approach to design an incremental nearest neighbor algorithm for the vp-tree, by specifying the search hierarchy according to the framework presented in Section 2.2. Being that the vp-tree is hierarchical, the search hierarchy essentially falls out of the existing hierarchical structure. Thus, the elements of the search hierarchy correspond to the objects (type 0) and the nodes in the vp-tree (type 1). Again, we observe that the ball regions for the vp-tree nodes play the same role as bounding

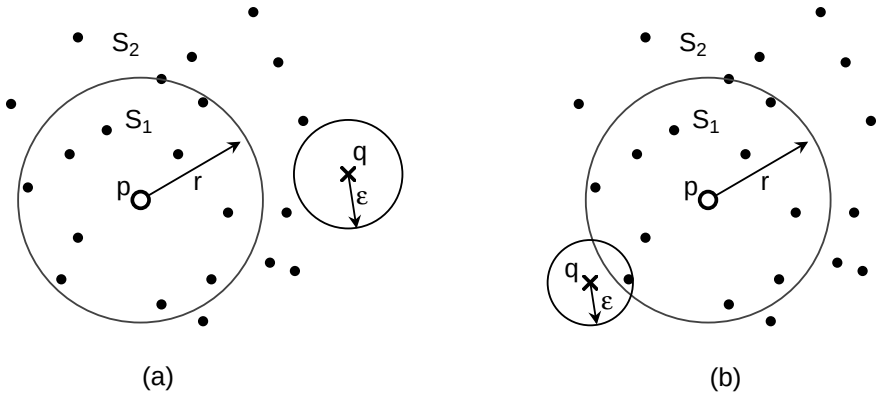

Figure 11: During a range query with query radius ε, the subtree corresponding to the inside of the ball need not be visited in (a) while it must be visited in (b).

rectangles in the R-tree. The resulting search hierarchy for a small sample vp-tree is depicted in Figure 12. Note that elements of type 1 can produce elements of type 0 and 1.

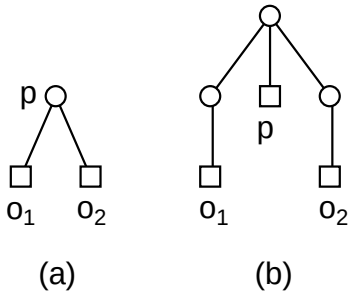

Figure 12: (a) An example vp-tree for three objects p, o_1, and o_2, and (b) the search hierarchy induced by the vp-tree.

We now define the distance functions d_t for the distance between the query object q and elements e_t of type t, $t = 0, 1$. Since elements e_0 of type 0 represent objects, d_0 is simply equal to d. As for d_1, recall that the value of $d_1(q, e_1)$ should be a lower bound on the distance $d(q, o)$ for any object o in the sub-hierarchy rooted at e_1. The information we have on hand to derive such a lower bound is the value of $d(q, p)$ and the fact that $d(p, o)$ is in the range $[r_{\text{lo}}, r_{\text{hi}}]$, where p is the pivot of the parent e_1 and $[r_{\text{lo}}, r_{\text{hi}}]$ defines the shell around p containing the objects in the subtree rooted at e_1 (i.e., $[0, r]$ if e_1 is a left child and $[r, \infty]$ if e_1 is a right child, where r is the ball radius). Thus, as we saw above for the range query, we can make use of Lemma 4, which gives lower and upper bounds on $d(q, o)$ based on exactly such information. In particular, the definition of d_1 as obtained from the lemma is:

$$d_1(q, e_1) \quad = \quad \max\{d(q, p) - r_{\text{hi}}, r_{\text{lo}} - d(q, p), 0\}.$$

This definition of d_1 is general in that it accounts for e_1 being either a left child (in which case $r_{\text{lo}} = 0$ and $r_{\text{hi}} = r$) or a right child (in which case $r_{\text{lo}} = r$ and $r_{\text{hi}} = \infty$)[9]. Furthermore, for either case, it also accounts for q being either inside or outside the region for e_1 (i.e., inside or outside the ball around p of radius r). Since the lemma guarantees that $d_1(q, e_1) \leq d(q, o)$ for any object o in the subtree rooted at

[9] As we pointed out before, tight distance bounds for each subtree could be stored in each vp-tree node instead of just the median [74], thereby causing d_1 and $\hat{d}_1$ to yield improved bounds. This can improve search performance, but at the cost of an increase in the storage requirement (i.e., four distance values in each node instead of just one).

e_1, we are assured that the incremental nearest neighbor algorithm is correct when applied to the vp-tree search hierarchy that we have defined (see Section 2.5).

The upper-bound distance function $\hat{d}_1$ can also be derived from the result in Lemma 4 (i.e., by Equation 5):

$$\hat{d}_1(q, e_1) \quad = \quad d(q, p) + r_{\text{hi}},$$

where p and r_{hi} are defined as above. Recall that upper-bound distance functions are used when performing farthest-neighbor queries and when a minimum distance bound is imposed on the query results (see Section 2.4). The correctness of such algorithm variants is guaranteed by the lemma, i.e., since $\hat{d}_1(q, e_1) \geq d(q, o)$ for any object o in the subtree rooted at e_1.

Given the search hierarchy defined above, incremental nearest neighbor search proceeds as described in Section 2.3. In particular, when the element obtained from the queue represents an object (i.e., is of type 0), we report it as the next nearest neighbor. Otherwise, the element is of type 1, representing a node n. If n is a nonleaf node with pivot p and ball radius r, we compute $d(q, p)$ and insert p into the priority queue as an element of type 0. Furthermore, the left and right children of n are inserted into the priority queue as elements of type 1 using the distance function defined above. If n is a leaf node, we perform the same action as for pivots of nonleaf nodes for the object(s) in the node.

In Section 2.5, we mentioned that the incremental nearest neighbor algorithm is correct even if some elements produce elements whose distance to the query object is smaller, provided that the correctness criterion with respect to the data objects is not violated. Such situations arise frequently in the vp-tree. In the example shown in Figure 13, the distance of n_2 from q is $d(q, p_1) - r_1$, where r_1 is the ball radius for p_1, while the distances of both the left child and right child of n_2 are less than this ($d(q, p_2) - r_2$ and 0, respectively). However, this does not violate correctness, since the objects in the subtree rooted at n_2 are still no closer to q than $d(q, p_1) - r_1$, even though the lower bounds based on the pivot p_2 and its ball radius r_2 are smaller. In other words, the objects in the left subtree of n_2 must be somewhere in the white region inside the ball for p_2 (denoted S_1 in the figure) and not in the darkly shaded region, and the objects in the right subtree must be in the lightly shaded region (denoted S_2) and not outside the ball for p_1.

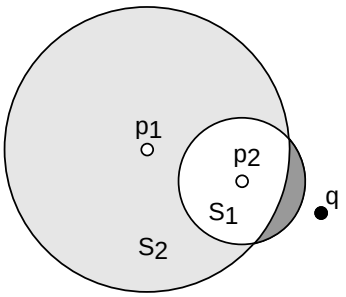

Figure 13: An example of pivots p_1 and p_2 for two nodes n_1 and n_2, respectively, in a vp-tree, where n_2 is the left child of n_1. The regions for the left and right child of n_2 are denoted S_1 and S_2, respectively, in the figure.

4.2.2 The VP^{sb}-Tree

When the vp-tree is constructed, we must compute the distances of an object o from each of the pivots on the path from the root to the leaf containing o. This information is useful, as it can often be used during search to either prune o from the search or include it in the search result without computing its distance. Based on this insight, Yianilos [74] proposed a version of the vp-tree, termed the vp^s-tree,

where we store, for each object (whether it functions as a pivot or is stored in a leaf node), its distance from all ancestral pivots (i.e., those higher in the tree on the path from the root). In the related vp^{sb}-tree, the leaf nodes can store more than one object, thus serving as "buckets". To see how we make use of the distances to ancestral pivots, consider an object o, one of its ancestral pivots p, and the query object q. Given $d(q,p)$ and $d(p,o)$, Lemma 3 allows us to bound the distance between q and o, i.e., $|d(q,p)-d(p,o)| \leq d(q,o) \leq d(q,p)+d(p,o)$. Thus, when performing a range query with radius ε, we can safely discard o if $|d(q,p)-d(p,o)| > \varepsilon$ or directly include it in the result if $d(q,p)+d(p,o) \leq \varepsilon$.

In order to take full advantage of all ancestral pivots, we define two functions $d_{\text{lo}}(q,o)$ and $d_{\text{hi}}(q,o)$ that provide upper and lower bounds on $d(q,o)$, respectively, for any object o:

$$\begin{aligned} d_{\text{lo}}(q,o) &= \max_i\{|d(q,p_i)-d(p_i,o)|\}, \text{ and} \\ d_{\text{hi}}(q,o) &= \min_i\{d(q,p_i)+d(p_i,o)\}, \end{aligned}$$

where $p_1, p_2, \ldots$ are the ancestral pivots of o. Observe that when evaluating these functions, no new distance computations are needed, as $d(q,p_i)$ will have been computed earlier in the query evaluation and $d(p_i,o)$ is stored with o in its vp-tree node. Clearly, a straightforward application of Lemma 3 yields $d_{\text{lo}}(q,o) \leq d(q,o) \leq d_{\text{hi}}(q,o)$.

Thus, we can discard object o from the result of a range query if $d_{\text{lo}}(q,o) > \varepsilon$ and directly include it if $d_{\text{hi}}(q,o) \leq \varepsilon$. This is true even if $|d(q,v)-d(v,o)| \leq \varepsilon$ or $d(q,v)+d(v,o) > \varepsilon$, respectively, for some pivot object v.

Incidentally, there is an interesting connection between d_{lo} and a class of mapping methods termed Lipschitz embeddings (see Section 3.1.3). In particular, the m ancestral pivots $p_1, \ldots, p_m$ of an object o can be regarded as m singleton sets, each of which corresponds to a coordinate axis, forming the mapping $h : S \to \mathbb{R}^m$ where $h(o)$ is the vector $(d(p_i,o))_i$. If we now map q and o according to h, the L_∞ distance $d_M(h(q),h(o))$ between $h(q)$ and $h(o)$ is equal to $d_{\text{lo}}(q,o)$.

How do we make use of these bounds when performing incremental nearest neighbor search? In particular, for determining some number k of neighbors, the algorithm should ideally be able to perform fewer actual distance computations (i.e., based on d) when using the bounds than when not using them. The obvious choice is to modify how objects are treated when encountered in leaf nodes. In other words, if $d_{\text{hi}}(q,o)$ is less than or equal to the distance value of the element at the front of the priority queue, o can be immediately reported as the next nearest object without evaluating $d(q,o)$. Unfortunately, this simple modification is unlikely to improve performance to any significant degree, in the sense of reducing the number of distance computations. To obtain a better solution, we must establish a new search hierarchy that makes use of $d_{\text{lo}}(q,o)$. In particular, in our new search hierarchy, type 0 represents objects, type 1 represents approximate objects, while type 2 represents vp-tree nodes. The distance functions for types 1 and 2 are as follows:

$$\begin{aligned} d_1(q,e_1) &= d_{\text{lo}}(q,o) \text{ and} \\ d_2(q,e_2) &= \max\{d(q,p)-r_{\text{hi}}, r_{\text{lo}}-d(q,p), 0\}. \end{aligned}$$

where o is the object represented by e_1, p is a pivot for the parent of e_2, and $[r_{\text{lo}}, r_{\text{hi}}]$ is the corresponding distance range for objects in the subtree. Similarly, the upper-bound distance functions are defined as

$$\begin{aligned} \hat{d}_1(q,e_1) &= d_{\text{hi}}(q,o) \text{ and} \\ \hat{d}_2(q,e_2) &= d(q,p)+r_{\text{hi}}. \end{aligned}$$

given the same definitions of o, p, and r_{hi}. Observe that elements e_2 of type 2 must carry along the ancestral pivot distances $d(q,p_i)$ to enable computing $d_{\text{lo}}(q,o)$ and $d_{\text{hi}}(q,o)$ for elements of type 1. The correctness of this search hierarchy is guaranteed by the results in Section 4.1. In particular, we have already shown that $d_1(q,e_1) \leq d(q,o) \leq \hat{d}_1(q,e_1)$ follows from Lemma 3 and that $d_2(q,e_2) \leq d(q,o) \leq \hat{d}_2(q,e_2)$ follows from Lemma 4, where o is a descendant of e_1 or e_2, respectively.

Incremental nearest neighbor search using this search hierarchy proceeds as follows (see Section 2.3). Initially, we simply insert the root of the vp-tree as an element of type 2. Subsequently, if the element obtained from the priority queue is of type 0, we report it as the next neighbor, as usual. If the dequeued element is of type 1, we enqueue an element of type 0 for the corresponding object, thus computing its actual distance (alternatively, if $\hat{d}_1(q,e_1)$ is smaller than or equal to the distance value for the new element at the front of the priority queue, we can report the corresponding object immediately without computing the actual distance). If the element is of type 2, it is treated similarly to what was done earlier for the vp-tree, except that elements of type 1 are generated for the data objects in leaf nodes.

The search hierarchy outlined above enables pruning some objects appearing in leaf nodes without computing their distance. In particular, it prunes objects o such that $d_{\text{lo}}(q,o) > d_{\text{lo}}(q,o_{\text{last}})$ where o_{last} is the last object retrieved in incremental nearest neighbor search — that is, the element of type 1 corresponding to o would remain on the priority queue without being processed).

A drawback of the search hierarchy is that such pruning is not possible for pivots in nonleaf nodes, since when processing such a node, we immediately compute $d(q,p)$ for the pivot p. However, if each nonleaf node only contains the ball radius r (and not tight distance ranges $[r_{\text{lo}}, r_{\text{hi}}]$ for each subtree), the distance according to d_2 from q of one of the two elements is always zero. In particular, suppose that e_2 is a search hierarchy element corresponding to a vp-tree nonleaf node n with pivot p, and let e_2' and e_2'' be the elements corresponding to the two children of n. Without the distance range information, either $d_2(q,e_2')$ or $d_2(q,e_2'')$ will always be zero, depending on whether $d(q,p)$ is smaller than or larger than r, the ball radius of n (i.e., whether q is inside or outside the ball centered at p). Thus, one of the child elements of e_2 should be processed next, and we cannot avoid computing the exact value of $d(q,p)$ to determine which one and to process it (since all ancestral pivot distances are needed).

If tight distance bounds are stored in each nonleaf node for each of the two subtrees (or, at least, more information than merely the ball radius), then it is in some cases possible to prune nonleaf nodes from the search without computing the distance from q of the corresponding pivot object. In particular, suppose that n is a nonleaf node with pivot p and that $[r_{1,\text{lo}}, r_{1,\text{hi}}]$ and $[r_{2,\text{lo}}, r_{2,\text{hi}}]$ are the distance bounds for the two subtrees. Based on $d_{\text{lo}}(q,p)$ and $d_{\text{hi}}(q,p)$, we can derive the lower bound $\max\{d_{\text{lo}}(q,p) - r_{1,\text{hi}}, r_{1,\text{lo}} - d_{\text{hi}}(q,p), 0\}$ on the distance between q and any object in the left subtree (i.e., for S_1), and the analogous one for the right subtree. Hence, if the minimum of $d_{\text{lo}}(q,p)$ and the lower bounds for the two subtrees is greater than zero, it may be possible to prune n without computing the distance $d(q,p)$. Based on this observation, we can augment the search hierarchy above with elements e_3 of type 3 denoting "approximate" nonleaf nodes, with distance function

$$d_3(q,e_3) = \min \left\{ \begin{array}{l} d_{\text{lo}}(q,p) \\ \max\{d_{\text{lo}}(q,p) - r_{1,\text{hi}}, r_{1,\text{lo}} - d_{\text{hi}}(q,p), 0\} \\ \max\{d_{\text{lo}}(q,p) - r_{2,\text{hi}}, r_{2,\text{lo}} - d_{\text{hi}}(q,p), 0\} \end{array} \right\},$$

where p is the pivot in the node represented by e_3 and $[r_{1,\text{lo}}, r_{1,\text{hi}}]$ and $[r_{2,\text{lo}}, r_{2,\text{hi}}]$ are the corresponding distance bounds for the subtrees. Thus, when processing an element e_2 of type 2 representing a nonleaf node n, we evaluate $d_3(q,e_3)$ where e_3 represents n. If $d_3(q,e_3) > D_f$ where D_f is the distance of the element at the front of the priority queue, then we enqueue e_3. Otherwise, we cannot avoid computing $d(q,p)$ and must process e_2 in the manner described earlier (i.e., producing elements for the pivot and the two children).

4.2.3 The MVP-Tree

A potential criticism of vp-tree and related metric tree variants is that the fan-out is low (i.e., just 2). As pointed out by Yianilos [74], the vp-tree can gain higher fan-out by splitting S into m subsets of roughly equal size instead of just two, based on $m+1$ bounding values $r_0, \ldots, r_m$ (alternatively, we can let $r_0 = 0$ and $r_m = \infty$). In particular, S is partitioned into $S_1, S_2, \ldots, S_m$ where $S_i = \{o \in S \setminus \{p\} \mid r_{i-1} \leq d(p,o) < r_i\}$. Observe that objects in the subsets lie on spherical "shells" around p. Applying this partitioning process recursively yields an m-ary tree. It is easy to adapt the search hierarchy defined above to this variant. In particular, the various distance functions defined above for search hierarchy elements that represent vp-tree nodes still apply, provided that we set r_{lo} and r_{hi} to the proper values — that is, $r_{\text{lo}} = r_{i-1}$ and $r_{\text{hi}} = r_i$ for the child corresponding to S_i (unless tighter bounds are maintained).

Another variant of vp-trees that achieves a higher fan-out, termed the mvp-tree, was suggested by Bozkaya and Ozsoyoglu [7, 8]. Each node in the mvp-tree is essentially equivalent to the result of collapsing the nodes at several levels of a vp-tree. There is one crucial difference between the mvp-tree and the result of such collapsing: only one pivot is used for each level inside an mvp-tree node (although the number of different ball radius values is unchanged). Thus, in an mvp-tree that corresponds to collapsing a vp-tree over every two levels, two pivots are used in each mvp-tree node with three ball radius values. An example of the top-level partitioning for such an mvp-tree is shown in Figure 14.

The motivation for the mvp-tree is that fewer distance computations are needed for pivots during search since there are fewer of them (e.g., for an mvp-tree node with two pivots, three pivots would be needed in the corresponding vp-tree). Observe that some subsets are partitioned using pivots that are not members of the sets, which does not occur in the vp-tree (e.g., p_2 is used to partition the subset inside the ball around p_1 in Figure 14a). Bozkaya and Ozsoyoglu [7, 8] suggest using multiple partitions for each pivot, as discussed above. Hence, with k pivots per node and m partitions per pivot, the fan-out of the nonleaf nodes is m^k. Furthermore, they propose storing, for each data object in a leaf node, the distances to some maximum number n of ancestral pivots (by setting a maximum n on the number of ancestral pivots, the physical size of all nodes can be fixed). This is analogous to the use of ancestral pivots in the vp$^{\text{sb}}$-tree, as described above, except that this distance information is only maintained in leaf nodes in the mvp-tree. Another minor departure from the vp-tree that enables additional pruning to take place is that each leaf node in the mvp-tree also contains k pivots (whereas pivots are not used in leaf nodes in the vp-tree). In addition, the distances between these pivots and the data objects are stored in the node (a version of the mvp-tree in which pivots are not used in leaves is also considered in [8]). Thus, the leaf node pivots essentially function like the ancestral pivots.

Clearly, a search hierarchy similar to that for the vp-tree can be defined for the mvp-tree, with the addition of approximate object elements as was done for the vp$^{\text{sb}}$-tree (these are generated for leaf nodes based on distances to ancestral pivots). The question is whether to treat the k pivots in each mvp-tree node separately. In other words, should we design the search hierarchy so that each mvp-tree node is represented by a single type of element, or should we introduce k types for each node, one of which represents the node itself, while the others represent partial decompositions based on some of the pivots in the node? For the second alternative, m first-pivot partial node elements are generated from each node element based on the first pivot, m second-pivot partial node elements are generated from each first-pivot partial node based on the second pivot, etc. Choosing this second alternative would only make sense if we could sometimes prune all m partial node elements without computing the actual distance to the second pivot of the corresponding mvp-tree node. Unfortunately, because of the way the mvp-tree is defined, this is not the case, since at least one of the m partial node elements generated for a node n must be processed before the next nearest neighbor can be reported. Similar reasoning applies here as that for pruning of nonleaf nodes in Section 4.2.2. In particular, if only ball radii are stored for partitions (i.e., not distance ranges for each partition), the distance value for one of the two partitions based on the first pivot will

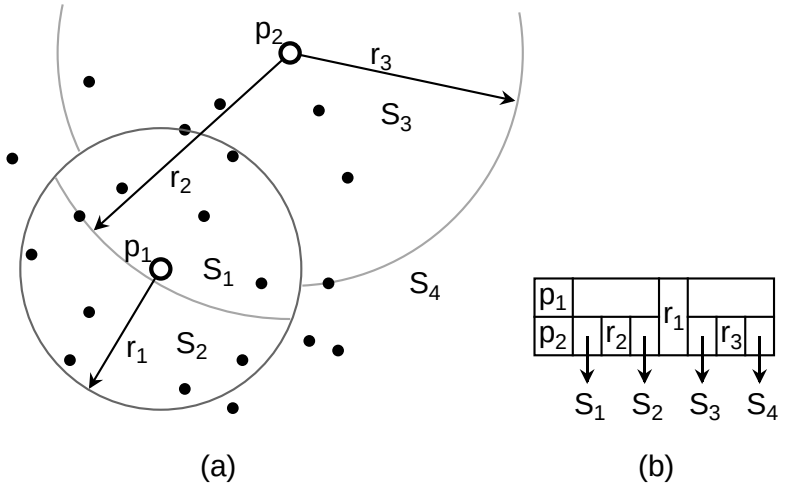

Figure 14: (a) Possible top-level partitionings of a set of objects (depicted as two-dimensional points) in an mvp-tree where two pivots are used in each node, and (b) a depiction of the corresponding mvp-tree node. The second pivot, p_2, partitions the inside of the ball for p_1 into subsets S_1 and S_2, and the outside of the ball into subsets S_3 and S_4.

always be zero. Augmenting mvp-tree nonleaf nodes with tighter bounds for the distance values in the two partitions of the first pivot might make it worthwhile to use partial node elements, but this is at the cost of more storage.

4.2.4 Other Methods Related to Ball Partitioning

A number of additional proposals of search structures that employ some form of ball partitioning have been made. Below, we summarize some of these ball partitioning methods. Unless otherwise mentioned, incremental nearest neighbor search can be performed in these structures using search hierarchies that are very similar to those described in Sections 4.2.1–4.2.3.

The vp-tree, one of the most common instances of ball partitioning, is actually a special case of what Knuth terms a *post-office tree* whose proposal he attributes to Bruce McNutt in 1972 [40, p. 563]. The difference is that each node in the post-office tree is a vp-tree node (p,r) with the addition of a tolerance δ which is associated with the radius r of the ball centered at p. In particular, given a value of δ, once pivot p and radius r have been chosen, the remaining objects are subdivided into two subsets S_1 and S_2 as follows:

$$S_1 = \{o \in S \setminus \{p\} \mid d(p,o) \leq r + \delta\}$$
$$S_2 = \{o \in S \setminus \{p\} \mid d(p,o) \geq r - \delta\}$$

Thus the objects in S_1 are inside the ball of radius $r+\delta$, while the objects in S_2 are outside a ball of radius $r-\delta$. Of course, some objects lie both in S_1 and S_2 — that is all objects o where $|d(o,p) - r| \leq \delta$[10].

Among the earliest published work on distance-based indexing was that of Burkhard and Keller [11]. One of the three structures they proposed employs ball partitioning. However, the distance function was assumed to be discrete, so that only a few different distance values are possible, say m. At the top level, some distinguished object $p \in S$ is chosen, and the remaining objects are partitioned into m subsets

[10]The idea of a loose partition so that the sons of a node are not disjoint is also used in the os-tree [44, 45], KD2-tree [54, 55], spatial k-d tree [53], and hybrid tree [12, 13].

$S_1, S_2, \ldots, S_m$ based on distance value. Applying this process recursively yields an m-ary tree. Clearly, p has the same role as a pivot in the vp-tree, and the result of the partitioning process is analogous to that of an m-ary vp-tree (see Section 4.2.3). In fact, as pointed out by Chávez et al. [15], a natural adaptation of Burkhard and Keller's technique to continuous distance functions is to choose the partition values $r_0, r_1, \ldots, r_m$ such that the objects are partitioned into m equiwidth shells around p. In other words, r_0 and r_m are chosen as the minimum and maximum distances, respectively, between p and objects in $S \setminus \{p\}$, and $r_i = \frac{i}{m}(r_m - r_0) + r_0$ for $i = 1, \ldots, m-1$.

Baeza-Yates et al. [3] proposed a variant of Burkhard and Keller's approach that they termed the *fixed-queries tree*. In this variant, all nodes at the same level in the tree use the same pivot, and the pivot objects also appear as data objects in leaf nodes of the tree[11] (unlike the vp-tree or Burkhard and Keller's approach). The rationale for using just one pivot per level is the same as in the mvp-tree — that is, so that fewer distance computations are needed during search, since only one distance computation is needed for visiting all nodes at a given level (as is the case when the search backtracks). The drawback is that the quality of the partitioning may suffer as a result of using fewer pivots. Fixed-height variants of this idea were also proposed, where all leaf nodes are at the same level. Thus, some internal nodes have only one child node (in cases where the node would otherwise have been a leaf node), and leaf nodes may contain arbitrary numbers of objects. Furthermore, each object has the same number of ancestral pivots, and thus requires the same number of distance computations when constructing the tree. This insight led to the proposal of the *fixed query array* [14], which is essentially a compact representation of the distances in a fixed-height fixed-queries tree in the form of an array of bit strings. In the fixed query array, movements in the equivalent fixed-height fixed-queries tree are simulated with binary search.

Yianilos [73] proposed a variant of vp-trees termed the *excluded middle vantage point forest* that is intended for radius-limited nearest neighbor search, i.e., where the nearest neighbor is restricted to be within some radius $r*$ of the query object. This method is based on the insight that most of the complexity of performing search in methods based on binary partitioning, such as the vp-tree, is due to query objects that lie close to the partition values, thereby causing both partitions to be processed. For example, in the vp-tree, these are objects q for which $d(q,p)$ is close to r, the partitioning value for a pivot p. The proposed solution is to exclude all data objects whose distances from a pivot are within $r*$ of the partition value (i.e., the ball radius). This process is applied to all pivots in the tree and a new tree is built recursively for the set of all excluded objects. Thus the final result is a forest of trees. Since the width of all exclusion regions is at least $2r*$, nearest neighbor search limited to a search region of radius $r*$ can be performed with no backtracking, but this is at the price of having to search all the trees in the forest. The fact that no backtracking is needed allows determining a worst-case bound on the search cost, based on the heights of the trees in the forest. Unfortunately, the method appears to provide good performance only for very small values of $r*$ [73], which is of limited value in most similarity search applications.

4.3 Generalized Hyperplane Partitioning Methods

4.3.1 The GH-Tree

Uhlmann [66] defined a metric tree using generalized hyperplane partitioning, which has been termed a gh-tree by later authors [9, 8, 27]. Instead of picking just one object for partitioning as in the vp-tree, this method picks two pivots p_1 and p_2 (e.g., the objects farthest from each other) and splits the set of remaining objects based on the closest pivot (see Figure 6b):

$$S_1 \quad = \quad \{o \in S \setminus \{p_1, p_2\} \mid d(p_1, o) \leq d(p_2, o)\}, \text{ and}$$

[11] When performing incremental nearest neighbor search with structures where pivot objects also appear in leaf nodes, care must be taken that such objects are only inserted once into the priority queue (either by inserting objects only as they are encountered in leaf nodes, or by somehow detecting the fact that an object has been inserted earlier).

$$S_2 \quad = \quad \{o \in S \setminus \{p_1, p_2\} \mid d(p_2, o) < d(p_1, o)\}.$$

In other words, the objects in S_1 are closer to p_1 than to p_2 (or equidistant from both), and the objects in S_2 are closer to p_2 than to p_1. This rule is applied recursively, resulting in a binary tree where the left child of a nonleaf node corresponds to S_1 and the right to S_2. This rule can be restated as stipulating that S_1 contains all objects o such that $d(p_1, o) - d(p_2, o) \leq 0$. Clearly, the two subsets S_1 and S_2 can be very different in size. Uhlmann [66] actually suggested partitioning based on a median value m, so that $d(p_1, o) - d(p_2, o) \leq m$ applies to roughly half the objects in S. For simplicity, we assume below that m is fixed at 0; the discussion is easily generalized to other values.

The term "generalized hyperplane partitioning" is derived from the fact that if the objects are points in an n-dimensional Euclidean space, the partitioning is equivalent to partitioning based on an $(n-1)$--dimensional hyperplane like that used in a k-d tree (in a k-d tree, however, the partitioning planes are axis-aligned). This hyperplane is the set of all points o that satisfy $d(p_1, o) = d(p_2, o)$. Consider how to compute a lower bound on the distance from a query object q to an object in one of the partitions, say, that for S_1. If q is in the partition (i.e., is closer to p_1), the lower bound is clearly zero. Otherwise, the lower bound is equal to the distance from q to the partitioning hyperplane, which is easy to compute in Euclidean spaces. For arbitrary metric spaces, however, we cannot form a direct representation of the "generalized hyperplane" that divides the two partitions, since we assume that the interobject distances are the only available information.

Fortunately, even given the limited information available in the gh-tree, Lemma 6 shows that it is possible to derive a lower bound on the distance from q to some object in a partition[12] (though an upper bound cannot be determined, since objects can be arbitrarily far from p_1 and p_2). In particular, for a range query with query radius ε, the left subtree for a pivot p must be visited if and only if $\frac{d(q,p_1)-d(q,p_2)}{2} \leq \varepsilon$ and the right one must be visited if and only if $\frac{d(q,p_2)-d(q,p_1)}{2} \leq \varepsilon$. Similarly, we can define a search hierarchy for the gh-tree, with the distance of an element of type 1, representing a node, defined as follows:

$$d_1(q, e_1) = \begin{cases} \max\left\{\frac{d(q,p_1)-d(q,p_2)}{2}, 0\right\}, & \text{if } e_1 \text{ is a left child,} \\ \max\left\{\frac{d(q,p_2)-d(q,p_1)}{2}, 0\right\}, & \text{otherwise.} \end{cases}$$

4.3.2 GNAT

GNAT [9] (Geometric Near-neighbor Access Tree) is a generalization of the gh-tree, where more than two pivots (termed *split points* in [9]) may be chosen to partition the data set at each node. In particular, given a set of pivots $P = \{p_1, \ldots, p_m\}$, we split S into $S_1, \ldots, S_m$ based on which of the objects in P is the closest. In other words, for any object $o \in S \setminus P$, o is a member of S_i if $d(p_i, o) \leq d(p_j, o)$ for all $j = 1, \ldots, m$. In case of ties, i is the lowest index among the ones that participate in the tie. Thus, applying such a partitioning process recursively yields an m-ary tree. Brin [9] left the value of m as a parameter, and also suggested a way to adaptively choose a different number of pivots at each node, based on the cardinalities of the partition sets. The method Brin [9] describes for choosing the pivot objects is based on a philosophy similar to that of Yianilos [74] for the vp-tree (and also suggested by others [8, 62]). In particular, initially, randomly pick $3m$ candidate pivot objects from S. Next, pick the first pivot object at random from the candidates, pick as the second the candidate farthest away from the first one, pick as the third the candidate farthest away from the first two, etc.

In addition to pivots and child pointers, the nodes in GNAT also store information about the ranges of distances between the pivots and objects in the subtrees, which enables more pruning during search. In

[12] The bound in the lemma is clearly much weaker than would be obtained in a Euclidean space by using the hyperplane directly.

particular, for each pair of pivots p_i and p_j in a node n, we store the range $[r_{\text{lo}}^{(i,j)}, r_{\text{hi}}^{(i,j)}]$ of $d(p_i, o)$ over all objects $o \in S_j \cup \{p_j\}$ (i.e., $r_{\text{lo}}^{(i,j)} = \min_{o \in S_j \cup \{p_j\}} \{d(p_i, o)\}$ and $r_{\text{hi}}^{(i,j)} = \max_{o \in S_j \cup \{p_j\}} \{d(p_i, o)\}$). Although not mentioned by Brin [9], it may also be advantageous to store the range $[r_{\text{lo}}^{(j,j)}, r_{\text{hi}}^{(j,j)}]$ for $d(p_j, o)$ over all m objects $o \in S_j$, bringing the total number of ranges to m^2 (as there are $m \cdot (m-1) + m$ ranges altogether). Figure 15 illustrates the distance bounds for two pivots p_i and p_j, where the dots clustered around p_j depict the objects in S_j.

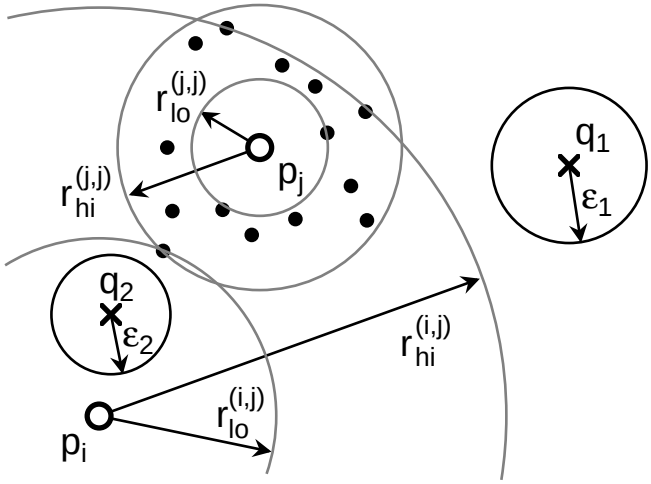

Figure 15: Depiction of the bounds for two pivots p_i and p_j in a GNAT. For the sample query object q with query radius ε shown in the figure, p_j and its subset would be eliminated from the search since $d(q, p_i) - \varepsilon > r_{\text{hi}}^{(i,j)}$.

If the objects are points in an n-dimensional Euclidean space, the objects in S_i are exactly the objects in $S \setminus P$ that fall into the Voronoi cell with p_i as a site. For Euclidean spaces, it is relatively straightforward to directly represent the Voronoi cells (although this becomes increasingly impractical as the dimensionality grows), and thus compute a lower bound on the distance from a query point to the points inside a given cell (i.e., based on the geometry of the cell). Unfortunately, for arbitrary metric spaces, computing a lower bound in this way is not feasible since, as we saw for the gh-tree, we do not have a direct representation of the "generalized Voronoi cells" formed by the pivots (termed *Dirichlet domains* in [9]). Clearly, we could simply apply Lemma 6, as we did for the gh-tree, which would yield the lower bound $(d(q, p_i) - d(q, p_j))/2$ on $d(q, o)$ for an object o in S_i (i.e., o is closer to p_i than to p_j), where p_j is the object in P closest to q (since this choice of p_j maximizes the lower bound). However, as shown below, tighter bounds can be obtained by using the distance ranges $[r_{\text{lo}}^{(i,j)}, r_{\text{hi}}^{(i,j)}]$ (based on Lemma 4), thus achieving better search performance. We can think of the distance bounds as effectively constraining the "shape" of the region represented by the child nodes so as to approximate the corresponding Voronoi cells. For example, in Euclidean spaces, the distance bounds represent spherical shells around the pivots, and the Voronoi cell for p_i is approximated by the intersection of the shells for all the pivots p_j. Of course, two approximate Voronoi cell regions may intersect each other, unlike actual Voronoi cells (which at most share a boundary).

The range query algorithm for GNAT described by Brin [9], for a query object q and query radius ε, proceeds in a depth-first manner. When processing a node n, the distances between q and the pivots are computed one by one, gradually eliminating subtrees when possible. The children of n are visited only after computing the distances of all pivots that could not be eliminated using the distances of pivots that were considered earlier. In particular, the process is initiated with the set P consisting of all pivots for n.

At each step, we remove one of the objects $p_i \in P$ whose distance from q has not been computed, and compute $d(q, p_i)$. If $d(q, p_i) \leq \varepsilon$, we add p_i to the query result. Next, for all $p_j \in P$, we remove p_j from P if $d(q, p_i) - \varepsilon > r_{\text{hi}}^{(i,j)}$ or $d(q, p_i) + \varepsilon < r_{\text{lo}}^{(i,j)}$ (or, equivalently, if $\max\{d(q, p_i) - r_{\text{hi}}^{(i,j)}, r_{\text{lo}}^{(i,j)} - d(q, p_i)\} > \varepsilon$, based on Lemma 4). Figure 15 depicts two sample query objects q_1 and q_2 and associated query radii ε_1 and ε_2, respectively, both of which would cause p_j to be removed from P since $d(q_1, p_i) - \varepsilon_1 > r_{\text{hi}}^{(i,j)}$ and $d(q_2, p_i) + \varepsilon_2 < r_{\text{lo}}^{(i,j)}$. After the distances from q for all the pivots in P have been computed (or P becomes empty), the children of n that correspond to the remaining pivots in P are searched recursively. Notice that a pivot p_j may be discarded from P before its distance from q is computed.

A naive definition of a search hierarchy for incremental nearest neighbor search is obtained by using only two types of elements, for objects and nodes, and deriving the lower-bound distance function based on all the distance ranges. In other words, for a node n' corresponding to pivot p_j in its parent n, we can derive the following lower bound (based on Lemmas 4 and 6) on the distance $d(q, o)$, where o is in the subtree rooted at n' (i.e., $o \in S_j$):

$$\max_{i \in \{1,\ldots,m\}} \{d(q, p_i) - r_{\text{hi}}^{(i,j)}, r_{\text{lo}}^{(i,j)} - d(q, p_i), \frac{d(q, p_j) - d(q, p_i)}{2}, 0\}.$$

Clearly, evaluating this lower bound requires computing the distances between q and all the pivots in n. This is a major drawback, as it means that range search on this search hierarchy would generally compute more distances than the range query algorithm described above, since the latter is sometimes able to discard pivots from consideration without computing their distances from q. The challenge we face in defining a more effective search hierarchy is that with incremental nearest neighbor search, we have no ε to base the pruning on. In particular, regardless of k, the number of nearest neighbors obtained, our goal is to perform no more distance computations than the range query algorithm would with $\varepsilon = d(q, o_k)$, where o_k is the k^{th} nearest neighbor.

To approach this goal, the search hierarchy must essentially embody gradual computation of the distances between q and the pivots in a node. We propose to do this in the following way. Elements of type 0 and 1 represent objects and GNAT nodes, as usual, but elements of type 2 represent partially processed GNAT nodes. For each element e_2 of type 2 representing a node n, we maintain information about which pivots and child nodes have already been enqueued, as well as bounds on the distances between q and the remaining pivots and child nodes (i.e., the objects in the subtrees rooted at the child nodes). In particular, if the child node n' corresponding to a pivot p_j has not yet been enqueued, we define

$$\begin{aligned} d_{\text{lo}}(q, T_j) &= \max_{p_i \in P'} \{d(q, p_i) - r_{\text{hi}}^{(i,j)}, r_{\text{lo}}^{(i,j)} - d(q, p_i), \frac{d(q, p_j) - d(q, p_i)}{2}, 0\}, \text{ and} \\ d_{\text{hi}}(q, T_j) &= \min_{p_i \in P'} \{d(q, p_i) + r_{\text{hi}}^{(i,j)}\}, \end{aligned}$$

where P' is the set of pivots of n whose distances from q have been computed, S_j is the set of objects in the subtree rooted at n', and $T_j = S_j$ if $p_j \in P'$ or $T_j = S_j \cup \{p_j\}$, otherwise. The values of $d_{\text{lo}}(q, T_j)$ and $d_{\text{hi}}(q, T_j)$ are computed incrementally as each new pivot is added to P' and its distance from q computed. Given these definitions, the distances for e_2 are defined as

$$\begin{aligned} d_2(q, e_2) &= \min_{T_j \in R} \{d_{\text{lo}}(q, T_j)\}, \text{ and} \\ \hat{d_2}(q, e_2) &= \max_{T_j \in R} \{d_{\text{hi}}(q, T_j)\}, \end{aligned}$$

where R is the set of sets T_j for which the child node corresponding to pivot p_j has not been enqueued. Similarly, the distances for an element e_1 of type 1 for a node n' are defined in terms of the current values

of $d_{\text{lo}}(q, T_j)$ and $d_{\text{hi}}(q, T_j)$ when e_1 was created, where p_j is the corresponding pivot in n, the parent of n'. The correctness of incremental nearest neighbor search using this scheme should be clear (by Lemmas 4 and 6).

When performing incremental nearest neighbor search using this hierarchy, it is easy to see what actions to take when processing elements of type 0 or 1. In particular, for an element of type 0 we simply report the corresponding object as the next nearest neighbor. For an element e_1 of type 1 corresponding to a node n, we effectively generate a new element e_2 of type 2 for the node, where $d_{\text{lo}}(q, T_j)$ and $d_{\text{hi}}(q, T_j)$ for each pivot p_j are initialized to 0 and ∞, respectively (thus, since $d_2(q, e_2) = 0$, e_2 will get processed next). It is less clear which of two actions to perform when processing an element e_2 of type 2, representing a partially processed node n. The two possible actions are: 1) compute the distance from q to a new pivot p_i of n (thereby generating an element of type 0 for p_i and updating $d_{\text{lo}}(q, T_j)$ and $d_{\text{hi}}(q, T_j)$ based on $d(q, p_i)$ for each $T_j \in R$), or 2) generate an element of type 1 for the child corresponding to pivot p_i of n (thereby removing T_i from R). For both actions, $d_2(q, e_2') \geq d_2(q, e_2)$ (to see why, observe that action 1 may increase some $d_{\text{lo}}(q, T_j)$ for $T_j \in R$ as it is based on a maximum and p_i has been added to P', while action 2 removes T_i from R, so the minimum over $d_{\text{lo}}(q, T_j)$ can only increase), where e_2' represents the element that results from performing the action on e_2. Clearly, if the distances from q of all the pivots of n have been computed, we can perform action 2 repeatedly until we have exhausted R, thereby generating elements of type 1 for all remaining children of n. However, if this is the only instance in which we chose action 2 over action 1, we are not much better off than we were with the naive search hierarchy definition, since the only instance in which we will not compute the distances between q and all the pivots of n before reporting the next nearest neighbor o is if o happens to be one of the pivots of n.

A simple heuristic that overcomes this drawback is one where we choose the action based on the set $T_j \in R$ with the lowest value of $d_{\text{lo}}(q, T_j)$. In particular, we choose action 1 if $d(q, p_j)$ has not been computed, where p_j is the pivot that corresponds to T_j, and action 2 otherwise. Letting o_k be the k^{th} nearest neighbor, consider a node n that is visited by the GNAT range query algorithm using $\varepsilon = d(q, o_k)$. The range query algorithm avoids computing the distances of pivots p_j whenever $d_{\text{lo}}(q, T_j) > d(q, o_k)$ before p_j is picked. The same is true in incremental nearest neighbor search based on the heuristic outlined above, since $d(q, p_j)$ is only computed if $d_{\text{lo}}(q, T_j) \leq d_{\text{lo}}(q, T_i)$ for any other pivot p_i whose corresponding child has not yet been enqueued — that is, if $d_{\text{lo}}(q, T_j) > d(q, o_k)$, then $d(q, o_k) > d_{\text{lo}}(q, T_i)$. Unfortunately, incremental nearest neighbor search may visit a child node n' of n, say that corresponding to pivot p_i, that would not be visited by the range query algorithm, since n' would only be visited by the range query algorithm after having computed the distances between q and all pivots in n that cannot be eliminated. Thus, we cannot quite attain our goal of always achieving the same number of distance computations as the range query algorithm. However, in practice, the range query algorithm cannot be used directly for nearest neighbor queries, since the value of $d(q, o_k)$ would not be known a priori. For nearest neighbor search, we cannot expect to complete the pruning of all pivots in a node before visiting the children, so some heuristic must be used to determine when to visit children.

4.3.3 Other Methods Related to Generalized Hyperplane Partitioning

A structure very similar to the gh-tree, termed the *monotonous* (sic) *bisector tree* (abbreviated below as *mb-tree*), was proposed by Noltemeier et al. [52] (and used by Bugnion et al. [10]). The intended application was to point data using Minkowski metrics (and an extension, the *mb*-tree*, was defined for complex objects, such as lines and polygons), but the mb-tree is generally applicable to arbitrary metrics. In the mb-tree, one of the two pivots in each nonleaf node n, except for the root, is inherited from its parent node (i.e., of the two pivots in the parent of n, the one that is closer to each object in the subtree rooted at n). Since this leads to fewer pivot objects, it can be expected to reduce the number of distance compu-

tations during search (provided the distances of pivot objects are propagated downward during search), at the possible cost of worse partitioning. Furthermore, the radius of the ball around the pivots (i.e., the maximum distance to objects in the corresponding subtree) is also stored, which enables more pruning. The TLAESA method of Micó et al. [47] also uses an mb-tree-like search structure in conjunction with a distance matrix to provide lower bounds on the distance from q to the pivot objects during search (see Section 4.6.3 for more details).

The gh-tree and GNAT (as well as the M-tree, described in Section 4.4) can be considered to be special cases of a general class of hierarchical clustering methods, as described by Burkhard and Keller [11] and Fukunaga and Narendra [28]. Using the description given by Burkhard and Keller [11], a set S of objects is clustered into m subsets $S_1, S_2, \ldots, S_m$ using some criterion. Then a pivot object p_i is chosen for each S_i and the radius $r_i = \max_{o \in S_i}\{d(p_i, o)\}$ is computed[13]. This process is applied recursively to each S_i, possibly with a different number m of clusters each time. Observe that for performing search (e.g., range search), a lower bound on the distances from a query object q to all objects in S_i can be derived based on p_i and r_i according to Lemma 4, as was done for the vp-tree in Section 4.2.1 (i.e., letting $r_{\text{lo}} = 0$ and $r_{\text{hi}} = r_i$). Besides the above general formulation, Burkhard and Keller also described a specific method of clustering, where each cluster is a *clique*, which they define to be a set of objects R such that the greatest distance between any two objects in R is no more than k[14]. The clique property was found to reduce the number of distance computations and allow more pruning during search [11], at the price of high preprocessing cost (for determining the cliques).

Fukunaga and Narendra [28] outlined a hierarchical clustering approach similar to that of Burkhard and Keller [11], with the additional property that the distances from a pivot object are stored for all data objects in the leaf nodes, allowing for greater pruning[15]. However, some aspects of their process of constructing the search structure depend on properties of vector spaces. In particular, Fukunaga and Narendra assume that each pivot is chosen as the centroid of the cluster (i.e., the point obtained by averaging the coordinate values of the points in the cluster), and the clustering method used in their experiments assumes vector data (in contrast to just making use of the interobject distance values). Nevertheless, their nearest neighbor algorithm is valid for general clustering methods (see Section 5.2).

4.4 The M-Tree

The distance-based indexing methods described in Sections 4.2 and 4.3 are either static, unbalanced, or both. Hence they are unsuitable for dynamic situations involving large amounts of data, where a disk--based structure is needed. The M-tree [19] is a distance-based indexing method designed to address this deficiency. Its design goal was to combine a dynamic, balanced index structure similar to the R-tree (see Section 2.1) with the capabilities of static distance-based indexes.

In the M-tree, as in the R-tree, all the objects being indexed are referenced in the leaf nodes[16], while an entry in a nonleaf node stores a pointer to a node at the next lower level along with summary information about the objects in the subtree being pointed at. Recall that in an R-tree, the summary information consisted of minimum bounding rectangles for all the objects in the subtree. For arbitrary metric spaces, we cannot explicitly form the "regions" that enclose a set of objects in the same manner. Instead, in the M-tree, "balls" around pivot objects (termed *routing objects* in [19]) serve the same role as the minimum bounding rectangles in the R-tree. Clearly, the pivots in the M-tree have a function similar to that of the

[13] GNAT [9] maintains more comprehensive distance information in each node.

[14] If we consider the objects to be nodes in a graph, with edges between objects whose distance is no more than k, a graph-theoretic clique in this graph corresponds to Burkhard and Keller's definition of a clique.

[15] This idea was later adopted and extended in the vp$^{\text{sb}}$-tree and mvp-tree; see Sections 4.2.2 and 4.2.3.

[16] The objects can either be stored directly in the leaf nodes, or externally to the M-tree, with object IDs stored in the leaf nodes.

pivots in GNAT (see Section 4.3). However, unlike GNAT, all objects in S are stored in the leaf nodes of the M-tree, so an object may be referenced multiple times in the tree (once in a leaf node, and as a pivot in one or more nonleaf nodes). For an object o in the subtree of a node n, the pivot p of that subtree is not always the one closest to o among all the pivots in n (i.e., we may have $d(p,o) > d(p',o)$ for some other pivot p' in n). In addition to this summary information, the entries in M-tree nodes also contain distance values that can aid in pruning during search, as is done in the vp^{sb}-tree (see Section 4.2.2).

More precisely, for a nonleaf node n, the entries are (p,r,D,T), where p is a pivot, r is the corresponding *covering radius*, D is a distance value (defined below), and T is a reference to a child node of n. For all objects o in the subtree rooted at T, we have $d(p,o) \leq r$. For each non-root node, let *parent object* denote its associated pivot, i.e., the pivot in the entry pointing to it in its parent. The distance value stored in D is the distance $d(p,p')$ between p and the parent object p' of n. As we shall see, these parent distances allow more pruning during search than would otherwise be possible. Similarly, for a leaf node n, the entries consist of (o,D), where o is a data object and D is the distance between o and the parent object of n. Clearly, the root has no parent, so $D = \infty$ for all the entries in the root. Observe that the covering radius for a nonleaf entry is not necessarily the minimum radius for the objects in the corresponding subtree (except when the M-tree is bulkloaded [18]).

Being a dynamic structure, the M-tree can be built gradually as new data arrives [19]. The insertion procedure first "routes" a new data object to a leaf node n, for each nonleaf node on the path, picking a child node that "best matches" the data object, based on heuristics. For example, a heuristic might first look for a pivot object whose "ball" includes the data object, and pick the one closest to the data object if there is more than one such pivot. The insertion into n may cause overflow, causing n to be split and a new pivot to be selected. Thus, overflow may cascade up to the root, and the tree actually grows in a bottom-up fashion. Ciaccia et al. [19] considered a number of heuristics for choosing the child node to route an object into and for splitting overflowing nodes. Bulk-loading strategies [18] have also been developed for use when an M-tree must be built for an existing set of data objects. An example of root node partitioning in an M-tree for a set of objects is shown in Figure 16, where we have three pivot objects, p_1, p_2, and p_3. Notice that the regions of some of the three subtrees overlap. This may give rise to a situation where an object can be inserted into more than one subtree, such as the object marked o, which can be inserted into the subtree of either p_1 or p_3.

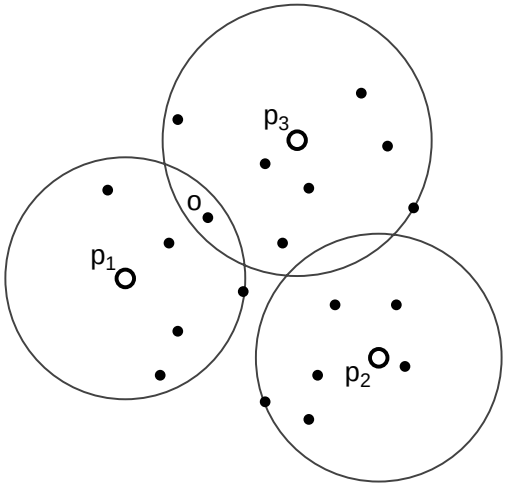

Figure 16: Possible top-level partitionings of a set of objects (depicted as two-dimensional points) in an M-tree. Objects that fall into more than one "ball", like o, can go into any of the corresponding subtrees.

Traina et al. [64] speed up the M-tree node insertion and node splitting algorithms while also improving the storage utilization. This is achieved, in part, by applying a post-processing step (termed the *Slim-down* algorithm) that attempts to reduce the overlap among node regions. They use the term *Slim-tree* to describe their method. An empirical study showed that these modifications led to a reduction in the number of disk accesses [64].

Range queries for query object q and query radius ε can be performed on the M-tree with a straightforward depth-first traversal, initiated at the root. Let n be a node that is being visited, and let p' be its parent pivot, i.e., p' is the pivot for the entry in n's parent that points to n. In order to exploit the parent distance D stored in the entries of n (i.e., to avoid as much as possible the computation of the distances from q to the pivots p stored in the entries of n), the value of $d(q,p')$ must be propagated downward in the depth-first traversal as n is visited (since the root has no parent, we use $d(q,p') = \infty$ when processing the root, and assume that $\infty - \infty$ evaluates to 0). Assume that n is a nonleaf node. We consider each entry (p,r,D,T) in turn. There are two cases:

1. If $|d(q,p') - D| - r > \varepsilon$, then the subtree pointed at by T need not be traversed and thus the entry is pruned. This criterion is based on the fact that $|d(q,p') - D| - r$ is a lower bound on the distance of any object in the subtree pointed at by T. Thus, if the lower bound is greater than ε, then no object in this subtree can be in the range. The lower bound can be established by making use of Lemmas 3 and 5. Lemma 3 yields a lower bound from q to any of the pivots (e.g., p) in node n. In this case, p and p' play the roles of o and p, respectively, in the Lemma which stipulates that $|d(q,p') - d(p',p)| = |d(q,p') - D| \leq d(q,p)$. The upper bound on the distance from q to any of the pivots (e.g., p) in node n is ∞. The distance from pivot p to any of the objects in the corresponding subtree T lies between 0 and r. We now apply Lemma 5 to obtain a lower bound on the distance from q to any object o in the subtree pointed at by T — that is, $r_{lo} = 0$, $r_{\text{hi}} = r$, $s_{\text{lo}} = |d(q,p') - D|$, and $s_{\text{hi}} = \infty$ — yielding $|d(q,p') - D| - r \leq d(q,o)$.

2. Otherwise, $|d(q,p') - D| - r \leq \varepsilon$. In this case, we can no longer avoid computing $d(q,p)$. However, having computed $d(q,p)$, we can still avoid visiting the node pointed at by T if the lower bound on the distance from q to any object o in T is greater than ε. This is the case if $d(q,p) - r > \varepsilon$ and is a direct result of applying Lemma 4 noting that the distance from p to o lies between 0 and r.

Leaf nodes are processed in a similar way: For each entry (o,D) in n with parent pivot p', we first check if $|d(q,p') - D| \leq \varepsilon$ (since we know from Lemma 3 that $|d(q,p') - d(p',o)| = |d(q,p') - D| \leq d(q,o)$, so if $\varepsilon < |d(q,p') - D| \leq d(q,o)$, we can immediately discard o without computing its distance), and only for such entries compute $d(q,o)$ and check whether $d(q,o) \leq \varepsilon$. Observe that once again we see that the parent distances sometimes allow us to prune node entries from the search based on the query radius ε, without computing the actual distances of the corresponding objects.

Finding nearest neighbors is more complicated than range search. For k-nearest neighbor search, Ciaccia et al. [19] propose using the distance of the farthest candidate k^{th} nearest neighbor in place of ε in the pruning conditions[17]. This technique is discussed further in Section 5.2, while below we describe the more general incremental approach where the number of desired neighbors is not known in advance. In particular, for incremental nearest neighbor search, not knowing the number of desired neighbors in advance means that the search radius is unbounded and thus the pruning condition used by Ciaccia et al. [19] is inapplicable. To overcome this dilemma, we introducing two new element types corresponding to approximate objects and approximate nodes to the search hierarchy. These new element types serve the same role as bounding rectangles in the case of an R-tree. Thus they provide a simple way to order

[17] This is a standard approach for extending a nearest neighbor algorithm to a k-nearest neighbor algorithm. For example, it was adopted by Seidl and Kriegel [60], as described in Section 5.1.

the subsequent processing of elements of both a leaf and nonleaf node without having to compute the actual distances of these elements from the query object. As we will see in Section 5.2, this method can also be applied to obtain a more efficient solution to the k-nearest neighbor problem than the algorithm of Ciaccia et al. [19].

The search hierarchy for the M-tree is defined in a manner similar to that for the vp^{sb}-tree in Section 4.2.2. In particular, we define four types of elements. Type 0 represents objects, type 1 represents approximate objects, type 2 represents nodes, and type 3 represents approximate nodes. Elements of type 1 and 3 are generated as a result of processing leaf nodes and nonleaf nodes, respectively. In particular, when processing a leaf (nonleaf) node n (i.e., when it reaches the front of the priority queue as an element of type 2), an element of type 1 (3) is generated from each of the entries in n. An element of type 0 is generated as a result of processing an element of type 1, and, similarly, each element of type 2 derives from an element of type 3. The distance functions for elements of type 1 through 3 are defined as follows:

$$\begin{aligned} d_1(q,e_1) &= \max\{|d(q,p')-D|,0\}, \\ d_2(q,e_2) &= \max\{d(q,p)-r,0\}, \text{ and} \\ d_3(q,e_3) &= \max\{|d(q,p')-D|-r,0\} \end{aligned} \tag{8}$$

where p' is the parent object and D the corresponding distance for the node entry from which e_1 and e_3 were generated, and where p and r are the pivot and covering radius for the node corresponding to e_2 and e_3. Using the same definitions, the upper-bound distance functions for types 1 through 3 are

$$\begin{aligned} \hat{d}_1(q,e_1) &= d(q,p')+D, \\ \hat{d}_2(q,e_2) &= d(q,p)+r, \text{ and} \\ \hat{d}_3(q,e_3) &= d(q,p')+D+r. \end{aligned}$$

To support distance computations for descendants, we must associate certain information with each element. In particular, an element of type 1 must include the identity of the corresponding object, an element of type 2 must include a pointer to the corresponding node n and the distance $d(q,p')$, where p' is the parent object of n, and an element of type 3 must include p, r, and T, where (p,r,D,T) is the nonleaf node entry that gave rise to it. Note that a depth-first range search on this search hierarchy is equivalent to the range query algorithm described above.

The correctness of incremental nearest neighbor search with these definitions can be shown by applying the results in Section 4.1. In particular, for an element e_1 and the corresponding object o, $d_1(q,e_1) = |d(q,p')-D| = |d(q,p')-d(p',o)| \le d(q,o) \le d(q,p')+d(p',o) = |d(q,p')+D| = \hat{d}_1(q,e_1)$ follows from Lemma 3. For an element e_2 and an object o in its subtree, we have $d_2(q,e_2) = \max\{d(q,p)-r,0\} \le d(q,o) \le d(q,p)+r = \hat{d}_2(q,e_2)$ from Lemma 4. Finally, for an element e_3 and an object o in the subtree, $d_3(q,e_3) = \max\{|d(q,p')-D|-r,0\} = \max\{|d(q,p')-d(p',p)|-r,0\} \le d(q,o) \le d(q,p')+d(p',p)+r = |d(q,p')+D|+r = \hat{d}_3(q,e_3)$ follows from the above lemmas and Lemma 5.

Observe that when $d_1(q,e_1)$ and $d_3(q,e_3)$ are computed for elements e_1 and e_3, respectively, the distance information that they are based on is already available, so no additional computation of actual distances is necessary. In particular, D is a distance value computed during the construction of the M-tree, and $d(q,p')$ was computed earlier in the processing of the query and stored in e_2, the node element from which e_1 or e_3 is generated (i.e., p' is the parent object of the node corresponding to e_2). Thus, assuming that incremental nearest neighbor search is terminated after object o_k has been reported, any element of type 1 that still remains on the priority queue represents an object that we were able to prune without computing its actual distance from q. A similar statement applies to elements of type 3.

4.5 The SA-Tree

The Voronoi diagram is a widely used method for nearest neighbor search in point data. For each "site" p, its Voronoi cell identifies the area closer to p than to any other site. Thus, given a query point q, nearest neighbor search simply involves identifying the Voronoi cell that contains q. Another, somewhat indirect, way of constructing a search structure for nearest neighbor search based on the Voronoi diagram is to build a graph termed a *Delaunay graph*, defined by Navarro [50] to be a graph where each object is a node and two nodes have an edge between them if their Voronoi cells have a common boundary[18]. In other words, the Delaunay graph is simply an explicit representation of neighbor relations that are implicitly represented in the Voronoi diagram. Search for the nearest neighbor of a query point q then starts with an arbitrary object, and proceeds to a neighboring object closer to q as long as this is possible. Once we reach an object o where the objects in its neighbor set $N(o)$ (i.e., the objects connected to o by an edge) are all farther away from q, we know that o is the nearest neighbor of q. The reason this search process works on the Delaunay graph of a set of points is that the Delaunay graph has the property that if q is closer to a point p than to any of the neighbors of p in the Delaunay graph, then p is the object in S closest to q. The same search process can be used on any graph that satisfies this *Voronoi property*. In fact, for an arbitrary metric space $(\mathbb{U},d)$, a Delaunay graph for a set $S \subset \mathbb{U}$ is a minimal graph that satisfies the Voronoi property (i.e., removing any edge would cause violation of the property). Thus, any graph that satisfies the Voronoi property must include a Delaunay graph as a subgraph. Note that the Delaunay graph is not necessarily unique as there can be several such minimal graphs (possibly even with a different number of edges).

The Voronoi diagram serves as the inspiration for the sa-tree [50], a distance-based indexing method. In Section 4.3 we defined two other methods, the gh-tree and GNAT, that are also based on Voronoi cell-like partitioning. However, these structures are based on hierarchical partitioning, where at each level, the space is partitioned into two or more Voronoi cell-like regions. In contrast, the sa-tree attempts to approximate the structure of the Delaunay graph; hence its name, which is an abbreviation for *Spatial Approximation Tree*. As we saw in Section 4.3.2, Voronoi cells (or, perhaps more accurately, Dirichlet domains [9]) for objects cannot be constructed explicitly (i.e., their boundaries specified) if only inter-object distances are available. Moreover, it is possible to show [50] that without more information about the structure of the underlying space $\mathbb{U}$, the set of interobject distances for a finite metric space (S,d), $S \subset \mathbb{U}$, does not uniquely determine the Delaunay graph for S based on d. In other words, two sets $S \subset \mathbb{U}$ and $S' \subset \mathbb{U}'$ with identical interobject distances (i.e., (S,d) and (S',d') are isometric), possibly drawn from different underlying spaces $\mathbb{U}$ and $\mathbb{U}'$, may have different Delaunay graphs[19]. Hence, given only the interobject distances for a set S, the only way to ensure that the search structure includes all the edges in the Delaunay graph is to use the complete graph, i.e., the graph containing an edge between all pairs of nodes. However, such a graph is useless for searching with, as deciding on what edge to traverse from the initial object requires computing the distances from the query object to all the remaining objects in S (i.e., it is as expensive, $O(N)$, as brute-force search). The idea behind the sa-tree is to approximate the

[18] Navarro [50] actually uses the term "Voronoi graph", but "Delaunay graph" is a more appropriate term as the concept is closely related to the concept of Delaunay triangulations, except that the edges in the Delaunay graph merely indicate that the Voronoi regions have a common boundary and do not have an associated geometric shape.

[19] For example, suppose that $\mathbb{U} = \mathbb{U}' = \{a,b,c,x\}$, $d(a,b) = d(a,c) = d(b,c) = 2$ and $d'(a,b) = d'(a,c) = d'(b,c) = 2$. Furthermore, assume that $d(a,x) = 1$, $d(b,x) = 2$, and $d(c,x) = 3$ while $d'(a,x) = 3$, $d'(b,x) = 2$, and $d'(c,x) = 1$. If $S = S' = \{a,b,c\}$, the distance matrices for the two sets are the same. The graph with edges (a,b) and (a,c) (i.e., $N(a) = \{b,c\}$ and $N(b) = N(c) = \{a\}$) satisfies the Voronoi property for (S,d), since the nearest neighbor of any query object drawn from $\mathbb{U}$ can be arrived at starting at any object in S by only transitioning to neighbors that are closer to or at the same distance from the query object. However, this is not the case for (S',d'), since starting at b with $q = x$, b's only neighbor a is farther away from x than b is, so we cannot transition to the nearest neighbor c of x. Even though the graph with edges (a,b) and (b,c) (i.e., $N(b) = \{a,c\}$ and $N(a) = N(c) = \{b\}$) does satisfy the Voronoi property for both (S,d) and $(S'd')$, its existence does not invalidate the above observations.

proper Delaunay graph with a tree structure that retains enough edges to be useful for guiding search, but not so many that an excessive number of distance computations are required when deciding on what node to visit next.

The sa-tree is defined as follows for a finite metric space (S,d) (see the example below to clarify some of the questions that may arise). An arbitrary object a is chosen as the root node of the tree (since each object is associated with exactly one node, we use the terms object and node interchangeably in this discussion). Next, the smallest possible set of neighbors $N(a)$ is identified, such that x is in $N(a)$ iff for all $y \in N(a) - \{x\}$, $d(x,a) < d(x,y)$. Intuitively, for a legal neighbor set $N(a)$ (i.e., not necessarily the smallest such set), each object in $N(a)$ is closer to a than to the other objects in $N(a)$, and all the objects in $S \setminus N(a)$ are closer to one of the objects in $N(a)$ than to a. The objects in $N(a)$ then become children of a. The remaining objects in S are associated with the closest child of a (i.e., the closest object in $N(a)$), and the subtrees are defined recursively in the same way for each child of a. The distance to the farthest object in a subtree can also be stored in each node, i.e., for a this is $\max_{b \in S} d(a,b)$. Figure 17b shows a sample sa-tree for the two-dimensional points a–w given in Figure 17a, with a chosen as the root. In this example, $N(\mathsf{a}) = \{\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e}\}$. Note that h is not in $N(\mathsf{a})$ as h is closer to b than to a.

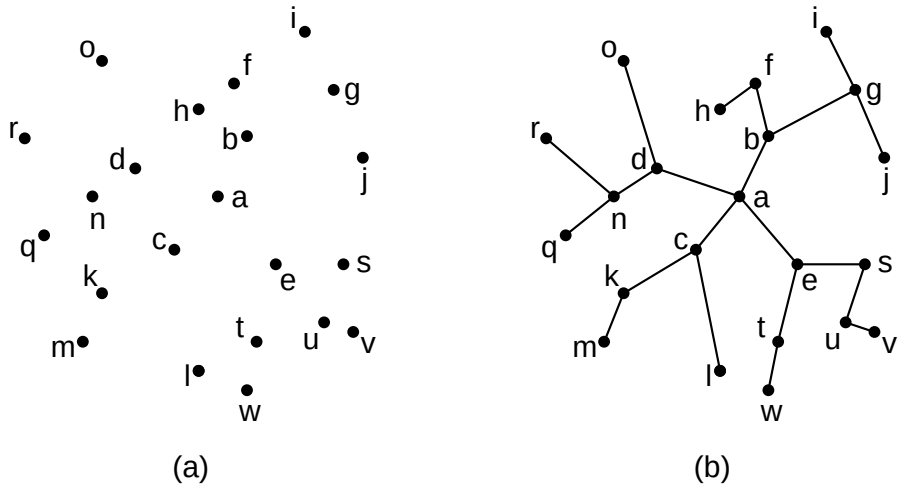

Figure 17: (a) A set of points in a two-dimensional Euclidean space, and (b) its corresponding sa-tree constructed using the algorithm of [50] when a is chosen as the root.

The fact that the neighbor set $N(a)$ is used in its definition (i.e., in a sense, the definition is circular) makes constructing a minimal set $N(a)$ expensive. In fact, Navarro [50] argues that its construction is an NP-complete problem. Thus, Navarro [50] resorts to a heuristic for identifying the set of neighbors. This heuristic considers the objects in $S \setminus \{a\}$ in the order of their distance from a, and adds an object o to $N(a)$ if o is closer to a than to the existing objects in $N(a)$. In fact, the sa-tree in Figure 17b has been constructed using his heuristic with a chosen as the root. An example of a situation where the heuristic would not find the minimal set of neighbors is shown in Figure 18, where approximate distances between four two-dimensional points a through d are labeled. The minimum neighbor set of a in this case is $N(\mathsf{a}) = \{\mathsf{d}\}$ (and $N(\mathsf{d}) = \{\mathsf{b},\mathsf{c}\}$) whereas use of the heuristic would lead to $N(\mathsf{a}) = \{\mathsf{b},\mathsf{c}\}$ (and $N(\mathsf{b}) = \{\mathsf{d}\}$). Although the heuristic does not necessarily find the minimal neighbor set, it is deterministic in the sense that for a given set of distance values, the same neighbor set is found (except for possible ties in distance values). Thus, using the heuristic, the structure of the sa-tree is uniquely determined once the root has been chosen. However, different choices of the root lead to different tree structures.

Using the sa-tree, it is easy to perform exact match queries (i.e., to search for an object in S) using the same procedure as in the Delaunay graph as described above. Of course, this is not very useful, as

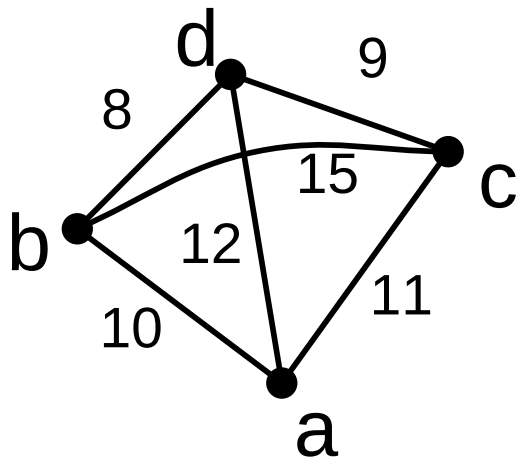

Figure 18: An example of four points a, b, c, d where the sa-tree construction algorithm does not find the minimal neighbor set $N(\mathrm{a})$.

the query object is typically not in S in most actual queries. Nearest neighbor and range search can be performed in the sa-tree for arbitrary query objects q by using the observation in Lemma 6. In particular, if a is the object corresponding to a root node, let c be some object in $\{a\} \cup N(a)$. Letting b be an arbitrary object in $N(a)$ and o be an object in the subtree associated with b (i.e., rooted at b), we know that o is closer to b than to c (or equidistant, e.g., if $c = b$). Thus, we can apply Lemma 6 to yield the lower bound $(d(q,b) - d(q,c))/2$ on $d(q,o)$ — that is, o is at a distance of at least $(d(q,b) - d(q,c))/2$ from q. Since o does not depend on c, we can select c in such a way that the lower bound on $d(q,o)$ is maximized, which occurs when $d(q,c)$ is as small as possible — that is, c is the object in $\{a\} \cup N(a)$ that is closest to q.

When performing range search with query radius ε, we can use the lower bound on the distances derived above to prune the search. In particular, when at node a, we first find the object $c \in \{a\} \cup N(a)$ such that $d(q,c)$ is minimized. Next, the search traversal visits each child $b \in N(a)$, except those for which $(d(q,b) - d(q,c))/2 > \varepsilon$ (or, equivalently, $d(q,b) > d(q,c) + 2\varepsilon$, as used in [50]), since in this case we know that $d(q,o) > \varepsilon$ for any object o in the subtree associated with b. Unlike exact match search, the above search process may require backtracking and the pursuit of several paths in the tree. Incidentally, we have identified a slight optimization over the search strategy presented in [50] (which is equivalent to what we described above). In particular, instead of basing the selection of c on the set $\{a\} \cup N(a)$ (i.e., on distances from q), we can use the larger set $\bigcup_{a' \in A(a)} (\{a'\} \cup N(a'))$, where $A(a)$ is the set of ancestors of a (with the understanding that a is an ancestor of itself). This strategy makes it more likely that c is close to q (since a larger set is used to select it), thus providing a larger value for the lower bound $(d(q,b) - d(q,c))/2$ on $d(q,o)$. Below, we prove the correctness of this approach.

The nearest neighbor search algorithm for the sa-tree proposed in [50] is a variant of the range search algorithm described above, and is based on a depth-first branch-and-bound strategy. Initially, ε is set to ∞, but is gradually reduced as objects are found at smaller distances to q than ε. The algorithm proceeds in a depth-first fashion and visits child nodes in order of increasing distance from q so as to increase the chance of quickly finding the nearest neighbor or at least reducing ε. Unfortunately, such a heuristic often chooses the wrong subtree to descend.

In order to alleviate these drawbacks, we propose to adapt our incremental nearest neighbor algorithm from Section 2 (Figure 3) to the sa-tree, thus replacing depth-first traversal of the tree with best-first traversal. For this, we must define the search hierarchy and the distance functions for each element type

in the search hierarchy. Naturally, the objects and the nodes of the sa-tree become elements of the search hierarchy, of types 0 and 1, respectively. Observe that each element of type 1 in the search hierarchy has one child element of type 0 (i.e., the object corresponding to the sa-tree node) and zero or more child elements of type 1. The distance function d_0 on the object elements is simply the distance function d on the objects. The distance function $d_1(q,e_1)$ is defined as

$$d_1(q,e_1) = \max\{\frac{d(q,b)-d(q,c)}{2}, d(q,b) - d_{\max}(b), 0\},$$

where $b \in N(a)$ is the object that corresponds to e_1, c is the object in $\bigcup_{a' \in A(a)}\{\{a'\} \cup N(a')\}$ that is closest to q, and $d_{\max}(b)$ is the greatest distance from b to an object in its subtree. Notice that here we use the optimization mentioned above, as well as the information about maximum distances in subtrees. For the purpose of defining $\hat{d}_1(q,e_1)$, only the maximum distances in subtrees are applicable, since the information about distances of neighbors does not provide an upper bound on distances. In other words, we obtain

$$\hat{d}_1(q,e_1) = d(q,b) + d_{\max}(b).$$

To permit computing the distances of descendants, an element e_1 of type 1 must not only carry a pointer to the corresponding node a, but also the distance value $d(q,c')$, where c' is the object in $A(a) \cup \bigcup_{a' \in A(a)\setminus\{a\}}\{N(a')\}$ that is closest to q. When processing an element e_1 corresponding to the node for object a, we evaluate $d(q,b)$ for each $b \in N(a)$, and then set D to the minimum of these distance values (i.e., $\min_{b\in N(a)}\{d(q,b)\}$) and of $d(q,c')$ (as stored in e_1). Clearly, D is now equal to $d(q,c)$, where c is the object in $\bigcup_{a' \in A(a)}\{\{a'\} \cup N(a')\}$ that is closest to q. Thus, by substituting D for $d(q,c)$ in the formula, we can evaluate $d_1(q,e_1')$ for each e_1' corresponding to $b \in N(a)$ with no additional distance computations. In other words, when processing e_1, only $|N(a)|$ distance computations are necessary.

To prove the correctness of our approach, we must show that $d_1(q,e_1) \leq d_0(q,e_0)$ for any ancestor e_1 of the element e_0 in the search hierarchy (see Section 2.5). Since $d_1(q,e_1)$ is the maximum of two quantities, we must show that neither is larger than $d_0(q,e_0)$. The first step is the following lemma:

Lemma 7 *If a is a node in an sa-tree and b' is an object in the subtree rooted at b, where $b \in N(a)$, then b' is closer to b than to any of its ancestors or their siblings — that is,*

$$d(b',c) \geq d(b',b), \forall c \in \bigcup_{a' \in A(a)}\{\{a'\} \cup N(a')\}.$$

Proof We first prove the lemma for $c \in \{a\} \cup N(a)$. Assume that $d(b',c) < d(b',b)$ for some $c \in \{a\} \cup N(a)$. There are two cases: 1) $c = a$, and 2) $c \in N(a)$. The first case, $d(b',a) < d(b',b)$, contradicts the assumption that b' is a descendant of b. In particular, since $d(b',b) \leq d(b',y)$ for all siblings y of b (or else b' would reside in the subtree of y), we have $d(b',a) < d(b',y)$ for all $y \in N(a)$. However, by the definition of a neighbor set, this implies that b' should be in $N(a)$. In the second case, $d(b',c) < d(b',b)$ for $c \in N(a)$, c cannot be equal to b, so it must be a sibling of b. However, this also contradicts the assumption that b' is a descendant of b, since b' should be in the subtree of the sibling that it is closest to (which it is not, since $d(b',c) < d(b',b)$). Thus in both cases we have a contradiction, and the statement holds for $c \in \{a\} \cup N(a)$.

The above reasoning can now be extended by induction to the general case where $c \in \bigcup_{a' \in A(a)}\{\{a'\} \cup N(a')\}$. In particular, we showed above that b' is closer to b than to a or to the siblings of b (or equidistant). Applying this principle to a and its parent a' shows that $d(b',c) \geq d(b',a)$, where $c = a'$ or $c \in N(a')$. Thus, since $d(b',a) \geq d(b',b)$, we also have $d(b',c) \geq d(b',b)$ for this case, or more generally, for $c \in \{\{a'\} \cup N(a')\}$ where a' is a or its parent. It should be clear that this loose induction argument applies all the way up to the root of the tree, thereby proving the lemma. ■

Based on the above lemma and on Lemma 6, we conclude that $(d(q,b) - d(q,c))/2 \leq d(q,b')$ for any object b' in the subtree rooted at b. The other lower bound, $d(q,b) - d_{\max}(b) \leq d(q,b')$, follows from Lemma 4 (as does the upper bound $d(q,b') \leq d(q,b) + d_{\max}(b)$). Thus, we have shown that $d_1(q,e_1) \leq d_0(q,e_0)$ always holds and, therefore, that the above hierarchy guarantees correctness.

Recall from Section 2.5 that the correctness of the incremental nearest neighbor algorithm implies that it is optimal with respect to the search hierarchy. In other words, it visits the same search hierarchy elements when computing k neighbors as a top-down range query using the distance between q and the k^{th} neighbor. Since the distance functions we defined on the search hierarchy are equivalent to the conditions used in the range query algorithm presented in [50], this shows that the incremental nearest neighbor algorithm achieves the same performance in terms of distance computations (on the data objects themselves). On the other hand, a depth-first nearest neighbor algorithm such as that proposed by Navarro [50] depends on a heuristic to bound the search, which cannot guarantee optimality.

4.6 Distance Matrix Methods

4.6.1 AESA

The distance-based index methods that we have considered so far impose a hierarchy on the set of objects that guides the order of distance computations during query evaluation. AESA (Approximating and Eliminating Search Algorithm) [67, 68][20] takes another approach. During preprocessing, all $O(N^2)$ inter-object distances are computed for the N objects in S and stored in a matrix. At query time, the distance matrix is used to provide lower bounds on distances to objects whose distances have not yet been computed, based on object distances already computed. The process is initiated by computing the distance from the query object to an arbitrary data object, allowing establishing the initial lower-bound distances of the remaining data objects. The algorithm uses these lower bounds to guide the order in which objects are chosen to have their distances from the query object q computed and to eliminate objects from consideration (hopefully without computing their actual distances from q). In other words, AESA treats all N data objects as pivot objects when performing search. Although designed for finding nearest neighbors, AESA can also be used with almost no modification to perform range searching.

According to experiments presented in [67], nearest neighbors can be obtained with AESA using remarkably few distance computations. In particular, AESA was observed to require at least an order of magnitude fewer distance computations than competing methods and was argued to have constant-time behavior with respect to the size of the data set [67]. These benefits are obtained at the expense of quadratic space complexity, quadratic time preprocessing cost, and linear time and storage overhead during search. Thus, although promising, the method is practical only for relatively small data sets, of at most a few thousand objects. For example, for 10,000 data objects, the distance matrix occupies about 400 MB, assuming 4 bytes per distance value. Nevertheless, if distances are expensive to evaluate and we can afford the large preprocessing cost, the search performance is hard to beat with other methods.

Of course, one could ask if it is really worthwhile to perform $N \cdot (N-1)/2$ distance computations between the objects, when by using brute force we can always find the nearest object to q using N distance computations. The payoff occurs when we can be sure that the set of objects is static and that there will be many queries (more than N, assuming that preprocessing time and query time are of equal importance), and that most of these queries will be nearest neighbor queries for low numbers of neighbors or range queries with small query radii (otherwise, AESA will tend to require $O(N)$ distance computations, like the brute-force approach). The complexity arguments made in favor of AESA must also bear in mind that the constant-time claim refers to the number of distance computations, while the distance matrix has to

[20] The difference between [67] and [68] lies in the presentation of the algorithm and in the order in which the objects are chosen whose distance from the query object is computed — that is, in the "approximating" step (see footnote 23 below).

be accessed many times for each query ($\Omega(N)$ for each nearest neighbor query[21]), although the distance computations are usually many orders of magnitude more complex than the operation of accessing the distance matrix.

The key to AESA [67] is the property described in Lemma 3: for any objects o and p in the data set S and any query object $q \in \mathbb{U}$, the following inequality holds:

$$|d(q,p) - d(p,o)| \leq d(q,o).$$

Thus, if $S_c \subset S$ is the set of objects whose distances from q have been computed, the greatest known lower bound $d_{\text{lo}}(q,o)$ on $d(q,o)$ for any object $o \in S \setminus S_c$ is

$$d_{\text{lo}}(q,o) = \max_{p \in S_c} \{|d(q,p) - d(p,o)|\} \tag{9}$$

The algorithm uses this lower bound to eliminate objects o in $S \setminus S_c$ whose lower-bound distances are greater than the distance of the nearest neighbor candidate o_n, i.e., $d_{\text{lo}}(q,o) > d(q,o_n)$ (for range search with query radius ε, the elimination criterion is $d_{\text{lo}}(q,o) > \varepsilon$)[22]. Hence, it maintains the set $S_u \subset S$ of objects whose distances have not been computed and that have not been eliminated based on their lower-bound distances. At each step of the algorithm, the next object $p \in S_u$ whose distance is to be computed is chosen as the one whose lower-bound distance $d_{\text{lo}}(q,p)$ is smallest (initially, an arbitrary object is chosen). Next, the algorithm computes $d(q,p)$ (which, we point out, may be very expensive), updates the nearest neighbor candidate o_n if necessary (in the case of range searching, we instead add p to the result set if $d(q,p) \leq \varepsilon$), and then eliminates objects from S_u that cannot be the nearest neighbor (or within the range for range searching) as described above. The algorithm is terminated once S_u becomes empty — that is, once the greatest known lower-bound distance $d_{\text{lo}}(q,o)$ for each object $o \in S \setminus S_c$ is greater than $d(q,o_n)$ (or ε in the case of range search). Observe that the lower-bound distance, $d_{\text{lo}}(q,o)$, for an object $o \in S_u$ need not be computed from scratch (i.e., for all $p \in S_c$) each time the algorithm uses it. Rather, the algorithm stores the current lower-bound distance for each object $o \in S_u$, and incrementally updates $d_{\text{lo}}(q,o)$ in each iteration as a new distance value is computed. Storing and maintaining this information accounts for the linear space and time overhead of the algorithm (besides the quadratic space and time for constructing and storing the distance matrix).

The rationale for picking the object p to process next based on the smallest lower bound d_{lo} is that, hopefully, such a choice ensures that p is relatively close to q. As pointed out by Vidal [68], the closer p is to q, the greater is the tendency for $|d(p,o) - d(q,p)|$ to be large, which means that the lower bound $d_{\text{lo}}(q,o)$ is larger and hence the potential for pruning increases. Of course, other strategies for picking the next object are also possible[23]. Some possible strategies include picking the object at random, choosing the object with the greatest value of $d_{\text{lo}}(q,p)$, or even basing the choice on the upper bound $d_{\text{hi}}(q,p)$, described below. Wang and Shasha [70] explored several different choices which are described briefly in Section 4.6.3.

AESA is easily extended to a k-nearest neighbor algorithm by maintaining a list of the k candidate nearest neighbors seen so far, and by using the largest distance among the k candidates for the elimination step. There are a number of ways to implement incremental nearest neighbor search within the

[21] To see why the number of accesses is at least proportional to N (i.e., $\Omega(N)$), observe that even if the first object picked as the candidate nearest neighbor turns out to be the actual nearest neighbor, the distances between that object and all the other objects must be accessed to establish that this is indeed the case.

[22] If $d_{\text{lo}}(q,o) > d(q,o_n)$ is satisfied, we know that $d(q,o) > d(q,o_n)$ since $d_{\text{lo}}(q,o) \leq d(q,o)$; similarly, $d_{\text{lo}}(q,o) > \varepsilon$ means that $d(q,o) > \varepsilon$.

[23] In the original formulation of AESA [67], the selection criterion was actually based on picking the object $p \in S_u$ that minimizes the value of $\sum_{s \in S_c} \{|d(q,s) - d(s,p)|\}$ rather than that of $d_{\text{lo}}(q,p) = \max_{s \in S_c} \{|d(q,s) - d(s,p)|\}$, which Vidal [68] later claimed was a better "approximation". One possible rationale for the claimed improvement is that the former minimizes the average lower bound while the latter minimizes the maximum of the lower bounds, which yields a tighter lower bound.

AESA framework. Below, we identify three approaches; most other approaches are variants of these three. Conceptually, all three algorithms maintain two sets, S_c and S_u, of data objects, where $S = S_c \cup S_u$, whose distances from the query object have been computed and have not been computed, respectively. At a given stage in an algorithm, k of the objects in S_c have already been reported as neighbors, while the unreported object with the smallest distance in S_c is a candidate for being the $k+1^{\text{st}}$ neighbor. Each object o in S_u has an associated lower-bound distance, $d_{\text{lo}}(q,o)$. If the distance of the $k+1^{\text{st}}$ candidate neighbor in S_c is smaller than the minimum lower-bound distance in S_u, the $k+1^{\text{st}}$ neighbor has been identified and can be reported. Otherwise, the algorithm takes some action on some or all of the objects in S_u. The algorithms differ in the way they represent S_u, which has implications for the accuracy of the lower-bound distance $d_{\text{lo}}(q,o)$, i.e., the number of objects in S_c that have been used to update $d_{\text{lo}}(q,o)$ using Equation 9. The effect is that the cost of updating lower-bound distances for a given number of neighbors varies among the algorithms. Note that the algorithms cannot eliminate any of the objects in S_u as is done in AESA, since we do not know how many neighbors will eventually be requested. However, by eliminating objects from S_u in AESA, the cost of future updating of lower-bound distances is reduced. A similar effect can be achieved for incremental nearest neighbor search by using suitable representations of S_u.

The simplest incremental nearest neighbor algorithm based on AESA is one that extends the k-nearest neighbor algorithm in a straightforward manner. The set S_u is maintained as a whole, with the lower-bound distances of the objects in S_u based on all the objects in S_c. Following the framework for INN search laid down in Section 2.2, the search hierarchy consists of objects (type 0, representing unreported objects in S_c) and sets of objects (type 1, representing S_u). The distance function d_0 is based on d, as before, but the distance function $d_1(q,e_1)$, where e_1 represents $S_u \subset S$, is defined as the smallest lower-bound distance among the objects in S_u, i.e.,

$$d_1(q,e_1) = \min_{o \in S_u} \{d_{\text{lo}}(q,o)\}.$$

The correctness criterion (see Section 2.5), $d_1(q,e_1) \leq d_0(q,e_0)$, is clearly satisfied for this scheme, where e_0 represents object o, e_1 represents subset S_u, and $o' \in S_u$, since

$$d_1(q,e_1) = \min_{o \in S_u} \{d_{\text{lo}}(q,o)\} \leq d_{\text{lo}}(q,o') \leq d(q,o') = d_0(q,o).$$

An upper-bound distance function for elements of type 1 can be derived in the same way:

$$\hat{d}_1(q,e_1) = \max_{o \in S_u} \{d_{\text{hi}}(q,o)\},$$

where $d_{\text{hi}}(q,o) = \min_{p \in S_c} \{d(q,p) + d(p,o)\}$.

The incremental nearest neighbor algorithm (Figure 3) using this search hierarchy works as follows. Initially, we insert S on the priority queue, with the lower-bound distances of all $o \in S$ set to 0. If the element on the priority queue with the smallest distance value is an object, we report it as the next nearest neighbor. Otherwise, if it is an element e_1 representing the set S_u, we choose the object $o \in S_u$ with the smallest lower-bound distance, $d_{\text{lo}}(q,o)$ (ties are broken arbitrarily). Next, we compute $d(q,o)$ and insert o on the priority queue. Finally, the lower-bound distances of the remaining objects in S_u are updated, and we insert an element e_1' representing $S_u \setminus \{o\}$ on the priority queue with a key of $d_1(q,e_1')$ (i.e., the minimum among the updated lower-bound distances). Thus, the search hierarchy forms a lopsided tree where each nonleaf node (representing a set of objects) has two children, one of which is a leaf node (representing an object).

The search hierarchy for incremental nearest neighbor search that we have described leads to the same number of distance computations for finding the k nearest neighbors as would the original AESA

algorithm. The problem is that no "elimination" is performed by the incremental algorithm. In other words, each time a new distance computation is performed (i.e., when processing the set of objects $S \setminus S_c$, where S_c is the set of objects whose distances have been computed) we must update the lower bound distance $d_{\text{lo}}(q,o)$ for $|S \setminus S'_c|$ objects, where $S'_c = S_c \cup \{o'\}$ and o' was the object whose distance was just computed. In contrast, AESA needs only to update the lower-bound distance for $|S \setminus S'_c \setminus S_e|$, where S_e is the set of objects that have been eliminated in previous steps. Hence, the incremental algorithm may have a much higher overhead.

To achieve more elimination, we can put the burden on the priority queue of the incremental nearest neighbor algorithm. In particular, instead of letting elements of type 1 represent sets of objects, we use the objects themselves, using the lower-bound distance as the distance function d_1. The algorithm initially picks an arbitrary object o, computes the distance $d(q,o)$ and uses it to establish a lower-bound distance for the remaining objects. The object o is inserted on the priority queue with its actual distance, while the remaining objects are inserted on the priority queue with their lower-bound distances. Each time a lower-bound object o is processed by the algorithm, we refine its lower-bound distance $d_{\text{lo}}(q,o)$ based on already computed distances (for objects in S_c) until $d_{\text{lo}}(q,o) > d_t(q,e_t)$ for the element e_t on the priority queue with the smallest distance (in which case we insert o back on the priority queue with its updated lower-bound distance), or compute $d(q,o)$ if the distances of all the objects in S_c have been used. The result is that the search hierarchy has a fan-out of $|S|$ at the root, while the remaining elements of the search hierarchy have only one child or are leaves.

Since each element of type 1 in the search hierarchy represents a single object, rather than the set $S_u = S \setminus S_c$, this approach minimizes the overhead in terms of the number of times $d_{\text{lo}}(q,o)$ is updated for each object. In fact, the updating overhead is lower with this approach than with AESA, but the price that must be paid is that more priority queue operations are performed. A reasonable compromise results from a blend of the two extreme approaches, where S_u is broken up into subsets of different sizes, and the lower-bound distances of the objects in each subset have been updated different numbers of times (based on different numbers of elements of S_c). As in the first approach, search hierarchy elements of type 1 represent sets of objects. However, we allow such elements to have several children; in other words, we partition the set of objects into several subsets, based on the lower-bound distances of the objects. As in the second approach, when processing an element representing a set S'_u of objects, a new distance computation is performed only if the distances of all the objects in S_c have already been applied to update the lower-bound distances of the objects in S'_u. If the objects in S_c have not all been applied, we update the lower-bound distances using all the objects in S_c, or until the minimum lower-bound distance for the objects in S'_u exceeds the distance of the element at the front of the queue (i.e., S'_u is no longer the smallest element on the priority queue). In our implementation, we found that partitioning into two subsets each time led to the best performance, i.e., the best balance between the cost of updating lower-bound distances, the cost of priority queue operations, and the cost of managing the subsets. Furthermore, it turned out to be better to quickly partition into two subsets of random size (as is done in the Quicksort algorithm) rather than to spend time partitioning the sets into two equal parts.

It is interesting to make an analogy between the search hierarchy resulting from our last approach and the hierarchical indexing methods described in Section 4.2. The search hierarchy superficially resembles a variant of the vp-tree structure, where the same pivot is used for all partitions at a given level (as in the fixed-queries tree that we briefly discussed in Section 4.2.4). The crucial difference between the search hierarchy and such fixed-pivot metric tree structures is that the choice of pivots (i.e., the order in which the objects are chosen for S_c) and the partition keys in the search hierarchy (i.e., the lower-bound distances) are highly dependent on the query object q.

Based on our experience, each of the three approaches to INN search outlined above performs well for the sizes of data sets for which AESA is practical. For a small number of neighbors, the last approach tends to be superior, with the second being next best and the first being worst, whereas for a large number

of neighbors (when many distance computations must be performed in any case), the first approach eventually becomes best. It must be admitted that the overhead of the algorithms is higher than that of AESA for computing a given number of neighbors (but the number of distance computations is the same). However, in situations where the number of neighbors is unknown in advance and distance computations are expensive, using the INN algorithms is superior to using k-NN AESA, since in the latter case, the number of neighbors must be guessed at. In particular, the INN algorithms perform the minimum numbers of distance computations for the number of neighbors that are eventually requested, whereas using a k-NN algorithm usually results in computing too many neighbors and thus in too many distance computations (or worse, too few neighbors are computed and the algorithm must be re-applied).

4.6.2 LAESA

Recall that AESA is impractical for all but the smallest data sets due to the large preprocessing and storage costs. LAESA (Linear AESA) [48, 49] alleviates this drawback by choosing a fixed number M of pivots (termed *base prototypes* by Micó et al. [48, 49]), whose distances from all other objects are computed. Thus, for N data objects, the distance matrix contains $N \cdot M$ entries rather than $O(N^2)$ for AESA (or more precisely $N(N-1)/2$ entries assuming that only the lower triangular portion of the matrix is stored). An algorithm for choosing the M pivots is presented by Micó et al. [49]. Essentially, this algorithm attempts to choose the pivots such that they are *maximally separated*, i.e., as far away from each other as possible (a similar procedure was suggested by Brin [9] for GNAT; see Section 4.3.2).

The LAESA search strategy is very similar to that of AESA, except that some complications arise from the fact that not all objects in S serve as pivot objects in LAESA (and the distance matrix does not contain the distances between non-pivot objects). In particular, as before, let $S_c \subset S$ be the set of objects whose distances from q have been computed and let $S_u \subset S \setminus S_c$ be the set of objects whose distances from q have yet to be computed and that have not been eliminated. The distances between the query object q and the pivot objects in S_c are used to compute a lower bound on the distances of objects in S_u from q, and these lower bounds allow eliminating objects from S_u based on the distance from q of the current candidate nearest neighbor o_n (or ε in the case of range search). The difference here is that non-pivot objects in S_c do not help in tightening the lower-bound distances of the objects in S_u, as the distance matrix stores only the distances from the non-pivot objects to the pivot objects and not to the remaining objects. Thus, Micó et al. [49] suggest treating the pivot objects in S_u differently than non-pivot objects when

1. selecting the next object in S_u to have its distance from q computed (since computing the distances of pivot objects early will help in tightening distance bounds), and

2. eliminating objects from S_u (since eliminating pivot objects that may later help in tightening the distance bounds is undesirable).

A number of possible policies can be established for this purpose. The policies explored by Micó et al. [49] are simple, and call for

1. selecting a pivot object in S_u over any non-pivot object, and

2. eliminating pivot objects from S_u only after a certain fraction f of the pivot objects have been selected into S_c (f can range from 0 to 100%; note that if $f = 100\%$, pivots are never eliminated from S_u).

As with AESA, several possible strategies can be pursued for INN search within the framework of LAESA. Since LAESA is practical for much larger data sets than AESA, the first two approaches that

we presented for AESA are too inefficient. In particular, too much overhead in terms of updating lower-bound distances results from the approach that maintains $S_u = S \setminus S_c$ as a whole and updates the lower-bound distances for all elements in S_u each time the distance of a pivot object is computed. Similarly, entering each element in S_u separately into the priority queue results in excessive priority queue cost (as the priority queue will initially hold N elements). Thus, we are left with the approach that partitions S_u into subsets of varying sizes. Unfortunately, using this strategy makes it difficult to apply arbitrary selection policies, since the resulting algorithm processes only one of the existing subsets of S_u at a time (the one containing the element having the minimum lower-bound distance $d_{\mathrm{lo}}(q,o)$). In particular, if the subset being processed does not contain the object that the selection policy would call for (e.g., the pivot with the smallest lower bound distance in the policy mentioned above), we must either settle for an inferior choice of an object whose distance is to be computed next, or we must inspect more than one subset on the priority queue to find a better choice, potentially at a high cost. To overcome this problem, we partition S_u into the sets S_u^p of pivot objects and S_u^n of non-pivot objects, and represent subsets of each of these sets on two separate priority queues, Q_a and Q_b, respectively. Thus, any selection policy that is devised for LAESA nearest neighbor search can be applied to decide from which priority queue, Q_a or Q_b, to take the next element to process. Priority queue Q_b is also used for unreported objects in S_c.

Fitting the above strategy into the INN search framework of Section 2.2, the search hierarchy elements are objects in S_c (type 0), subsets of S_u^n (type 1), and subsets of S_u^p (type 2). As in Section 4.6.1, the distance d_0 is the same as the distance function d for the corresponding objects. The distance functions $d_1(q,e_1)$ and $d_2(q,e_2)$ are defined in terms of the lower-bound distances of the objects in T, the set of objects corresponding to e_1 or e_2, i.e., $\min_{o \in T} d_{\mathrm{lo}}(q,o)$. We can imagine that the search hierarchy has a root, representing the data set S itself, having two child elements of types 1 and 2, respectively. The two subtrees of the root represent the non-pivot objects and pivots, respectively, and each has a structure similar to that of the last search hierarchy described in Section 4.6.1. Of course, the INN search strategy described above departs from the framework in that it uses two priority queues, Q_a containing only elements of type 2 and Q_b containing elements of types 0 and 1. Usually, the algorithm processes the element in either priority queue having the smallest distance, but the selection policy may interfere with this, as described below. As always, processing an element of type 0 involves outputting the corresponding object as the next nearest neighbor. For an element of type 1 or 2, representing a set of objects T, the action depends on whether all pivots in S_c have already been used to update the lower-bound distances of the objects in T. If not, we update the lower-bound distances based on all the unused pivots in S_c, or until the minimum lower-bound distance among the objects in T exceeds the smallest distance of the elements in either priority queue[24]. Finally, T is partitioned into several subsets based on lower-bound distance (e.g., by random bipartitioning, as suggested in Section 4.6.1), and the resulting subsets are inserted into the proper priority queue. Otherwise, if all the pivots in S_c have already been applied on the objects in T, the action is somewhat different for elements of type 1 and 2 (i.e., according to whether T is a set of non-pivot objects or of pivots). For elements of type 1, we compute the distances of the objects in T by updating lower-bound distances, until the lower-bound distance of the next object is larger than or equal to the smallest distance of an element in either Q_a or Q_b. For elements of type 2, we compute the distance of the pivot in T having the minimum lower-bound distance, update the lower-bound distances of the remaining objects in T, and partition T as described above.

Let e_a and e_b be the elements in Q_a and Q_b, respectively, having the smallest distances. As mentioned above, we usually process the element having the smaller distance of the two. However, if element e_b is of type 1 (as opposed to type 0) $d_1(q,e_a) \leq d_2(q,e_b)$, and all pivots in S_c have been used to update the lower-bound distances in the corresponding set, then the choice between processing e_a or e_b is left to the

[24] If T is large, we may want to update the lower-bound distances only once, even if the minimum lower-bound distance is still too low.

selection policy. For example, using the policy presented in [49], we would always process e_b in such a case.

The incremental nearest neighbor search strategy described above performs the same number of distance computations for a given number of neighbors as the LAESA nearest neighbor search strategy, when the same selection policy is used (for deciding between computing the distance of a pivot or a non-pivot object). In particular, if e_t is an element of type 1 or 2 that corresponds to $T \subset S$, and e_0 corresponds to object $o \in T$, then

$$d_t(q,e_t) \leq d_{\text{lo}}(q,o) \leq d(q,o) = d_0(q,e_0),$$

where $d_{\text{lo}}(q,o)$ is any lower-bound distance for o. Thus, the resulting algorithm is correct with respect to the search hierarchy, as established in Section 2.5, if the selection policy is taken into account. Moreover, besides the exceptions due to the selection policy, we only compute distances of objects whose lower-bound distances are lower than the distances of all elements remaining on the two priority queues.

A somewhat simpler INN search strategy is obtained by computing the distances of all pivots in the initialization phase of the search. In other words, S_c initially contains all the pivots in S, and $S_u = S \setminus S_c$ contains only non-pivot objects. Thus, in this case, only one priority queue is needed, and the search hierarchy contains objects in S_c (type 0) and subsets of S_u (type 1). The actions performed when processing elements of type 1 are the same as described above. The disadvantage of such a strategy is that the distances of all the pivots must be computed, whereas this is not always the case in LAESA, depending on the selection and pivot elimination policy. However, for some policies [49], the distances of all the pivots must usually be computed. Furthermore, precomputing the distances of all the pivots allows ordering them by distance. In [67], the author argued that basing lower-bound distances on objects close to the query object was more effective. Thus, using the pivots in order of distance when updating lower-bound distances should lead to more effective search.

4.6.3 Other Distance Matrix Methods

Shapiro [62] described a nearest neighbor algorithm (which is also applicable to range searching) that is closely related to LAESA, which also uses an $N \cdot M$ distance matrix based on M pivot objects. The order in which the data objects are processed in the search is based on their positions in a list $(o_1, o_2, \ldots)$ sorted by distance from the first pivot object p_1. Thus, the search is initiated at the object whose distance from p_1 is most similar to $d(q,p_1)$, where q is the query object — that is, the element at the position j for which $|d(q,p_1) - d(p_1,o_j)|$ is minimized (this value is a lower bound on $d(q,o_j)$, as shown in Lemma 3). The goal is to eliminate object o_i from consideration as soon as possible, thereby hopefully avoiding the need to compute its distance from q. Therefore, when object o_i is processed during the search, we check whether the pruning condition $|d(q,p_k) - d(p_k,o_i)| > d(q,o_n)$ is satisfied for each pivot object $p_1, p_2, \ldots$ in turn until o_i can be eliminated; otherwise, we compute $d(q,o_i)$ (and possibly update o_n). The search continues alternating in the two directions — that is, for $i = j+1, j-1, j+2, j-2, \ldots$, stopping in either direction when the pruning condition $|d(q,p_1) - d(p_1,o_i)| > d(q,o_n)$ is satisfied, where o_n is the current candidate nearest neighbor[25].

Observe that Shapiro's algorithm is less sophisticated than LAESA in two ways:

1. the order used in the search is based on position in the sorted list ordered by distance from p_1, and
2. only the first pivot p_1 affects the order in which the data objects are processed.

In contrast, LAESA uses the lower-bound distances as determined by *all* pivot objects that have been applied so far to guide the search (i.e., to choose the pivot to use next and to decide when to compute the

[25] Recall that range search can be performed by basing the pruning condition on ε instead of $d(q,o_n)$.

actual distances of data objects). In other words, rather than applying all pivots for each object in turn as done by Shapiro, LAESA applies each pivot in turn for all objects (the difference can be characterized roughly in terms of processing the pivot-object distance matrix in row-major or column-major order).

Wang and Shasha [70] described a search method based on distance matrices that is similar to AESA. However, they allow for the case where only some of the distances have been precomputed, as in LAESA. In contrast to LAESA, no assumptions are made about the pairs of objects for which the distance is precomputed (so that no distinction is made between pivot and non-pivot objects). In other words, we are given a set of inter-object distances for arbitrary pairs of objects in S. Search is facilitated by the use of two matrices D_{lo} and D_{hi} (called *ADM* and *MIN* in [70]), constructed on the basis of the precomputed distances, where $D_{\text{lo}}[i,j] \leq d(o_i,o_j) \leq D_{\text{hi}}[i,j]$, given some enumeration $o_1, o_2, \ldots, o_N$ of the objects in S[26]. In other words, all entries in D_{lo} and D_{hi} are initialized to zero and ∞, respectively, except that the entries on their diagonals are set to zero, and if $d(o_i,o_j)$ has been precomputed, then $D_{\text{lo}}[i,j]$ and $D_{\text{hi}}[i,j]$ are both set to $d(o_i,o_j)$.

A dynamic programming algorithm is described by Wang and Shasha [70] that utilizes a generalized version of the triangle inequality[27] to derive values for the entries of D_{lo} and D_{hi} whose distance values are missing, in such a way that they provide as tight a bound as possible, based on the precomputed distances that are available. In particular, the generalized triangle inequality property was used by Wang and Shasha to derive rules for updating $D_{\text{lo}}[i,j]$ and $D_{\text{hi}}[i,j]$ based on the values of other entries in D_{lo} and D_{hi} (some of these rules use entries in D_{lo} to update entries in D_{hi}, and others do the opposite). At search time, the matrices D_{lo} and D_{hi} are augmented so that the query object q is treated as if it were object o_{N+1}. In particular, $D_{\text{lo}}[i,N+1]$ and $D_{\text{hi}}[i,N+1]$ are initialized to 0 and ∞, respectively. Observe that the values of $D_{\text{lo}}[i,N+1]$ and $D_{\text{hi}}[i,N+1]$ correspond to our definitions of $d_{\text{lo}}(q,o_i)$ and $d_{\text{hi}}(q,o_i)$, respectively, in Section 4.6.1.

The nearest neighbor algorithm presented by Wang and Shasha [70] follows the same general outline as AESA. Thus any object o_i satisfying $D_{\text{lo}}[i,N+1] > d(q,o_n)$ can be pruned from the search, where o_n is the current candidate nearest neighbor. The difference here is that when $d(q,o_k)$ is computed for some candidate object o_k, their method attempts to update $D_{\text{lo}}[i,j]$ and $D_{\text{hi}}[i,j]$ (by applying their generalized triangle inequality property) for all pairs of objects $o_i, o_j \in S$ whose actual distances are not available (i.e., either precomputed or computed during the search), thereby possibly yielding a tighter bound on $d(o_i,o_j)$. In contrast, in AESA, only the values of $d_{\text{lo}}(q,o_i)$ and $d_{\text{hi}}(q,o_i)$ are updated for all objects $o_i \in S$, corresponding to $D_{\text{lo}}[i,N+1]$ and $D_{\text{hi}}[i,N+1]$, respectively.

Since updating the entire matrices D_{lo} and D_{hi} can be expensive if done for all pairs at each stage of the algorithm, Wang and Shasha [70] describe two alternatives, one of which is almost equivalent to the updating policy used in AESA (the difference is that in AESA, upper-bound distances are not maintained, whereas such upper bounds can be used to update the values of $d_{\text{lo}}(q,o)$ in the same way as is done for $D_{\text{lo}}[N+1,i]$ in the method of Wang and Shasha [70]). Wang and Shasha [70] identify four heuristics for picking the next candidate object during search. The next object o_i for which to compute $d(q,o_i)$ is chosen as the object in S_u (as defined in Section 4.6.1) having

1. the least lower bound $D_{\text{lo}}[i,N+1]$,
2. the greatest lower bound $D_{\text{lo}}[i,N+1]$,
3. the least upper bound $D_{\text{hi}}[i,N+1]$, or
4. the greatest upper bound $D_{\text{hi}}[i,N+1]$.

[26] Note that the matrices are symmetric and that their diagonals are zero. Thus, only the lower triangular part of each matrix is actually maintained.

[27] For example, based on $d(o_1,o_4) \geq d(o_1,o_3) - d(o_3,o_4)$ and $d(o_1,o_3) \geq d(o_1,o_2) - d(o_2,o_3)$ we can conclude that $d(o_1,o_4) \geq d(o_1,o_2) - d(o_2,o_3) - d(o_3,o_4)$.

According to their experiments, the best choice is the object with the least lower-bound distance estimate (i.e., item 1), which is the same as used in AESA. Thus, essentially the same search hierarchies can be used to implement incremental nearest neighbor search in this setting as those we outlined in Section 4.6.1.

Micó et al. [47] proposed a hybrid distance-based indexing method termed TLAESA that makes use of both a distance matrix and hierarchical clustering, thereby combining aspects of LAESA [49] (see Section 4.6.2) and the mb-tree [52] (see Section 4.3.3). The hierarchical search structure used by TLAESA applies the same variation on the gh-tree as is used in the mb-tree: two pivots are used in each node for splitting the subset associated with the node (based on which pivot is closer), where one of the pivots in each nonroot node is inherited from its parent. The search algorithm proposed by Micó et al. uses a partial distance matrix as in LAESA, thus introducing a second set of pivots (termed 'base prototypes' by Micó et al. [47]). Initially, the algorithm computes the distances between q and all distance matrix pivots. Next, when traversing the tree structure, TLAESA uses the distance matrix pivots to compute lower bounds on the distances of the tree pivots from q, rather than computing their actual distances from q. In other words, if $p_1, p_2, \ldots, p_M$ are the distance matrix pivots and p is a tree pivot, a lower bound $d_{\text{lo}}(q,p)$ on $d(q,p)$ is obtained by applying Lemma 4 to all the distance matrix pivots. Therefore, $d_{\text{lo}}(q,p) \leq d(q,p)$ where

$$d_{\text{lo}}(q,p) = \max_i \{|d(q,p_i) - d(p_i,p)|\}.$$

Now, if r is the ball radius corresponding to the tree pivot p, $d_{\text{lo}}(q,p) - r$ is the lower bound on the distances between q and all the objects in the subtree rooted at the child node corresponding to p (via Lemma 5, setting $r_{\text{lo}} = 0$, $r_{\text{hi}} = r$, $s_{\text{lo}} = d\text{lo}(q,p)$, and $s_{\text{hi}} = \infty$). The actual distances of data objects (other than distance matrix pivots) are then computed only when reaching leaf nodes of the tree.

Several other variants of AESA and LAESA have been developed (e.g., [57, 69]). For example, Ramasubramanian and Paliwal [57] presented a variant of AESA that is tailored to vector spaces, allowing them to reduce the preprocessing cost and space complexity to $O(nN)$, where n is the dimensionality of the vector space (thus, there are significant savings compared to $O(N^2)$ since $n \ll N$). This algorithm appears to be quite related to LAESA.

Although both AESA and LAESA usually lead to a low number of distance computations when searching, they do have an overhead of $O(N)$ in terms of computations other than distance. Vilar [69] presents a technique (termed *Reduced Overhead AESA*, or *ROAESA* for short), applicable to both AESA and LAESA, that reduces this overhead cost by using a heuristic to limit the set of objects whose lower-bound distances d_{lo} are updated at each step of the algorithm. In particular, rather than updating d_{lo} for all objects in S_u (to use the notation in Section 4.6.1), ROAESA partitions S_u into two subsets which are termed *alive* (S_a) and *not alive* (S_d), and only updates the d_{lo} values of the objects in S_a. ROAESA starts by picking an object o_1 whose distance from q is computed, and o_1 is entered into S_c. Next, it computes d_{lo} for all objects in $S_u = S \setminus S_c$ on the basis of o_1, and makes the object o_a in S_u with the lowest d_{lo} value alive — that is, initially, $S_a = \{o_a\}$ and $S_d = S \setminus \{o_a\}$.

In the main loop that constitutes the search, the object in S_a with the smallest d_{lo} value is picked as the next object whose distance is computed and the d_{lo} values of the objects in S_a are updated. Then, in an inner loop, the objects in S_d are considered in order of their d_{lo} value (i.e., which was based on the initial object o_1), and made alive (i.e., moved from S_d to S_a) if their d_{lo} value is lower than the minimum of d_n and d_a, where d_n is the distance of the current candidate nearest neighbor and d_a is the minimum d_{lo} of an object in S_a (note that d_a may change in each iteration of the inner loop)[28]. Note that ROAESA

[28] Vilar [69] employs a performance improvement technique, in which all of the objects in S are sorted in the preprocessing step of AESA/LAESA on the basis of their distance from o_1. It can be shown that this means that all alive objects lie in consecutive locations in the sorted array, so that the next object to become alive will be one of the objects just beyond the region of alive objects.

has no effect for range searching as in this case d_n is replaced by ε and now S_a is the set of all elements of S_u that have not been eliminated by virtue of their $d_l o$ values being greater than ε.

Interestingly, some of the search hierarchies that we devised for AESA and LAESA (see Sections 4.6.1 and 4.6.2) are related to Vilar's technique, as they also aim at reducing the amount of updating in a somewhat analogous, but more powerful, manner. In particular, in some of the search hierarchies that we proposed, S_u is partitioned into any number of subsets rather than just two (i.e., the alive and not alive objects in ROAESA), where a different number of objects in S_c are used to define d_{lo} for each subset.

5 Comparison to Existing Nearest Neighbor Algorithms

In Sections 3 and 4 we outlined how to perform incremental nearest neighbor search for a database of complex objects using different representations. k-nearest neighbor algorithms have already been proposed for most of these representations (some of which are limited to $k = 1$). However, in all cases, the performance of our incremental nearest neighbor algorithm is at least as good as (and often considerably better than) these existing methods when determining the same number of neighbors, as measured by the number of distance computations. This is a direct consequence of Lemma 2 in Section 2.5, since we designed the search hierarchy for each representation in such a way that the algorithm would be correct (in the case of a mapping-based approach, the mapping must be contractive). Furthermore, the search hierarchies were constructed so that the number of distance computations (using d) performed by a single iteration of the loop in Figure 3 is no larger than would be performed by any existing k-nearest neighbor algorithm in the equivalent situation.

The primary advantage of an incremental nearest neighbor algorithm over a k-nearest neighbor algorithm for the same representation is revealed in circumstances when the number of neighbors desired is unknown in advance. In that case, when using a k-nearest neighbor algorithm, we must guess the value of k. If k is too small, the algorithm must be reapplied with a larger value; if k is too large, the algorithm wastes effort computing neighbors that are not needed. In either case, wasted effort results. Moreover, even when the number of neighbors is known in advance, with an incremental algorithm we can show the user the first few neighbors before all k neighbors have been found, whereas with a k-nearest neighbor algorithm, we must wait until completion of the search. This is important for interactive query interfaces when the distance function d is very expensive to compute. For example, in [41], it was reported that it took an average of almost 13 seconds to compute the morphological distance function.

The price that must be paid for the advantages of the incremental algorithm is that the overhead of the algorithm may be higher than that of non-incremental algorithms for the same representation. For the most part, the additional overhead is due to priority queue operations. However, with an efficient priority queue implementation, the cost of manipulating the priority queue should not be a very significant factor in the execution cost of the algorithm, since computing distances using the distance function d is generally much more expensive than priority queue operations. This is especially true if parts of the representation reside on disk and must be accessed at query time (e.g., if we use an R-tree in the case of a mapping-based approach or an M-tree in the case of distance-based indexing). Furthermore, the additional overhead is often offset (completely or partially) by better pruning achieved by the incremental algorithm, and by not needing to maintain data structures in the corresponding k-nearest neighbor algorithm that are replaced by the priority queue.

Several algorithms proposed for nearest neighbor search in the spatial domain apply a best-first search strategy, like the incremental nearest neighbor algorithms proposed by us [35, 36] and Henrich [32]. An early k-nearest neighbor algorithm is known as Elias's algorithm (e.g., [58]) which partitions the data based on a grid and then proceeds to search the grid cell containing the query point q and its immediate neighboring grid cells in the order of their distance from q. Lower bounds on the distance from a grid

cell to q are used to eliminate most of the grid cells from consideration once a candidate set of k nearest neighbors has been found. A version of this algorithm was proposed for the VA-file [72] which deals with high-dimensional data. Another early algorithm due to Friedman, Baskett, and Shustek [26] is an example of a mapping-based approach which employs dimensionality reduction. It sorts the data on the basis of just one coordinate (i.e., feature) f, and then processes the objects in the order of their f-distance from q until k objects have been found. The fact that the mapping is contractive enables this method to eliminate objects from consideration once a set of k candidates has been found. The approximate nearest neighbor algorithm of Arya et al. [2] also applies best-first traversal.

In this section, we briefly describe several nearest neighbor and k-nearest neighbor algorithms that have been proposed for similarity search, and compare them with our incremental nearest neighbor algorithm (Section 2) as used with the search hierarchies described in Sections 3 and 4. In particular, Section 5.1 reviews existing nearest neighbor algorithms for the mapping-based approach while Section 5.2 reviews existing nearest neighbor algorithms for distance-based indexes.

5.1 Existing Nearest Neighbor Algorithms for the Mapping-Based Approach

Two filter-and-refine k-nearest neighbor algorithms have been proposed for the mapping-based approach. Both require that the mapping F be contractive (see Section 3.2) for the results to be correct, i.e., that $\delta(F(o_1),F(o_2)) \leq d(o_1,o_2)$ for all $o_1,o_2 \in S$.

The algorithm due to Korn et al. [41] first performs a k-nearest neighbor query on the spatial index that represents $F(S)$, using $F(q)$ as a query object and distance function δ (see Section 3.2), resulting in a set R'' of k candidate nearest neighbors of q. Next, the algorithm computes the actual distances (based on d) of all the objects in R'' and determines the distance of the object farthest from q, $\varepsilon = \max_{o \in R''}\{d(q,o)\}$. A range query is then performed on $F(S)$ with a query radius of ε, resulting in a candidate set R'. Finally, the set R of the k nearest neighbors of q is determined by computing the distances of all objects in R' and retaining the k objects that are closest to q.

The correctness of this algorithm can be shown as follows. By the contractiveness of F and the definition of ε, we know that $\delta(F(q),F(o)) \leq d(q,o) \leq \varepsilon$ for all $o \subset R''$. Thus, since R' is the set of all o in S for which $\delta(F(q),F(o)) \leq \varepsilon$, we have $R'' \subset R'$. Therefore, since R'' contains k objects, R' contains at least k objects, and thus the k nearest neighbors of q must be in R'. The drawback of the algorithm is that ε may overestimate the distance of the k^{th} nearest neighbor of q by a considerable margin, as shown by Seidl and Kriegel [60], so R' may be significantly larger than necessary. Furthermore, the fact that two queries are issued to the spatial index that represents $F(S)$ (i.e., the k-nearest neighbor query yielding R'' and the range query yielding R') means that some duplication of effort is inevitable.

Seidl and Kriegel [60] proposed an improved algorithm that is partially based on incremental nearest neighbor search. In particular, the algorithm performs an incremental nearest neighbor query on the spatial index storing $F(S)$, using $F(q)$ as the query object and a distance measure δ (see Section 3.2). As the algorithm obtains the neighbors one by one, it computes the actual distance of each object using d, and inserts the objects into a list L of the candidate nearest neighbors of q (termed the *candidate list*). If this insertion causes L to contain more than k objects, the object farthest from q (based on d) is discarded. Clearly, if all the objects in S were inserted into the candidate list L in this way, L would eventually contain the actual k nearest neighbors. However, since the mapped objects are obtained from the incremental nearest neighbor query in order of the values of $\delta(F(q),F(o))$, the algorithm can usually be terminated long before inserting all the objects into L. In particular, once $\delta(F(q),F(o)) > D_k$ for the object o just obtained from the incremental nearest neighbor query, where D_k is the distance of the current k^{th} candidate nearest neighbor, contractiveness of F guarantees that $d(q,o') \geq \delta(F(q),F(o')) > D_k$ for all objects o' that have not yet been retrieved. Thus, when this condition arises, the candidate list L contains the actual k nearest neighbors of q and the search can be halted.

Our incremental nearest neighbor algorithm, described in Section 3.3, is clearly closely related to the *k*-nearest neighbor algorithm of Seidl and Kriegel. However, our algorithm takes the incremental approach a step further by integrating the candidate list L into the priority queue used by the incremental nearest neighbor algorithm for the spatial index. In other words, when our algorithm encounters an object o (i.e., when $F(o)$ reaches the front of the priority queue), instead of inserting o into a separate candidate list, o is inserted into the priority queue, using the actual distance $d(q,o)$ as a key. In this way, we obtain an overall incremental process, whereas the algorithm of Seidl and Kriegel is a *k*-nearest neighbor algorithm, requiring the number of desired neighbors to be known in advance.

For obtaining any fixed number k of neighbors (using our algorithm, the process is simply terminated once k objects have been output), the two algorithms have very similar performance characteristics. In particular, let o_k be the k^{th} nearest neighbor of q. Both our incremental algorithm and the *k*-nearest neighbor algorithm compute the distances only for objects o such that $\delta(F(q),F(o)) \leq d(q,o_k)$. However, our algorithm has a slight edge in that it sometimes performs fewer spatial index node accesses and δ distance computations than Seidl and Kriegel's *k*-nearest neighbor algorithm. To see this, observe that Seidl and Kriegel's algorithm terminates only after retrieving the first mapped object $F(o_t)$ such that $\delta(F(q),F(o_t)) > d(q,o_k)$. Therefore, that algorithm must access spatial index nodes at a distance from $F(q)$ of up to $\delta(F(q),F(o_t)) \geq d(q,o_k)$. In contrast, our integrated incremental nearest neighbor algorithm only accesses nodes with distances from $F(q)$ of up to $d(q,o_k)$ (since o_k is the last element obtained from the priority queue). Observe that using Seidl and Kriegel's algorithm results in accessing more nodes only if the next non-object element after o_k to be retrieved from the priority queue in our incremental algorithm represents a node in the spatial index. This is a relatively rare situation, so the two algorithms will usually access exactly the same number of nodes.

5.2 Existing Nearest Neighbor Algorithms for Distance-Based Indexes

Most existing *k*-nearest neighbor algorithms for distance-based indexes use a depth-first *branch and bound* strategy (e.g., [34]). A few algorithms employ best-first search, much like our incremental nearest neighbor algorithm, but the full power of best-first search is not always exploited. We have shown that the search hierarchies that we proposed for distance matrix methods in Section 4.6 compute exactly the same numbers of distances to find the same numbers of neighbors as *k*-nearest neighbor algorithms (which can be viewed as applying a depth-first branch and bound strategy without any backtracking)[29].

Fukunaga and Narendra [28] presented one of the earliest nearest neighbor algorithms for distance-based indexes. Actually, as mentioned in Section 4.3.3, the search structure that they use is partly specific to vector data, but their search algorithm is generally applicable to any index based on hierarchical clustering. In fact, they make some of the same observations that we made in Section 4.1, albeit in a somewhat different form. A generalized version of Fukunaga and Narendra's algorithm for an arbitrary search hierarchy, as defined in Section 2.2, is as follows. The elements in the hierarchy are visited in a depth-first traversal, starting at the root, while maintaining a list L of the current candidate k nearest neighbors. Let D_k be shorthand for the distance between q and the farthest object in L (i.e., $D_k = \max_{o \in L}\{d(q,o)\}$), or ∞ if L contains fewer than k objects. Observe that D_k is monotonically non-increasing over the course of the search traversal, and eventually reaches the distance of the k^{th} nearest neighbor of q. If the element e_t being visited represents an object o (i.e., $t = 0$), then o is inserted into L, causing the removal of the object in L farthest from q if this causes L to contain $k+1$ objects (of course, the object removed may

[29] In order to see this, consider AESA [67] as described in Section 4.6.1, and represent its execution logic by a binary tree. In particular, the first pivot that is picked becomes the root of the tree, while all objects eliminated from the search are associated with the left subtree, and the rest of the objects are associated with the right subtree. Recursively applying the same procedure to the set associated with the right subtree yields a binary tree. In essence, using AESA is analogous to performing a depth-first search on this tree, with no backtracking, as one of the subtrees at each node always corresponds to eliminated objects.

be o itself). When visiting an element e_t, $t \geq 1$, we construct an *active list* $A(e_t)$ of child elements of e_t, ordered by distance from q as determined by the appropriate distance function d'_t for each child element $e_{t'}$ of type t'. Next, the elements in $A(e_t)$ are visited recursively in order, until all have been visited or until reaching an element $e_{t'} \in A(e_t)$ such that $d_{t'}(q, e_{t'}) > D_k$[30], at which time the traversal backtracks to the parent of e_t, or terminates if e_t is the root. Observe that L gets updated in the recursive visits to the children of e_t as objects are encountered.

Variants of this general depth-first k-nearest neighbor algorithm have been presented for the vp-tree [74, 17], the sa-tree [50], and a number of other structures. However, such depth-first algorithms may achieve much less pruning of the search space and thereby perform substantially worse, as measured by the number of visited search hierarchy elements, than algorithms that apply best-first search on the same search hierarchy, as is done by our incremental algorithm. To see why, note that a depth-first algorithm must make local decisions about visiting elements, choosing among the children of a single element (the elements in $A(e_t)$ that have not yet been visited) based on the current value of D_k. In contrast, a best-first algorithm makes global decisions about what elements to visit. The global list maintained by the best-first algorithm essentially corresponds to the union of the unvisited portions of the active lists $A(e_t)$ in the depth-first algorithm for all elements e_t on the path from the root to the currently visited element. Thus, while the best-first approach visits an element e_t only if $d_t(q, e_t) \leq d(q, o_k)$ where o_k is the k^{th} nearest neighbor (since the elements are visited in order of distance; see Section 2.5), the depth-first approach may visit an element $e_{t'}$ with $d_{t'}(q, e_{t'}) > d(q, o_k)$, since D_k may be much larger than $d(q, o_k)$ for much of the duration of the search, especially early on. Notice that these observations are also true for variants of the depth-first algorithms that attempt to improve performance by introducing aggressive pruning (effectively by reducing D_k), such as was described for the vp-tree by Chiueh [17][31]. While such strategies can narrow the performance gap relative to a best-first algorithm, they cannot visit any fewer elements in the search hierarchy than a best-first algorithm without risking missing some of the k nearest neighbors of q. In fact, when such an aggressive algorithm detects that too much pruning has taken place (which may well be a common occurrence), the search hierarchy must be traversed again, this time with less aggressive pruning. Hence, the average performance gain of aggressive pruning may be modest.

Ciaccia et al. [19] proposed a k-nearest neighbor algorithm for the M-tree that applies best-first traversal, and thus is somewhat similar to our incremental nearest neighbor algorithm as presented in Figure 3. We might think that it could be transformed into an incremental algorithm without any significant algorithmic changes, but this is not the case, because the algorithm has a certain depth-first-like aspect to it in the way it locally prunes a candidate list, and thus does not take full advantage of the best-first traversal. In particular, the k-nearest neighbor algorithm of Ciaccia et al. uses the priority queue (which serves to guide the search) only for the nodes in the M-tree. In contrast, the search hierarchy for incremental nearest search in the M-tree that we proposed in Section 4.4 consists of four types of elements, all of which are handled by the priority queue. Besides node elements, there are object elements, as well as two other types that provide approximate distances of nodes and objects. Instead of placing the objects on the priority queue, the k-nearest neighbor algorithm of Ciaccia et al. places them on a list L of the k candidate nearest neighbors of q, as is done in the depth-first algorithms described above. Thus the condition for terminating the search in their algorithm is $d_2(q, e_2) \geq D_k$,[32] where e_2 represents the

[30] This stopping condition ensures that all objects at the distance of the k^{th} nearest neighbor are reported. If only k objects are to be reported, then the halting condition should be $d_{t'}(q, e_{t'}) \geq D_k$.

[31] Chiueh [17] suggested using an estimate of the lower bound of the nearest neighbor in each subtree to bound the search in that subtree, thus yielding an algorithm that performs "bounded" nearest neighbor search of each subtree (in the reported experiments, a simpler strategy was employed that made use of an estimate of the smallest lower bound over all nodes). Of course, if the lower bound estimate is too low, it may be necessary to reapply the algorithm with a higher estimate.

[32] This condition means that the algorithm will report exactly k neighobrs regardless of how many objects are at the same distance from q as the k^{th} nearest neighbor. If it is desired to report all of the objects that are at the same distance from q as the

node that was most recently retrieved from the priority queue, d_2 is as defined in Equation 8, and D_k is the distance of the farthest object from q in L (or ∞ if $|L| < k$). This is analogous to placing the objects on the priority queue and terminating the search once the k^{th} object has been retrieved from the priority queue (the primary difference is that in the candidate list approach, at most k objects are remembered at any given time, whereas the priority queue can contain any number of objects in our approach). However, the algorithm of Ciaccia et al. also uses D_k to locally prune entries as it processes nodes obtained from the priority queue. This local pruning takes the place of the approximate node and object elements used in our approach, as mentioned above. In particular, using our notation and the distance functions in Equation 8, when processing a node n, an entry is pruned if $d_1(q,e_1) \geq D_k$ or $d_3(q,e_3) \geq D_k$, depending on whether n is a leaf or a nonleaf node, respectively. As we saw above in the case of the depth-first branch and bound algorithms, such local pruning inevitably leads to more distance computations than a comprehensive best-first solution like ours, since D_k may converge slowly to its final value (i.e., $d(q,o_k)$ where o_k is the k^{th} nearest neighbor of q).

As an example of how use of the algorithm of Ciaccia et al. results in more distance computations than our incremental algorithm, suppose that $D_k = 6$ for the current candidate k^{th} nearest neighbor. Moreover, suppose that we are currently processing a leaf node n with two objects a and b, and that the "parent" object of n is p. Therefore, we know $d(q,p)$ (since it was computed when n was inserted into the priority queue), and we know $d(p,a)$ and $d(p,b)$ (since they are stored in the entries of M-tree node n). Furthermore, we know that $|d(q,p) - d(p,a)|$ and $|d(q,p) - d(p,b)|$ are lower bounds on $d(q,a)$ and $d(q,b)$, respectively. Let us assume that $|d(q,p) - d(p,a)| = 5$, $|d(q,p) - d(p,b)| = 3$, $d(q,a) = 7$, and $d(q,b) = 4$. We first examine the execution of our incremental algorithm. It inserts the "approximate" versions of a and b in the priority queue, and uses the lower bound distance values 5 and 3, respectively. Assume that the next element that is removed from the priority queue is the "approximate" b, which would lead to computing $d(q,b)$ and inserting b on the priority queue. Next, the algorithm removes b from the queue, which is the next neighbor. If the search is now terminated, then the computation of $d(q,a)$ has been avoided.

To see why our incremental algorithm is better in this example, we point out that at the time the leaf node n was processed, the algorithm of Ciaccia et al. would compute both $d(q,a)$ and $d(q,b)$, since their lower bounds are both smaller than $D_k = 6$. In essence, Ciaccia et al.'s use of the lower-bound distances is local (as they are used to make an immediate decision about whether to compute the actual distance), whereas our incremental nearest neighbor algorithm makes a global use of them (by putting them on the priority queue, and only computing the actual distance once they are removed from the queue). In other words, our incremental algorithm defers the computation of the distances via the use of the approximate objects and nodes. Nevertheless, both our incremental algorithm and the algorithm of Ciaccia et al. explore the same number of M-tree nodes, as in both algorithms these nodes are only explored once they reach the head of the priority queue, which happens in the same way in the two algorithms. However, use of the approximate node for n does result in deferring the computation of $d(q,p)$ and replacing it by a quantity that involves $|d(q,p') - D|$ which has already been computed before processing n. Thus a distance computation has also been possibly avoided.

6 Performance Evaluation

In this section, we report some preliminary results from an empirical study of the performance of our incremental nearest neighbor algorithm when applied to similarity search using both a mapping-based approach and distance-based indexing. We also compare the performance of the incremental nearest neighbor algorithm with that of existing k-nearest neighbor algorithms.

k^{th} nearest neighbor, then the condiditon for terminating the search is modified to be $d_2(q,e_2) > D_k$.

6.1 Experimental Environment

In the experiments, we used the R*-tree to test the multistep algorithms of Section 3. The R*-tree implementation was based on the freely distributed GiST indexing framework [31]. For experiments with the M-tree, we used the GiST-based implementation that is made available by its inventors [19]. To build the M-trees used in the experiments, we used their bulk-loading algorithm [20]. The source code was written in C++ and was built with the GNU C++ compiler with maximum optimization (-O3). We ran the experiments on a Sun Ultra 1, which is rated at 6.17 SPECint95 and 11.80 SPECfp95. In one of the graphs, we report execution times of a set of queries, but in the others we report the execution cost in terms of the number of distance computations or node I/Os.

For the mapping-based approach, we stored the feature vectors (the mapped objects) directly in the leaf nodes of the R*-tree, while the actual object data were stored in a simple table in an external file (termed an *object table*). In the M-tree, we also employed an object table, and stored pointers into this table in the tree itself (for both routing and data objects). Thus, unlike the experiments reported in [19] where the data is stored directly in the tree, the M-trees used in our experiments had the same maximum fan-out regardless of dimension. In both cases, we used node sizes of 4K, which led to a maximum fan-out of about 25 for the R*-tree assuming 10D feature vectors, and about 100 for the M-tree. While it may sometimes be inefficient to store the objects outside the M-tree, thus usually requiring disk accesses for distance computations (unless the objects are already in the buffers), this does not affect the number of distance calculations and node I/Os for queries. However, different values of maximum fan-out can potentially affect both, as they lead to a different tree structure. Nevertheless, although we do not show this, we found that reducing the fan-out of the M-tree down to 25 usually did little to reduce the number of distance calculations for queries, while it increased the number of node I/Os (as would be expected).

We used three types of data in our experiments: color histograms, tumor images, and synthetic data.

- The color histograms were taken from an image database [60], and are of dimensions 64, 112, and 256. The 64D and 256D sets contained about 12000 items while the 112D set contained about 8000. For the mapping-based approach, we used the Karhunen-Loève Transform (KLT) to extract the 10 most significant coordinates for storage in an R*-tree. As a distance metric, we used Euclidean distance. The KLT method can be shown to result in feature vectors whose Euclidean distances lower-bound the Euclidean distances of the original vectors (see Section 3.1.2).

- The tumor image data came from Korn et al. [41]; we used a set of 5000 images for our experiments. As mentioned in Section 3, the morphological distance measure defined in [41] is very expensive to evaluate. In [41] each distance computation is reported to take about 13 seconds. Using full compiler optimization, some algorithm optimizations and a faster machine, we got the time down to an average of about 1/3 of a second. This is still a long time for a large database, and in particular much longer than a node I/O, so this application represents an interesting extreme case. The feature vectors (stored in the R*-trees) derived from the images are 11-dimensional and the distance metric used is L_∞.

- The synthetic data sets contained multidimensional points that formed 10 normally-distributed clusters [19]. They were generated for dimensions ranging from 20 to 100, and contained 10,000 points each. As before, we used the Euclidean distance metric. Again, we extracted the 10 most significant coordinates to store in the R*-tree for the mapping-based approach.

6.2 Benefit of Incremental Aspect

In this section we show experiments demonstrating the benefit of the incremental nature of our algorithms. As mentioned in Section 1, two important situations where incremental nearest neighbor algo-

rithms prove useful are 1) to allow quick display of the first results to the user in interactive applications, and 2) when the number of desired neighbors is unknown in advance, e.g., when processing a complex query where the "nearest" condition is only one of the conditions. In the first situation, the user may for example want to view the 10 most similar images. Using a *k*-nearest neighbor algorithm, we would have to wait until the entire search has been completed, whereas with an incremental nearest neighbor algorithm we can quickly display the closest image before the next closest image has been determined. In the second situation, it is awkward to use a *k*-nearest neighbor algorithm since we do not know an appropriate *k*, so we must guess. Too high a guess causes extraneous processing, whereas too low a guess may force us to re-invoke the algorithm. One way to proceed is to first set *k* to a low number, say 5, and then multiply by 2 each time we must re-invoke the algorithm (e.g., if the 6^{th} neighbor is requested).

Figures 19 and 20 show the numbers of distance computations and node I/Os for queries on the 64D histogram data set using the mapping-based approach. The figures show the cost of finding from 1 to 20 of the nearest neighbors for the multistep incremental nearest neighbor algorithm (using the R*-tree), and the cost of finding the 10 and 20 nearest neighbors using the multistep *k*-nearest neighbor algorithm, as well as the cost of using it when the number of desired neighbors is unknown, using an initial value of $k = 5$, and multiplying by 2 each time. Figures 21 and 22 show the same information for the M-tree algorithms. The figures dramatically show the advantage of using the incremental nearest neighbor algorithms over the *k*-nearest neighbor algorithms, especially as regards the number of distance computations. For the mapping-based approach, the INN algorithm reports the first neighbor using only 17% of the number of distance computations needed to find the first 10 neighbors using the *k*-NN algorithm, and 11% for the first 20.

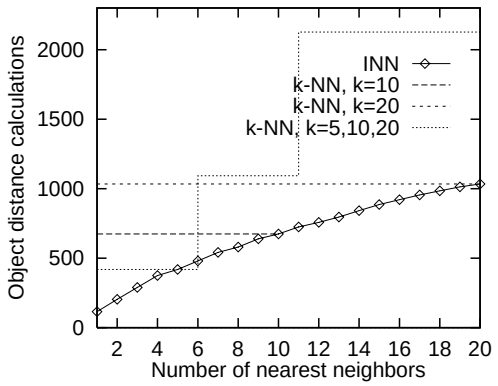

Figure 19: Number of distance computations for 64D histogram data and a varying number of neighbors using an R*-tree index (mapping-based approach).

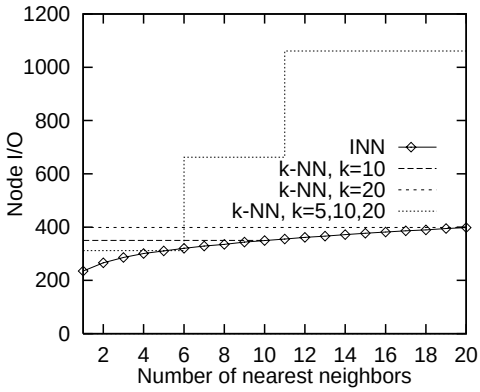

Figure 20: Number of node I/Os for 64D histogram data and a varying number of neighbors using an R*-tree index (mapping-based approach).

If the cost of distance computations is much higher than that of node I/Os, the importance of reducing the number of distance computations increases. This is the case for the tumor shape data. Figure 23 reports the execution times for queries on a tumor shape database consisting of 5000 images using both approaches, the mapping based approach using the R*-tree (R) and the M-tree (M). It shows the cost of finding from 1 to 10 neighbors using the incremental nearest neighbor algorithms and the cost of finding the 10 nearest neighbors using the *k*-nearest neighbor algorithms. While this graph does not show as great an advantage of the incremental approach as some of the previous graphs, the advantage is still substantial. In both cases, the incremental nearest neighbor algorithm reports the first nearest neighbor in about 65% of the time it takes to find the 10 nearest neighbors. Interestingly, for this data set, the

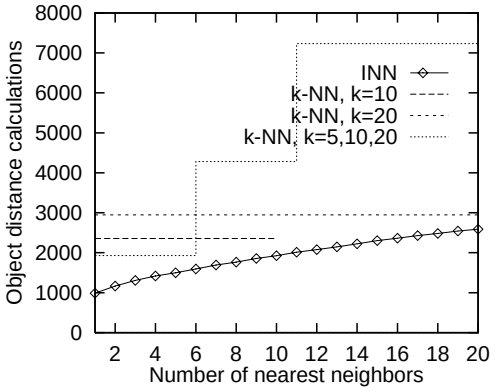

Figure 21: Number of distance computations for 64D histogram data for varying number of neighbors using an M-tree index.

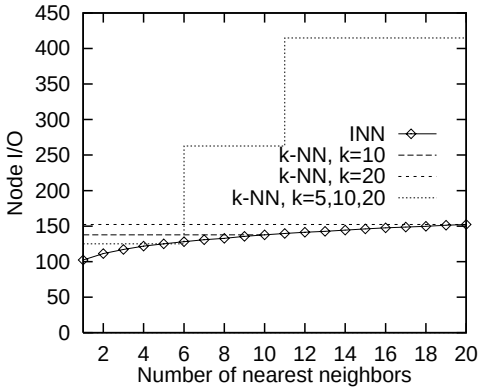

Figure 22: Number of node I/Os for 64D histogram data for varying number of neighbors using an M-tree index.

R*-tree based method is up to 50% faster than using the M-tree.

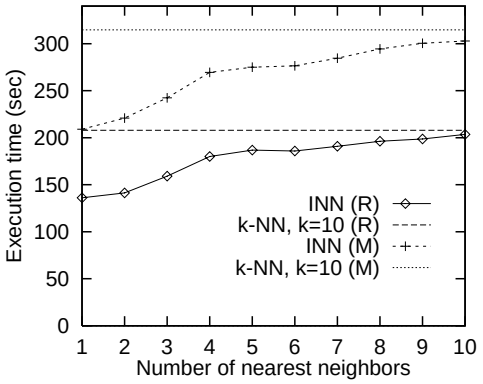

Figure 23: Execution time for tumor shape data and a varying number of neighbors.

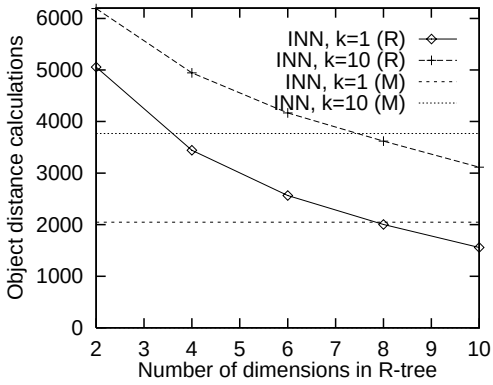

Figure 24: Number of distance computations for 112D histogram data and a varying number of dimensions in an R*-tree (R; mapping-based approach) vs. an M-tree (M).

6.3 Comparison of Mapping-based Approach and Distance-Based Indexing

If we compare Figures 19 and 21 we can see that the R*-tree based method performed many fewer distance computations than the M-tree based method. In the experiment reported in these figures, the number of dimensions of the mapped objects stored in the R*-tree is 10. An interesting question is what happens when we vary the number of dimensions of the mapped objects. Clearly, the lower the number of dimensions, the poorer an approximation of the objects we get in the mapped objects. In Figure 24 we show the results of varying the dimensionality of the mapped objects in the R*-tree for the 112D histogram data, for two different numbers of neighbors (1 and 10). We also show, for comparison, the result for the same queries in the M-tree. As we might have expected, the number of distance computations decreases with increasing number of dimensions. At eight dimensions, the R*-tree method is already better

than the M-tree. For the 64D and 256D histogram data, we found that the cut-off point was four dimensions.

6.4 *k*-nearest Neighbor Queries

In Section 5.1, we showed that when applied to the mapping-based approach, our incremental nearest neighbor algorithm performs exactly as many distance computations as the *k*-nearest neighbor algorithm of Seidl and Kriegel [60], and sometimes fewer node I/Os, but never more. Similarly, we showed in Section 5.2 that when applied to the M-tree, our algorithm performs exactly as many node I/Os as the M-tree *k*-nearest neighbor algorithm of Ciaccia et al. [19], and usually fewer distance computations, but never more.

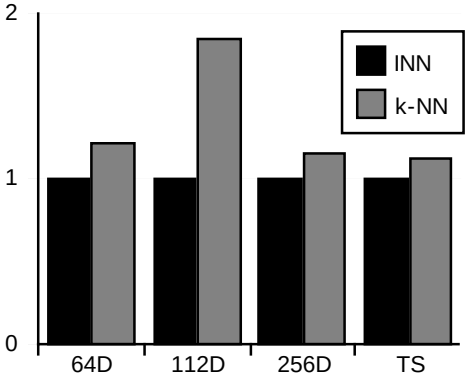

Figure 25: Relative numbers of distance computations for 10-NN queries implemented with the INN and *k*-NN methods in different datasets using an M-tree index.

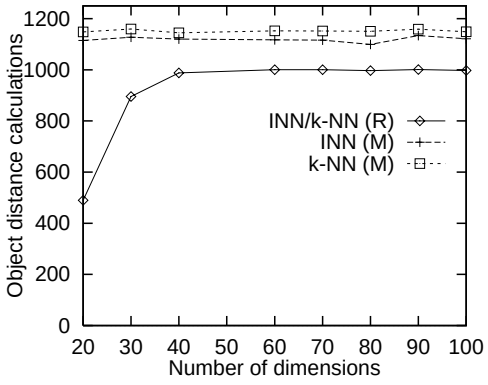

Figure 26: Numbers of distance computations for 10-NN queries on synthetic data sets of varying dimension.

Figure 25 shows the relative numbers of distance computations performed by the INN and *k*-NN algorithms for the M-tree when computing 10-NN queries on the three histogram datasets and on the tumor shape dataset. In most cases, the *k*-NN algorithm performed roughly 15-20% more distance computations, but for the 112D histogram set, it performed almost 85% more. The number of node I/Os performed by the multistep INN algorithm is usually at most 1-2% less than that of the multistep *k*-NN algorithm, so we do not show a graph for that case.

Figures 26 and 27 show the numbers of distance computations and node I/Os for 10-NN queries on synthetic data sets of 10000 vectors of dimension 20 through 100. In the figures, we show the results of the INN and *k*-NN algorithms together when they are identical or very similar. The performance of both approaches does not appear to depend much on the dimensionality of the data. This is perhaps not surprising given that the data sets are constructed with the same parameters (i.e., using 10 clusters of vectors and the same distribution around each cluster). However, the number of distance computations for the mapping-based approach is significantly lower for 20 and 30 dimensions. This is because in this case, the 10 dimensional feature vectors stored in the R*-tree evidently retain more of the information than for higher dimensions, thus effecting better filtering.

7 Concluding Remarks

We have introduced a general framework for performing incremental nearest neighbor search, and have applied it to two classes of similarity search methods, mapping-based approaches and distance-based

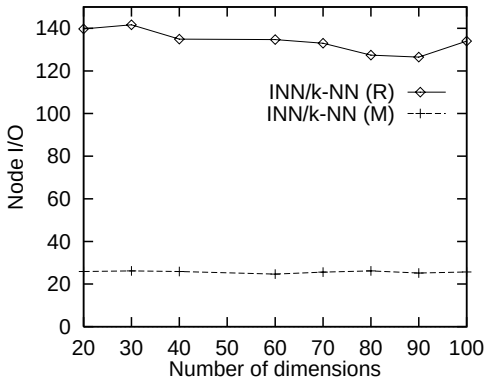

Figure 27: Number of node I/Os for 10-NN queries on synthetic data sets of varying dimension.

indexes. Similarity search involves finding objects in a data set $S \subset \mathbb{U}$ that are similar to a query object $q \in \mathbb{U}$, based on some distance measure d. Our algorithm reports the result objects one by one in order of similarity, with as little effort as possible expended to produce each new result object. The algorithm was shown to be correct for all the indexing methods that we applied it to, given a suitable definition of a "search hierarchy" on which the algorithm operates. In the case of incremental nearest neighbor search, correctness means that the objects are reported in order of increasing (or, more accurately, non-decreasing) distance from the query object. However, we discussed methods that aim to improve the performance of the algorithm, by relaxing the correctness criterion, implying that the objects may be reported somewhat out of order. For many applications, such approximate behavior is acceptable, within limits, and users are willing to trade off accuracy for speed. In addition to discussing such an approximate version of the algorithm, we also described how the algorithm can exploit limits (upper and lower) on the distances of the desired objects and limits on the cardinality of the result, thus incorporating features of range search and k-nearest neighbor search. Furthermore, we showed how the algorithm can be varied to produce the objects in order of decreasing distance, so that the farthest neighbor is reported first.

We showed that our incremental nearest neighbor algorithm is optimal with respect to the search hierarchy that it is applied to, in that as small a portion of the search hierarchy is visited as possible, for any number of neighbors. In particular, for obtaining k neighbors of a query object q, the portion of the search hierarchy visited is the same as would be visited by a range query with query object q and radius equal to the distance of the k^{th} neighbor of q. When applied to similarity search structures, this means in practice that it performs the fewest possible distance computations (on the objects in S using the distance measure d), and if the underlying indexing structure is disk-resident, the fewest possible I/O operations. Of course, this form of optimality does not imply that the algorithm will perform the least amount of work in some absolute sense; the actual performance depends on the underlying indexing structure and the definition of the search hierarchy.

During our discussion, we surveyed a number of different mapping methods and distance-based indexing methods. We also discussed existing nearest neighbor and k-nearest neighbor algorithms for these methods, and showed that, in most cases, these algorithms achieve less pruning of the search hierarchy for any fixed number of neighbors (but never more, of course). Furthermore, our algorithm has the distinct advantage of being incremental, which gives it much better performance for *distance browsing* queries, wherein we browse through a database based on distance and may terminate the browsing at any time — that is, the number of desired objects is unknown in advance. An experimental study confirms our reasoning, showing considerable improvement over the k-nearest neighbor algorithms in areas where the incremental nature of the algorithm can be exploited. Another benefit of our algorithm in the setting of

interactive query interfaces, even in cases where the number of desired neighbors is fixed in advance, is that the first few neighbors can be reported to the user as soon as the algorithm determines them. Our experiments also confirmed that the first neighbors often can be determined much sooner than, say, the 20^{th} neighbor. Admittedly, in some cases, the cost of determining only the first neighbor may be very high. In particular, let $C(m)$ be the cost of determining m neighbors, and let k be the desired number of neighbors. The benefit of the incremental algorithm depends on the value of $C(k)/C(1)$: if this value is large, we get the first neighbor much more quickly than the k^{th} neighbor. Unfortunately, the greater the inherent dimensionality of the data set S, the smaller $C(k)/C(1)$ tends to be.

Our experiments also provide a comparison between the two approaches to similarity search. Under the parameters that we used, the mapping-based approach always performed fewer distance computations compared to using the distance-based index, but the mapping-based approach led to more node I/O operations. This appears to suggest that the mapping-based approach is preferable in situations where the distance function that measures the similarity of two objects is expensive to compute. Naturally, this depends on the quality of the filtering that the mapped objects provide during query execution. Nevertheless, we found that a remarkably small number of dimensions for the mapped objects was adequate for the data sets that we used. In fact, the query execution time when using the mapping-based approach was never higher than that for distance-based indexing (given a suitable dimensionality of the mapped object).

An important future task in this area is to develop cost models for our algorithm in various settings. Such cost models necessarily depend on the particular indexing structure being employed, but some general assumptions can possibly be formulated that apply reasonably well to a large class of structures. There are three important parameters to such a cost model. First, the expected number k of desired neighbors of the query object q. Second, the expected distance r of the k^{th} nearest neighbor of q. Third, the expected cost C of performing a range query with query radius r. Clearly, the measure C of the cost of the range query must include the number of distance computations on S, since they are typically expensive, but for a disk-resident indexing structure, we must also take into account the number of I/O operations. The relative weight of these two factors clearly depends on the relative cost of distance computations vs. I/O operations. Some headway has been made in recent years in developing cost models for proximity queries, e.g., for high-dimensional vector spaces [4] and for M-trees [20]. Based on some simplifying assumptions, this work focuses on estimating the r parameter based on k and/or the C parameter based on r. However, the assumptions do not apply to all similarity search methods, so more remains to be done. In situations where the number of desired neighbors is not precisely known in advance, it will also be necessary to estimate k. A reasonable approach might be to take a "trailing average" of the number of requested neighbors in some of the recent queries.

Other future work includes performing more experiments using other mapping methods and distance-based indexes on more varied data sets. Thus, we would aim at providing an empirical basis for choosing the appropriate method (e.g., whether to use a mapping-based approach or a distance-based index) for a given application using our algorithm.

8 Acknowledgments

We are grateful to Flip Korn for providing the tumor image data used in our experiments and code for performing the distance computations, and to Thomas Seidl for providing the color histogram data. Also, we wish to thank the authors of the M-tree [19] and GiST [31] for making their code freely available (at `http://www-db.deis.unibo.it/~patella/MMindex.html` and `http://gist.cs.berkeley.edu/`, respectively).

References

[1] M. Ankerst, G. Kastenmüller, H.-P. Kriegel, and T Seidl. 3D shape histograms for similarity search and classification in spatial databases. In *Advances in Spatial Databases — Sixth International Symposium, SSD'99*, R. H. Güting, D. Papadias, and F. H. Lochovsky, eds., pages 207–226, Hong Kong, July 1999. Also Springer-Verlag Lecture Notes in Computer Science 1651.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6):891–923, November 1998.

[3] R. A. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, pages 198–212, Asilomar, CA, June 1994. Also Springer-Verlag Lecture Notes in Computer Science 807.

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

[5] S. Berchtold, C. Böhm, H.-P. Kriegel, J. Sander, and H. V. Jagadish. Independent quantization: An index compression technique for high-dimensional data spaces. In *Proceedings of the 16th IEEE International Conference on Data Engineering*, pages 577–588, San Diego, CA, February 2000.

[6] 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 (VLDB)*, T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, eds., pages 28–39, Mumbai (Bombay), India, September 1996.

[7] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proceedings of the ACM SIGMOD Conference*, J. Peckham, ed., pages 357–368, Tucson, AZ, May 1997.

[8] T. Bozkaya and M. Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Transactions on Database Systems*, 24(3):361–404, September 1999.

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

[10] E. Bugnion, S. Fei, T. Roos, P. Widmayer, and F. Widmer. A spatial index for approximate multiple string matching. In *Proceedings of the 1st South American Workshop on String Processing (WSP'93)*, R. Baeza-Yates and N. Ziviani, eds., pages 43–53, Belo Horizonte, Brazil, September 1993.

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

[12] K. Chakrabarti and S. Mehrotra. High dimensional feature indexing using hybrid trees. Technical Report TR-MARS-98-14, Department of Information and Computer Science, University of California, Irvine, CA, July 1998.

[13] K. Chakrabarti and S. Mehrotra. The hybrid tree: an index structure for high dimensional feature spaces. In *Proceedings of the 15th IEEE International Conference on Data Engineering*, pages 440–447, Sydney, Australia, March 1999.

[14] E. Chávez, J. Marroquín, and G. Navarro. Overcoming the curse of dimensionality. In *European Workshop on Content-Based Multimedia Indexing*, pages 57–64, Toulouse, France, October 1999.

[15] E. Chávez, G. Navarro, R. Baeza-Yates, and J.L. Marroquín. Searching in metric spaces. Technical Report TR/DCC-99-3, DCC, University of Chile, Santiago, Chile, June 1999.

[16] J.-Y. Chen, C. A. Bouman, and J. C. Dalton. Hierarchical browsing and search of large image databases. *IEEE Transactions on Image Processing*, 9(3):442–455, March 2000.

[17] T. Chiueh. Content-based image indexing. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, J. Bocca, M. Jarke, and C. Zaniolo, eds., pages 582–593, Santiago, Chile, September 1994.

[18] P. Ciaccia and M. Patella. Bulk loading the M-tree. In *Proceedings of the 9th Australasian Database Conference (ADC'98)*, pages 15–26, Perth, Australia, February 1998.

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

[20] P. Ciaccia, M. Patella, and P. Zezula. A cost model for similarity queries in metric spaces. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 59–68, Seattle, WA, June 1998.

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

[22] C. A. Duncan, M. Goodrich, and S. Kobourov. Balanced aspect ratio trees: Combining the advantages of *k*-d trees and octrees. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 300–309, Baltimore, MD, January 1999.

[23] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3(3/4):231–262, 1994.

[24] 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.

[25] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi. Vector approximation based indexing for non-uniform high dimensional data sets. In *Proceedings of the Ninth International Conference on Information and Knowledge Management (CIKM)*, pages 202–209, McLean, VA, November 2000.

[26] J. H. Friedman, F. Baskett, and L. J. Shustek. An algorithm for finding nearest neighbors. *IEEE Transactions on Computers*, 24(10):1000–1006, October 1975.

[27] A. W.-C. Fu, P. M.-S. Chan, Y.-L. Cheung, and Y. S. Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *VLDB Journal*, 9(2):154–173, June 2000.

[28] 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.

[29] 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.

[30] J. L. 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.

[31] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, U. Dayal, P. M. D. Gray, and S. Nishio, eds., pages 562–573, Zurich, Switzerland, September 1995. `http://gist.cs.berkeley.edu/`.

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

[33] 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.

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

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

[36] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999. Also Computer Science TR-3919, University of Maryland, College Park, MD.

[37] G. R. Hjaltason and H. Samet. Contractive embedding methods for similarity searching in metric spaces. Computer Science TR-4102, University of Maryland, College Park, MD, February 2000.

[38] G. Hristescu and M. Farach-Colton. Cluster-preserving embedding of proteins. Technical report, Rutgers University, Piscataway, NJ, 1999.

[39] K. V. R. Kanth, D. Agrawal, A. El Abbadi, and A. Singh. Dimensionality reduction for similarity searching in dynamic databases. *Computer Vision and Image Understanding*, 75(1/2):59–72, July/August 1999.

[40] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, vol. 3. Addison-Wesley, Reading, MA, second edition, 1998.

[41] 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 (VLDB)*, T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, eds., pages 215–226, Mumbai (Bombay), India, September 1996.

[42] J. B. Kruskal and M. Wish. Multidimensional scaling. Technical report, Sage University Series, Beverly Hills, CA, 1978.

[43] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15:215–245, 1995.

[44] S. Maneewongvatana and D. M. Mount. The analysis of a probabilistic approach to nearest neighbor searching. In *Proceedings of the 2001 Workshop on Algorithms and Data Structures (WADS'01)*, Providence, RI, August 2001. To appear, Also Springer-Verlag Lecture Notes in Computer Science 2125.

[45] S. Maneewongvatana and D. M. Mount. An empirical study of a new approach to nearest neighbor searching. In *Third Workshop on Algorithm Engineering and Experimentation (ALENEX'01)*, pages 172–187, Washington, DC, January 2001. Also Springer-Verlag Lecture Notes in Computer Science 2153, `ftp.cs.umd.edu/pub/faculty/mount/Papers/alenex01.ps.gz`.

[46] B. S. Manjunath and W. Y. Ma. Texture features for browsing and retrieval of image data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(8):837–842, August 1996.

[47] L. Micó, J. Oncina, and R.C. Carrasco. A fast branch & bound nearest neighbor classifier in metric spaces. *Pattern Recognition Letters*, 17(7):731–739, June 1996.

[48] L. Micó, J. Oncina, and E. Vidal. An algorithm for finding nearest neighbours in constant average time with a linear space complexity. In *Proceedings of the 11th International Conference on Pattern Recognition*, vol. II, pages 557–560, The Hague, The Netherlands, August–September 1992.

[49] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15(1):9–17, January 1994.

[50] G. Navarro. Searching in metric spaces by spatial approximation. In *Proceedings String Processing and Information Retrieval (SPIRE '99)*, Cancun, Mexico, September 1999.

[51] 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.

[52] H. Noltemeier, K. Verbarg, and C. Zirkelbach. A data structure for representing and efficient querying large scenes of geometric objects: Mb*-trees. In *Geometric Modelling*, pages 211–226, Springer-Verlag, Vienna, Austria, 1993.

[53] B. C. Ooi, K. J. McDonell, and R. Sacks-Davis. Spatial *k-d*-tree: an indexing mechanism for spatial database. In *Proceedings of the Eleventh International Computer Software and Applications Conference COMPSAC*, pages 433–438, Tokyo, Japan, October 1987.

[54] P. van Oosterom. *Reactive data structures for geographic information systems*. PhD thesis, Department of Computer Science, Leiden University, Leiden, The Netherlands, December 1990.

[55] P. van Oosterom and E. Claassen. Orientation insensitive indexing methods for geometric objects. In *Proceedings of the Fourth International Symposium on Spatial Data Handling*, vol. 2, pages 1016–1029, Zurich, Switzerland, July 1990.

[56] J. A. Orenstein. Redundancy in spatial databases. In *Proceedings of the ACM SIGMOD Conference*, pages 294–305, Portland, OR, June 1989.

[57] V. Ramasubramanian and K. K. Paliwal. An efficient approximation-elimination algorithm for fast nearest neighbour search based on a spherical distance coordinate formulation. *Pattern Recognition Letters*, 13(7):471–480, July 1992.

[58] R. L. Rivest. On the optimality of Elias's algorithm for performing best-match searches. *Information Processing 74*, pages 678–681, 1974.

[59] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, NJ, 1994.

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

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

[62] M. Shapiro. The choice of reference points in best-match file searching. *Communications of the ACM*, 20(5):339–343, May 1977.

[63] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[64] C. Traina Jr., A. J. M. Traina, B. Seeger, and C. Faloutsos. Slim-trees: High performance metric trees minimizing overlap between nodes. In *Proceedings of the 7th International Conference on Extending Database Technology — EDBT'2000*, C. Zaniolo, P. C. Lockemann, M. H. Scholl, and T. Grust, eds., pages 51–65, Konstanz, Germany, March 2000. Also Springer-Verlag Lecture Notes in Computer Science 1777.

[65] J. K. Uhlmann. Metric trees. *Applied Mathematics Letters*, 4(5):61–62, 1991.

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

[67] E. Vidal. An algorithm for finding nearest neighbours in (approximately) constant average time. *Pattern Recognition Letters*, 4(3):145–158, July 1986.

[68] E. Vidal. New formulation and improvements of the nearest-neighbour approximating and eliminating search algorithm (AESA). *Pattern Recognition Letters*, 15(1):1–7, January 1994.

[69] J. M. Vilar. Reducing the overhead of the AESA metric-space nearest neighbour searching algorithm. *Information Processing Letters*, 56(5):265–271, December 1995.

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

[71] X. Wang, J. T. L. Wang, K.-I. Lin, D. Shasha, B. A. Shapiro, and K. Zhang. An index structure for data mining and clustering. *Knowledge and Information Systems*, 2(2):161–184, May 2000.

[72] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, A. Gupta, O. Shmueli, and J. Widom, eds., pages 194–205, New York, NY, August 1998.

[73] P. N. Yianilos. Excluded middle vantage point forests for nearest neighbor search. Technical report, NEC Research Institute, Princeton, NJ, July 1998.

[74] P. Y. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 311–321, Austin, TX, January 1993.

[75] F. W. Young and R. M. Hamer. *Multidimensional Scaling: History, Theory, and Applications*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.