Index-Driven Similarity Search in Metric Spaces

GISLI R. HJALTASON and HANAN SAMET
University of Maryland, College Park, Maryland

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 similarity measure. In this article, we focus on methods for similarity search that make the general assumption that similarity is represented with a distance metric d. Existing methods for handling similarity search in this setting typically fall into one of two classes. The first directly indexes the objects based on distances (distance-based indexing), while the second is based on mapping to a vector space (mapping-based approach). The main part of this article is dedicated to a survey of distance-based indexing methods, but we also briefly outline how search occurs in mapping-based methods. We also present a general framework for performing search based on distances, and present algorithms for common types of queries that operate on an arbitrary "search hierarchy." These algorithms can be applied on each of the methods presented, provided a suitable search hierarchy is defined.

Categories and Subject Descriptors: E.1 [**Data Structures**]: *Trees*; H.2.4 [**Database Management**]: Systems—*query processing, multimedia databases*; H.2.8 [**Database Management**]: Database Applications—*image databases*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Hiearchical metric data structures, similarity searching, distance-based indexing, nearest neighbor queries, range queries, ranking

1. INTRODUCTION

Classical database methods are designed to handle data objects that have some predefined structure. This structure is usually captured by treating the various attributes associated with the objects as independent dimensions, and then representing the objects as records. These records are stored in the database using some appropriate model (e.g., relational, object-oriented, object-relational,

This work was supported in part by the National Science Foundation under grants EIA-99-00268, EIA-99-01636, IIS-00-86162, and EIA-00-91474.
Authors' addresses: G. R. Hjaltason, School of Computer Science, University of Waterloo, Waterloo, Ont., Canada N2L 3G1; email: gisli@db.uwaterloo.ca; H. Samet, Computer Science Department, Center for Automation Research, Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742; email: hjs@cs.umd.edu.

hierarchical, network, etc.). The most common queries on such data are exact match, partial match, range, and join applied to some or all of the attributes. Responding to these queries involves retrieving the relevant data. The retrieval process is facilitated by building an index on the relevant attributes. These indexes are often based on treating the records as points in a multidimensional space and using what are called point access methods (e.g., Gaede and Günther [1998] and Samet [1990, 1995]).

More recent applications involve data that has considerably less structure and whose specification is therefore less precise. Some example applications include collections of more complex data such as images, videos, audio recordings, text documents, time series, DNA sequences, etc. The problem is is that usually the data can neither be ordered nor is it meaningful to perform equality comparisons on it. Instead, proximity is a more appropriate retrieval criterion. Such data objects are often described via a collection of features, and the result is called a *feature vector*. For example, in the case of image data, the feature vector might include color, color moments, textures, shape descriptors, etc., all of which are usually described using scalar values. In the case of text documents, we might have one dimension per word, which leads to prohibitively high dimensions. Correcting misspelled text or searching for semantic equivalents is even more difficult. Video retrieval involves finding overlapping frames which is somewhat like finding subsequences in DNA sequences. The goal in these applications is often one of the following:

(1) Find objects whose feature values fall within a given range or where the distance, using a suitably defined distance metric, from some query object falls into a certain range (range queries).
(2) Find objects whose features have values similar to those of a given query object or set of query objects (nearest neighbor queries). In order to reduce the complexity of the search process, the precision of the required similarity can be an approximation (approximate nearest neighbor queries).
(3) Find pairs of objects from the same set or different sets which are sufficiently similar to each other (closest pairs queries).

The process of responding to these queries is termed *similarity searching*, also referred to as *content-based* or *similarity* retrieval. Given a query object q, this involves finding objects that are similar to q in a database S of N objects, based on some similarity measure. Both q and S are drawn from some "universe" $\mathbb{U}$ of objects, but q is generally not in S. In this article, we assume that the similarity measure can be expressed as a distance metric d, such that $d(o_1, o_2)$ becomes smaller as o_1 and o_2 are more similar. Thus, (S, d) is said to be a finite metric space.

As in the case of structured data, the retrieval process is facilitated by building an index on the various features, which are analogous to attributes. Again, these indexes are based on treating the records as points in a multidimensional space and use point access methods. The primary challenge in performing similarity search (e.g., finding the nearest neighbors) when the data is indexed in this way lies in the realization that the process of evaluating the distance

function d is typically rather expensive. Moreover, the "intrinsic" dimensionality of the metric space is often high on account of the many features that are used to describe the data, which is an important difference from structured data.

Searching in high-dimensional spaces can be time-consuming. Interestingly, exact match, partial match, and range queries in high dimensions are relatively easy, from the standpoint of computational complexity, as they do not involve the computation of distance. In particular, such searches through an indexed space usually involve simple comparison tests. However, if we have to examine all of the index nodes, then the process is time-consuming. In contrast, computing similarity, as is the case for nearest neighbor queries, makes use of distance, and the process of computing the distance can be computationally complex in a high-dimensional space. For example, computing the Euclidean distance between two points in a high-dimensional space, say of dimension n, requires n multiplication operations and $n - 1$ addition operations, as well as a square root operation (which can be omitted).

The above discussion has been based on the premise that we know the features that describe the objects (and hence the dimensionality of the underlying feature space). In fact, it is often quite difficult to identify the features and thus we frequently turn to experts in the application domain from which the objects are drawn for assistance in this process. Nevertheless, frequently, the features cannot be easily identified even by the domain experts. In this case, the only information that we have available is the distance function d, which as we pointed out before may be quite expensive to compute, that indicates the degree of similarity (or dis-similarity) between all pairs of objects, given a set of N objects. Sometimes, the degree of similarity is expressed by use of a similarity matrix which contains interobject distance values, for all possible pairs of the N objects.

There are two alternatives when the only information that we have is the distance function. The first is to derive "features" purely based on the inter-object distances (e.g., methods described in Faloutsos and Lin [1995], Hristescu and Farach–Colton [1999], Linial et al. [1995], and Wang et al. [1999]). Thus, given N objects, the goal is to choose a value of k and find a set of N corresponding points in a k-dimensional space, via an appropriate mapping F that is applicable to all elements of $\mathbb{U}$ and thus also to the query objects q, so that the distance between the N corresponding points, using a suitably defined distance function δ, is as close as possible to that given by the original distance function d for the N objects. The rationale for 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 $o' \in F(S)$. Moreover, the attractiveness of such methods is that we can now index the points using multidimensional indexes thereby speeding up the search in $F(S)$. These methods are known as *embedding methods* and they can also be applied to objects represented by feature vectors. In this case, the advantage of using these methods lies in choosing k to be considerably smaller than the number of features. This leads to a reduction in the dimensionality of the feature vector thereby lowering the dimension of the space which is being indexed.

An important property for facilitating similarity search when using embedding methods is that the mapping F be contractive [Hjaltason and Samet 2003b], which implies that it does not increase the distances between objects, that is, that $\delta(F(o_1), F(o_2)) \leq d(o_1, o_2)$ for all $o_1, o_2 \in \mathbb{U}$. In particular, this allows using the mapped objects as a "filter" during query processing without suffering *false dismissals*, and then "refine" the result by using d (e.g., Korn et al. [1996] and Seidl and Kriegel [1998]). For example, the result of applying a range query on $F(S)$ with query object $F(q)$ and radius ϵ includes all the objects in the range query on S with query object q and radius ϵ. Thus, we obtain the exact query result for S by first computing the range query on $F(S)$ and then eliminating objects o with $d(q, o) > \epsilon$. We do not discuss embedding methods in this paper except to point out where they are related to other methods that we present.

The second alternative when the only information that we have is a distance function is to use the given distance function to index the data (i.e., objects) with respect to their distance from a few selected objects. We use the term *distance-based indexing* to describe such methods. The advantage of distance-based indexing methods is that distance computations are used to build the index, but once the index has been built, similarity queries can often be performed with a significantly lower number of distance computations than a sequential scan of the entire dataset, as would be necessary if no index exists. Of course, in situations where we may want to apply several different distance metrics, then distance-based indexing techniques have the drawback of requiring that the index be rebuilt for each different distance metric, which may be nontrivial. On the other hand, this is not the case for the multidimensional indexing methods which have the advantage of supporting arbitrary distance metrics (however, this comparison is not always entirely fair, since the assumption, when using distance-based indexing, is that often we do not have any feature values as for example in DNA sequences).

In this article, we survey a number of such *distance-based indexing* methods, and show how they can be used to perform similarity search. We do not treat multidimensional indexing methods (see Böhm et al. [2001] for a recent survey). Some of the earliest distance-based indexing methods are due to Burkhard and Keller [1973], but most of the work in this area has taken place within roughly the past decade. Typical of distance-based indexing structures are *metric trees* [Uhlmann 1991a, 1991b], which are binary trees that result in recursively partitioning a data set into two subsets at each node. Uhlmann [1991b] 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* [Yianilos 1993], that is, into the subset that is inside and the subset that is outside a ball around the object (e.g., Figure 1(a)). 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, that is, 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., Figure 1(b)).

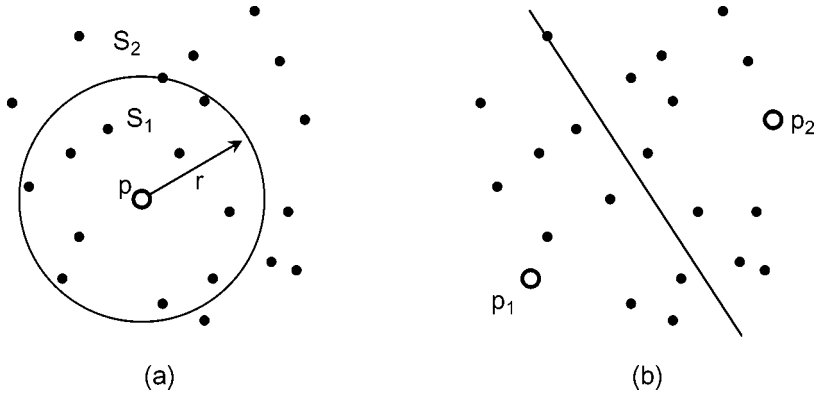


Fig. 1. Possible top-level partitionings of a set of objects (depicted as two-dimensional points) in a metric tree using (a) ball partitioning and (b) generalized hyperplane partitioning.

In this article, 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. [2001b]. 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, for example, 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 $p' \in S$.

While most distance-based indexing structures are variations on and/or extensions of metric trees, there are also other approaches. Several methods based on distance matrices have been designed [Micó et al. 1994; Vidal Ruiz 1986; Wang and Shasha 1990]. 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 [Micó et al. 1996]. The *sa-tree* [Navarro 2002] 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.

The remainder of the article is organized as follows. In Section 2, we present a general framework for performing similarity search. In Section 3, we describe three common types of queries and present query algorithms for each one, expressed within this general framework. In Section 4, we discuss properties of distance metrics that are exploited by the various indexes to provide lower bounds on the distances from the query object(s) of objects in one or more subsets of S thereby enabling the search to be pruned. In Section 5, we describe the *vp-tree* [Yianilos 1993] and other variants of metric trees that

employ ball partitioning. In Section 6, we present the *gh-tree* [Uhlmann 1991b] and other variants of metric trees that employ generalized hyperplane partitioning. In Section 7, we describe the *M-tree* [Ciaccia et al. 1997], a dynamic and balanced metric tree variant, suitable for disk-based implementation. In Section 8, we introduce the sa-tree [Navarro 2002]. In Section 9, we describe AESA [Vidal Ruiz 1986] and LAESA [Micó et al. 1994], methods that rely on distance matrices. In Section 10 we review the different indexing methods that we presented and provide an alternative method of characterizing them which shows the close relationship between some of them and the embedding methods that were briefly discussed above. Finally, Section 11 contains concluding remarks.

Note that this article differs from the excellent recent survey by Chávez et al. [2001b] on searching in metric spaces in that we focus more on practical aspects of performing similarity search with distance-based indexing structures, while Chávez et al. [2001b] focus on the theoretical aspects of the various indexing structures. Generally speaking, compared to Chávez et al. [2001b], our article provides a more detailed and, perhaps, more intuitive description of the methods that we cover. Note that Chávez et al. [2001b] also present a unifying framework, but the aim of their framework is to classify the different methods, in order to compare and contrast them, while our focus is more on how they can be used for similarity searching using a unified search hierarchy.

Our article has two goals. The first goal is to serve as a starting point in exploring this topic for newcomers to similarity searching in metric spaces, as well as to be a reference for practitioners in the field. We describe a number of indexing structure in considerable depth, and provide pointers for further reading on those and related structures. Along the way, our treatment points out aspects that different structures have in common, and where they diverge. Although we often describe construction algorithms for the different indexing structures, our emphasis is on search algorithms. The second goal is to show how the general framework for performing similarity search can be adapted for the different distance-based indexing methods. Many of the resulting algorithms are new. The similarities in the different adaptations help in providing a unified view of the different structures with regard to similarity searching.

2. GENERAL SEARCH FRAMEWORK

In this section, we present a general framework that is essentially an abstraction of the common elements of virtually all methods for organizing sets of metric space data and performing similarity searching on it. Loosely speaking, it applies to methods that are based on some form of the philosophy of "divide and conquer." Hence, the framework is hierarchical in nature and defines the abstract concept of *search hierarchy*, on which general search algorithms can be applied (see Section 3). In later sections, we present concrete instances of search hierarchies, which then yield concrete similarity search algorithms from the general ones.

2.1 Search Hierarchy

The common elements of most similarity search methods in metric spaces are as follows. From a database S of N objects, we wish to find an object, or objects, that is similar to a query object q. More precisely, S and q are drawn from a metric space $(\mathbb{U}, d)$, where $\mathbb{U}$ is the "universe" of objects and d is a distance metric that indicates dissimilarity; that is, greater distances mean less similarity. The search is performed with the aid of a data structure T that organizes the set S or provides information about it, and possibly some auxiliary structures that only exist during the search. Given the entire problem P of searching in S, one or more subproblems $P_1, P_2, \ldots$ are identified, each of which applies to an associated subset of S (typically, the subsets are disjoint). The subproblems may have some relative "priority" and some of them may be immediately known to yield no result. The subproblems, in turn, yield subproblems of their own, and so on, until we have subproblems that are sufficiently elementary to be "solved" directly.

The concept of search hierarchy is an abstraction of the essence of methods that fit the above general description. The nodes in the hierarchy are referred to as *elements* (to avoid confusion with nodes in data structures), and each element represents a subproblem of the search; in fact, we often refer to the elements as "search problems." Thus, the root of the hierarchy represents P, as defined above, while its children represent the subproblems of P, and the leaves of the hierarchy represent the elementary subproblems. Equivalently, each element e represents a subset of S, denoted $s[e]$, with the leaves denoting single objects in S (or, possibly, a "small" set of objects). The elements are classified into different types, $t = 0, \ldots, t_{\max}$, based on the nature of the associated subproblem, with type 0 denoting the objects in S; the different element types typically have different kinds of associated information. (For example, for the search hierarchy given in Section 2.2 for spatial data, there are three types of elements, denoting objects, bounding rectangles, and nodes.) We denote the type of an element e with $t[e]$. A search hierarchy on a set of 14 objects, A_0 though N_0, with three types of elements, is shown in Figure 2. Elements of type 1 and 2 are depicted with hollow and shaded circles, respectively, while elements of type 0 are depicted with hollow boxes. As depicted in the figure, an element e can give rise to one or more "child" elements of types 0 through $t_{\max}$; child elements are often all of the same type, but this is not a requirement. In this presentation, we assume that the child elements of e represent disjoint subsets of e's subset, or, equivalently, that each element has only one "parent" in the search hierarchy, and, in particular, that each object is represented only once in the hierarchy. However, many of the search algorithms presented in Section 3 can be adapted to handle multiple parents and duplicate object instances.

Elements of each type t have an associated distance function $d_t(q, e)$ for measuring the distances from a query object q to elements e of that type. The distance $d_t(q, e)$ "summarizes" in some manner the distances of objects in $s[e]$, and reflects knowledge gained in solving the subproblem associated with the parent p of e in the hierarchy. Thus, the computation of $d_t(q, e)$ must be based solely on the specific information attached to p or on information computed

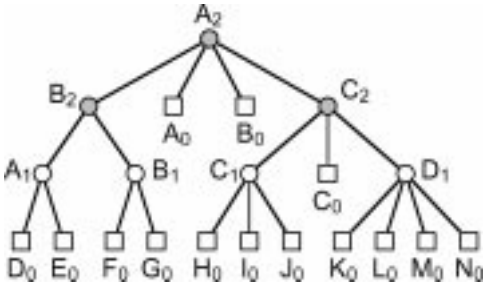


Fig. 2. A sample search hierarchy for objects A_0 through N_0. Elements of type 1 and 2 are depicted with hollow and shaded circles, respectively, while elements of type 0 are depicted with hollow boxes.

earlier in the search process. Furthermore, if e is not of type 0, this computation should be substantially less expensive than the total cost of computing $d(q, o)$ for all objects $o \in s[e]$, or else it would not be worthwhile to extract e as a subproblem of p. Observe that the distances of the children of p can be used to indicate the priority of the associated subproblems and/or for immediate pruning. Typically, the distance functions are defined such that they lower-bound the distances of the objects represented by the elements: $d_t(q, e) \leq d(q, o)$ for any object $o \in s[e]$. Most of the algorithms presented in Section 3 rely on this property for their correctness, but relaxing it may yield better performance at the cost of reduced accuracy. Some types of search (such as farthest neighbors) make use of another set of distance functions, $\hat{d}_t(q, e)$, that bound from above the distances from q of the objects in a subtree: $\hat{d}_t(q, e) \geq d(q, o)$ for any object $o \in s[e]$.

Observe that when the data structure T is itself of a hierarchical nature, the search hierarchy usually arises naturally from the hierarchical structure of T. For a given similarity search method, nevertheless, it is often possible to define more than one search hierarchy. In this case, the different choices of hierarchies can be thought of as reflecting different search policies. In some cases, however, search algorithms exploit specific features of a data structure that cannot easily be abstracted into a search hierarchy; for example, see Section 6.2.

2.2 Example 1: Spatial Data

The concept of a search hierarchy, as defined above, also applies to search in geometric space, and to data structures and algorithms defined on spatial data. In fact, many of the algorithms presented in Section 3 were originally proposed for spatial indexing structures. Hence, to obtain a better grasp of the notion of search hierarchy and how it relates to search, it is a useful exercise to consider an example in geometric space. Figure 3 depicts seven two-dimensional spatial objects and a hierarchical partitioning of them as produced by a hypothetical spatial indexing technique (e.g., the R-tree [Guttman 1984]), and a range query specified by a query point q and radius r. Each of the seven objects o_1 through o_7 has a bounding rectangle, denoted b_1 through b_7, respectively, and the space partitioning on them consists of the leaf node regions n_1, n_2, and n_3, and that of their parent node n. The range query gives rise to the shaded query region

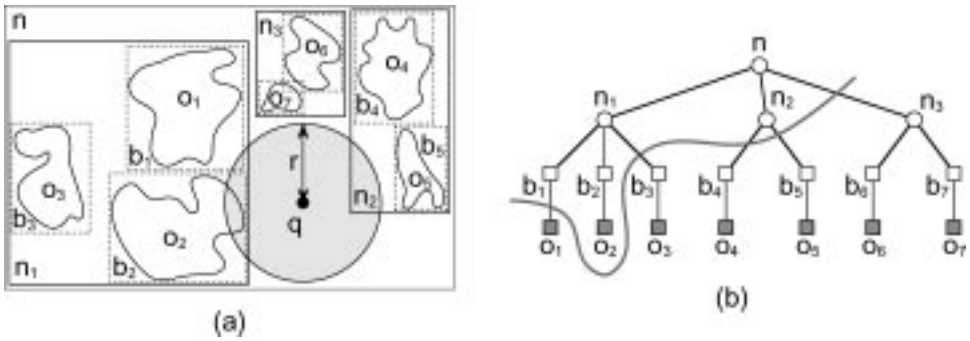


Fig. 3. A small collection of two-dimensional spatial objects, represented by a space hierarchy, and a sample range query on the collection: (a) a geometric view of the objects, the space partitioning on them, and the query region; and (b) the search hierarchy that naturally arises from the space partitioning in (a), and the boundary below which the range search need not pass.

in Figure 3(a), which can be seen to intersect n, n_1, n_2, b_1, b_2, and o_2. Thus, to answer the query, we must at a minimum explore the portion of the search hierarchy in Figure 3(b) above the boundary cutting through it, which contains the elements whose regions intersect the query region.

Using the terminology introduced in Section 2.1, the search hierarchy shown in Figure 3(b) consists of three types of elements: the objects o_i are elements of type 0, the bounding rectangles b_i are elements of type 1, while the nodes n, n_1, n_2, and n_3 are of type 2. Assuming that we use the Euclidean distance metric to measure distances from the query object q, as implied by the query region in Figure 3(b), it naturally follows that we should use Euclidean distance as a basis for the distance functions for all three element types. In other words, if $d_E(p_1, p_2)$ denotes the Euclidean distance between points p_1 and p_2, we define $d_t(q, e) = \min_{p \in R[e]} d_E(q, p)$ for $t = 1, 2$, where $R[e]$ denotes the spatial region associated with element e of type t. This definition is in accordance with the geometric depiction of Figure 3(a), as the resulting distances are less than r for all the elements corresponding to regions intersecting the query region. In particular, the distance between q and an element e_i is equal to the distance from q to the closest point on the boundary of the spatial region for e_i, or zero if q is inside the region (which is true for n). Furthermore, observe that the fact that each spatial object o is enclosed by the regions of all ancestor elements e in the hierarchy directly implies that this definition naturally satisfies the lower-bounding criteria $d_t(q, e) \leq d_0(q, o)$.

In performing the range search on the hierarchy in Figure 3(b), we need only explore the portion of the hierarchy above the boundary line, as mentioned above. We say that the elements above the line must be *visited* by the search, while the portion of the hierarchy below the line can be *pruned*. For any visited element e of the hierarchy, we must compute the distance between q and all child elements of e. Therefore, in addition to the visited elements, the search must compute the distance from q of elements that are immediately below the boundary (i.e., that are incident to edges that intersect the boundary), namely o_1, b_3, b_4, b_5, and n_3 in the figure. Having done that, however, the search can

prune these elements and all their descendants without further distance computations, and, in particular, we need not compute the actual distances from q to the objects o_3 through o_7.

By analogy with the geometric example above, we obtain an alternative view of the abstract concept of search hierarchies that can be helpful to gain an intuitive understanding. In particular, each non-object element e can be thought of as having an associated "region" $R[e]$ that "overlaps" all objects $o \in s[e]$. This captures the upper- and lower-bounding criteria on the distance functions. In many cases, search hierarchies satisfy the stronger criteria that $R[e']$ is completely contained in $R[e]$ for the child elements e' of e, as is the case for our geometric example. However, this is not a requirement, and does not hold for some of the search hierarchies that we present below (e.g., the one for the vp-tree in Section 5.1).

2.3 Example 2: Feature Vectors

As mentioned in Section 1, it is quite common in similarity search applications to use (feature) vectors to describe the data objects, where the vectors may be obtained using (domain-specific) feature extraction, dimension-reduction, or general embedding methods. These feature vectors are indexed using multidimensional indexing structures and as they apply a form of spatial indexing, the discussion in Section 2.2 is applicable to them as well, and a search hierarchy can be defined for performing similarity (or proximity) search on the set of vectors. When the feature vectors are obtained using some mapping F as in the embedding methods discussed in Section 1, then the complete execution of a similarity search query requires the added step of refining the query result based on the original object data. In other words, as we pointed out in Section 1, for the query object $q \in \mathbb{U}$ and data set S, the initial, "filter," step of the query processing involves computing the vector $F(q)$ and performing the query on the set of vectors $F(S)$ using a distance function δ with the aid of the multidimensional index. In the "refinement" step (which may actually occur concurrently with the filter step), the actual similarity to q is evaluated for candidate objects from the filter stage, and irrelevant objects discarded.

Such "multistep" query processing can actually be thought of as being applied on a particular form of search hierarchy, formed by extending the search hierarchy used on the set of vectors $F(S)$, for example, based on a multidimensional indexing structure. In particular, in the extended search hierarchy, each element for a vector $F(o)$ has the corresponding object $o \in S$ as its child. Thus, the situation is very similar to the example in Figure 3, with vectors replacing bounding rectangles.

The lower-bounding property $d_t(q, e) \leq d(q, o)$ implies that the mapping F and the distance function δ must satisfy $\delta(F(q), F(o)) \leq d(q, o)$ for any $q \in \mathbb{U}$ and $o \in S$. We recall from Section 1 that when this holds, F is said to be *contractive*, and it ensures that we do not suffer *false dismissals* in the filter step. This is demonstrated in Figure 4, which shows both objects and the vectors resulting from mapping the objects; the figure can be thought to represent arbitrary domains of objects and their mapped versions, with the understanding that the

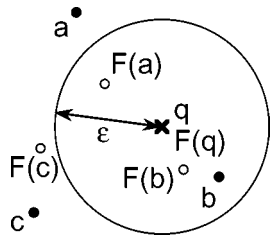


Fig. 4. Range query, with query object q and query radius ϵ, on three data objects a, b, and c, represented as two-dimensional points. The mapped versions of the objects are also depicted as two-dimensional points in the same space. Only b is in the range, but the distance d(q, a) must also be computed, since F(a) is in the range.

two-dimensional distances in the figure represent relative distances between q and the data objects, on the one hand, and between F(q) and the corresponding vectors, on the other. Observe that c can be pruned from the search without computing d(q, c), by virtue of F(c) being outside the range. If the mapping were not contractive, then c could be *inside* the range even if F(c) is *outside* it, in which case a false dismissal would have occurred in the example.

3. QUERIES AND SEARCH ALGORITHMS

In this section, we define three basic types of queries that are commonly used for similarity search and outline a few useful variations of these queries. We also present search algorithms for each query, defined on the basis of the framework presented in Section 2.

(1) *Range*: Given a query object q and $\epsilon \geq 0$, find all $o \in S$ such that $d(q, o) \leq \epsilon$.
(2) *Nearest neighbor*: Given a query object q and $k > 0$, find the k objects in S with smallest distance from q.
(3) *Ranking*: Given a query object q, report the objects in S in order of distance from q, subject to some stopping condition.

Each successive query type is more difficult to handle than the previous one, in the sense that given values for an unknown parameter, later queries can be computed with the one before. Thus, for the nearest neighbor query, if we knew in advance the distance D_k of the kth nearest neighbor, we could answer it with a range query using $\epsilon = D_k$. Similarly, for the ranking query, if we knew the number k_r of objects in the result of a ranking query that are implied by the stopping condition, the result can be provided by applying a nearest neighbor query using $k = k_r$ (provided the result is ordered by distance). Indeed, it can be useful to combine the criteria of the queries, adding a maximum distance criteria (cf. range) to the nearest neighbor query, and including maximum distance and/or maximum cardinality (cf. number of neighbors) in the stopping condition of the ranking query.

The stopping condition for the ranking query can be based on arbitrary criteria. Clearly, these criteria can be based on properties that are inherent in the search hierarchy, such as maximum distance or maximum cardinality of the result set, as mentioned above. More interestingly, however, the criteria may be

based on external properties. Typically, these external properties arise from the fact that the distance function d is based on only a partial aspect of the objects in S. In database terminology, what this implies is that the objects in S have multiple attributes, and d is based on only one (or a few) of these attributes. For example, in a geometric setting as noted in Section 2.2, S might represent a "city" relation, with attributes for name, population, location, etc. Thus, we might answer the query "find the city of population at least one million closest to the Grand Canyon" with a ranking query on the location with a stopping condition on the population.

The most useful algorithms for the ranking query are ones that compute the result progressively, such that results are reported as early as possible. This allows making use of pipelined query execution in complex queries involving ranking queries as subqueries [Carey and Kossmann 1997; Fagin et al. 2001].

Although we restrict our focus to "elementary" queries that involve a single set of objects, there are important types of queries that involve two or more sets of objects—that is, what is termed *join* queries in database applications. For example, in a geometric setting, the query "find stores and warehouses within 20 miles of each other" is a join with a "within" criteria (analogous to a range query), while the query "find the hotel closest to a French restaurant" is a join with a "closest" criteria (analogous to a nearest neighbor query). A number of join algorithms have been proposed for particular indexing structures, for example, Brinkhoff et al. [1993], Hjaltason and Samet [1998], Shafer and Agrawal [1997] and Shin et al. [2000]. Many of these algorithms can be generalized to apply to search hierarchies.

In the rest of this section, we describe algorithms for each type of query. Section 3.1 reviews a depth-first algorithm for the range query. Section 3.2 presents two algorithms for the nearest neighbor query. Section 3.3 gives an incremental algorithm for the ranking query. Finally, Section 3.4 discusses variants on the different queries and algorithms.

3.1 Range Query

The range query for query object q and radius ϵ on a search hierarchy T can be handled with a straight-forward depth-first traversal of T. This is illustrated in Figure 5, where the depth-first traversal is performed by the RangeTraversal procedure. In the algorithm, the *RangeSet* variable refers to the set of result objects that is accumulated in the course of the traversal. The Range procedure initializes this set to be empty and invokes RangeTraversal on the root of the search hierarchy. We make the simplifying assumption that there are at least two objects in the data set S, which implies that the root of the search hierarchy is not an object.

The **for**-loop in RangeTraversal iterates over the children of element e in the search hierarchy. The distance of a child element e' is tested in line 2 and the rest of the loop is skipped if e' is too far from q. This looks perhaps somewhat awkward, but this makes it easier to see the depth-first nearest neighbor

```
RANGE(q, ε, T)
 1  e ← root of the search hierarchy T
 2  RangeSet ← NEWSET() /* a set for accumulating result */
 3  RANGETRAVERSAL(q, ε, RangeSet, e)
 4  return RangeSet

RANGETRAVERSAL(q, ε, RangeSet, e)
 1  for each child element e' of e do
 2    if d_t[e'](q, e') > ε then
 3      Continue loop with next child element of e (if any)
 4    if t[e'] = 0 then /* e' is an object */
 5      INSERT(RangeSet, e')
 6    else
 7      RANGETRAVERSAL(q, ε, RangeSet, e')
```

Fig. 5. Range search algorithm for a search hierarchy T, query object q, and radius ϵ.

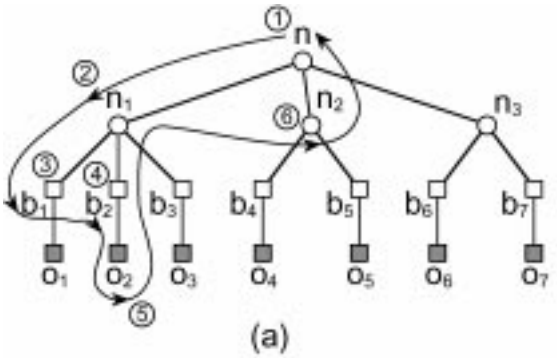
algorithm in Section 3.2.1 as an extension of this algorithm. Also, note that we assume that the element e may have as children elements of type 0 (i.e., objects) as well as of other types, so we must check the type of e' inside the **for**-loop (i.e., in line 4). If objects are the sole kind of children of elements of type 1, say, then it would be more natural to use two separate **for**-loops depending on the type of e. In other words, the **for**-loop for type 1 would contain the insertion statement of line 5, while for other types it would perform the recursive call of line 7.

The portion of the search hierarchy that must be visited by the range query illustrated in Figure 3(a) is shown in Figure 3(b), as noted above. The range search algorithm given in Figure 5 visits exactly the necessary portion shown in Figure 3. A further illustration of the traversal performed by the algorithm is shown in Figure 6(a) where the arrows and the circled numbers indicate the order in which the traversal proceeds. We also show in Figure 6(b) the distances of the various search hierarchy elements from the query point q where, of course, the distance of the six elements visited by the algorithm is smaller than the search radius $r = 73$.

3.2 Nearest Neighbor Query

As defined above, a nearest neighbor query involves finding the k closest objects in S to a query object q. There are numerous ways of performing the search for such queries, primarily depending on how the search hierarchy is traversed. We present two algorithms that use two different traversal orders. The first algorithm makes use of depth-first traversal and is a straightforward extension of the range search algorithm. The second algorithm, on the other hand, uses "best-first" traversal, which is based on the distances and, in a sense, breaks free of the shackles of the search hierarchy.

3.2.1 *Depth-First Algorithm.* The key idea in extending the range search algorithm of Figure 5 for k nearest neighbor search is to make use of objects that are found during the traversal to bound the search, that is, to serve as



Item	Dist.
n	0
n_2	44
n_1	51
b_2	51
o_2	57
b_1	63
n_3	80
b_7	80
o_7	82
b_5	85
b_4	88
o_5	91
o_1	98
b_6	103
o_6	104
o_4	112
b_3	203
o_3	205

r=73

(b)

Fig. 6. (a) A depiction of the traversal for the range query in Figure 3(a), where the arrows indicate the direction of the traversal and the circled numbers denote order in which the elements are visited. (b) Some of the search hierarchy elements closest to query object q along with their distances from q in increasing order. Using the same distance unit, the value of r in Figure 3(a) is 73.

ϵ in the range search algorithm. In particular, the value of ϵ is ∞ until at least k objects have been seen, and from then on is set to the kth smallest distance seen so far. Clearly, the value of ϵ converges quicker to the distance of the kth nearest neighbor of q if we see objects that are close to q as early as possible. A heuristic that aims at this goal is such that at each nonleaf element e that is visited, we visit the children of e in order of distance. The term "branch and bound" is sometimes used to characterize algorithms of this form: After "branching" to the child element e' with the smallest value of $d(q, e')$, we have hopefully reduced ϵ, thus helping to "bound" the search.

The nearest neighbor algorithm in Figure 7 makes use of the above observations. In the algorithm, the list *NearestList* is used to store the k candidate nearest neighbors (i.e., the k objects seen so far that are closest to q), similar to the use of *RangeSet* in the range search algorithm. The expression MaxDist(*NearestList*) replaces the use of ϵ in the range search algorithm, and has the value of the greatest distance among the objects in *NearestList*, or ∞ if there are still fewer than k objects in the list. The call to SortBranchList in line 2 sorts the children of e in order of distance, which determines the order in which the following **for**-loop iterates over the children. When the distance from q of a child element is found to be greater than MaxDist(*NearestList*) (line 4), the ordering ensures that all subsequent child elements also have distances that are too large, so the **for**-loop can be terminated.

The algorithm in Figure 7 is essentially a generalized version of the early nearest neighbor algorithm presented Fukunaga and Narendra [1975], and later extended [Kamgar-Parsi and Kanal 1985; Larsen and Kanal 1986], and

```
DFNEAREST(q, k, T)
 1 e ← root of the search hierarchy T
 2 NearestList ← NEWLIST(k) }/* a list for accumulating result */}
 3 NEARESTTRAVERSAL(q, NearestList, e)
 4 return NearestList

NEARESTTRAVERSAL(q, NearestList, e)
 1 ActiveBranchList ← child elements of e
 2 SORTBRANCHLIST(ActiveBranchList, q)
 3 for each element e′ in ActiveBranchList do
 4    if d_t[e′](q, e′) > MAXDIST(NearestList) then
 5       Exit loop
 6    if t[e′] = 0 then }/* e′ is an object */}
 7       INSERT(NearestList, e′, d_0(q, e′))
 8       /* element with largest distance is removed */
 9       /* if already k objects in list */
10    else
11       NEARESTTRAVERSAL(q, NearestList, e′)
```

Fig. 7. A depth-first k-nearest neighbor algorithm on a search hierarchy T given a query object q.

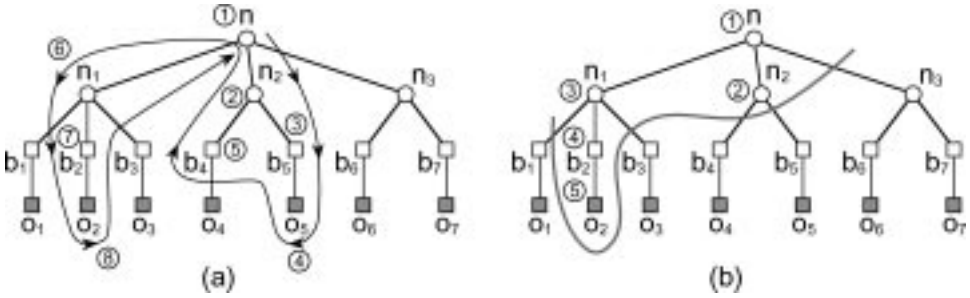


Fig. 8. (a) A depiction of the traversal for the depth-first nearest neighbor algorithm for the data and query objects in Figure 3(a) for k = 1, where the arrows indicate the direction of the traversal and the circled numbers denote order in which the elements are visited (i.e., the leaf and nonleaf elements that have not been pruned). (b) A depiction of the traversal for the best-first nearest neighbor algorithm for the same argument values, with the curved line delimiting the portion of the search hierarchy that is visited.

applied to other structures (e.g., Chiueh [1994], Navarro [2002], Roussopoulos et al. [1995] and Yianilos [1993]). The main difference in our presentation is that we do not explicitly prune (i.e., remove) members of the *ActiveBranchList* after each recursive invocation (line 11). Instead, the termination of the **for**-loop in line 5 implicitly prunes the remaining members of the list.

Figure 8(a) shows a trace of the algorithm with $k = 1$ on the set of objects (and associated bounding rectangles and nodes) and query object q given in Figure 3(a). Initially, *NearestList* is empty, which implies that MAXDIST(*NearestList*) in line 4 evaluates to ∞. The root node n is naturally the first to be visited during the traversal. The *ActiveBranchList* computed for n (in line 2) is $\{(n_2, 44), (n_1, 51), (n_3, 80)\}$; the distance values come from Figure 6(b). The second node to be visited is then n_2, whose *ActiveBranchList* is $\{(b_5, 85), (b_4, 88)\}$. The bounding rectangle b_5 is visited next, followed by the associated object o_5 at a distance of 91, which becomes the first candidate nearest

neighbor to be inserted into *NearestList* (line 7). Backtracking to n_2, we must now visit the next element on its *ActiveBranchList*, namely b_4 since its distance value is less than 91 (so the test in line 4 is false), but o_4 is pruned from the search (via line 5) since $d(q, o_4) = 112 > 91$. Thus, the algorithm backtracks to n, where the next element on the *ActiveBranchList* is n_1, which is promptly visited since its associated distance 51 is less than 91. The *ActiveBranchList* for n_1 is $\{(b_2, 51), (b_1, 63), (b_3, 203)\}$, and since $d(q, b_2) = 51 < 91$, we next visit b_2, followed by the associated object o_2, since its distance from q of 57 is less than 91. At this point, o_2 is inserted into *NearestList* (line 7), causing o_4 to be ejected, and o_2 thus becomes the new candidate nearest neighbor. Finally, since the distance values of b_1 (on the *ActiveBranchList* of n_1) and n_3 (on the *ActiveBranchList* of n) are both greater than 57, the traversal backtracks beyond the root and the algorithm terminates, having determined that o_2 is the nearest neighbor of q.

3.2.2 *Best-First Algorithm.* The nearest neighbor algorithm presented in Section 3.2.1 is as good as possible for a depth-first algorithm given only the lower-bound distances present in the search hierarchy (i.e., without making use of some aspect of the search structure that cannot be abstracted into a search hierarchy). However, it turns out that we can do better with different traversal strategies. To see why, observe that once we visit a child e' of an element e, we are committed to traverse the entire subtree rooted at e' (subject to the pruning condition) before another child of e can be visited. For example, in the traversal illustrated in Figure 8(a), we must visit b_5 and o_5 after the traversal reaches n_2, even though n_1 is closer to q than b_5. The above property is inherent in the fact that the algorithm of Figure 7 maintains a separate *ActiveBranchList* for each element on the path from the root of the search hierarchy down to the current element. Thus, it may seem that we might improve on the depth-first strategy by somehow combining *ActiveBranchList*'s for different elements.

The *best-first* traversal strategy is indeed driven by what is in effect a global, combined *ActiveBranchList* of all elements that have been visited. In the nearest neighbor algorithm of Figure 9, this global list is maintained with a priority queue *Queue*, with distances serving as keys. Initially, the root of the search hierarchy is inserted into the queue. Then, at each step of the algorithm, the element with the smallest distance is removed from the queue (i.e., it is "visited"), and its child elements either inserted into *Queue* or, for objects, into *NearestList*. The *NearestList* variable and MaxDist(*NearestList*) have the same meaning as in the depth-first algorithm, while MinDist(*Queue*) denotes the smallest distance among the elements in *Queue*. Observe that MinDist(*Queue*) and MaxDist(*NearestList*) converge toward each other (from 0, being increased by line 7, and from ∞, being decreased by line 11, respectively), and that when the **while**-loop terminates, the value of MaxDist(*NearestList*) has reached the distance from q of its kth nearest neighbor. Also, note that, as a variation on this algorithm, we could remove elements from *Queue* after the execution of the **for**-loop in case MaxDist(*NearestList*) has decreased (due to one or more insertions in line 11), which renders unnecessary the second part of the condition

```
BFNearest(q, k, T)
 1  Queue ← NewPriorityQueue()
 2  NearestList ← NewList(k)
 3  e ← root of the search hierarchy T
 4  Enqueue(Queue, e, 0)
 5  while not IsEmpty(Queue) and
 6        MinDist(Queue) < MaxDist(NearestList) do
 7     e ← Dequeue(Queue)
 8     for each child element e′ of e do
 9        if d_t[e′](q, e′) ≤ MaxDist(NearestList) then
10           if t[e′] = 0 then /* e′ is an object */
11              Insert(NearestList, e′, d_t[e′](q, e′))
12              /* element with largest distance is removed */
13              /* if already k objects in list */
14           else
15              Enqueue(Queue, e′, d_t[e′](q, e′))
16  return NearestList
```

Fig. 9. A best-first k-nearest neighbor algorithm on a search hierarchy T given a query object q.

in line 6. However, in this case, the priority queue implementation must support efficient ejection of elements having keys greater than some value (i.e., MaxDist(*NearestList*)).

One way to obtain an intuition about the best-first nearest neighbor algorithm is to consider the geometric case. If q is a two-dimensional point, as in Figure 3, the search effectively proceeds by first drilling down the search hierarchy and then expanding a circular query region with q as its center. Each time that the query region hits a nonleaf element e, we visit e, and the search terminates once the query region intersects at least k objects. The order in which search hierarchy elements would be visited for the above example is depicted in Figure 8b. Initially, *NearestList* is empty and $Queue = \{(n, 0)\}$. Thus, we "drill down" to n, causing its children to be inserted into the priority queue, which results in $Queue = \{(n_2, 44), (n_1, 51), (n_3, 80)\}$ (all enqueued in line 15). At this point, the circular query is in effect expanded until its radius becomes equal to 44, the distance of n_2 from q. Visiting n_2 causes b_4 and b_5 to be inserted into the priority queue, resulting in $Queue = \{(n_1, 51), (n_3, 80), (b_5, 85), (b_4, 88)\}$. Next, n_1 is visited and b_1 through b_3 enqueued, yielding $Queue = \{(b_2, 51), (b_1, 63), (n_3, 80), (b_5, 85), (b_4, 88), (b_3, 203)\}$. Visiting b_2 leads to the insertion of o_2 with distance value of 57 into *NearestList* (in line 11), which has hitherto been empty. Since we now have $63 = \text{MinDist}(Queue) \not< \text{MaxDist}(NearestList) = 57$ (i.e., the second part of the condition in line 6 now evaluates to false), and the traversal is terminated with o_2 as the nearest neighbor.

The best-first nearest neighbor algorithm can be thought of as a special case of the classic A*-algorithm (e.g., see Russel and Norvig [1994]). Nevertheless, it was not popularized until relatively recently (e.g., see Arya et al. [1994]). One reason is perhaps that the depth-first algorithm performs quite adequately in many cases. Another reason may be that priority queues are somewhat expensive to maintain. Thus, when the cost of priority queue operations is on a similar order as the cost of visiting elements (e.g., computing distances or performing

```
INCRANKING(q, T)
 1  Queue ← NEWPRIORITYQUEUE()
 2  e ← root of the search hierarchy T
 3  ENQUEUE(Queue, e, 0)
 4  while not ISEMPTY(Queue) do
 5    e ← DEQUEUE(Queue)
 6    if t[e] = 0 then /* e is an object */
 7      Report e as the next result object
 8    else
 9      for each child element e' of e do
10        ENQUEUE(Queue, e', d_t[e'](q, e'))
```

Fig. 10. Incremental ranking algorithm on a search hierarchy T given a query object q.

disk I/O), the depth-first algorithm may be preferred over the best-first algorithm. However, when distances are expensive to compute and/or the search hierarchy represents a disk-based structure, the improved pruning exhibited by the best-first algorithm comfortably outweighs the cost of maintaining the priority queue (e.g., see Hjaltason and Samet [1999]).

3.3 Ranking Query

In computing the ranking query, we must produce a list of the objects in S in order of distance from the query object q. However, it is rare that we wish to rank all the objects in S, and the stopping condition typically causes only a fraction of the objects in S to be ranked. Furthermore, as mentioned above, it is most useful in actual systems if the ranking query is computed in a progressive fashion, such that the result is produced incrementally. In fact, it turns out that the best-first nearest neighbor algorithm of Figure 9 can be extended to do just that. The key insight is that at any state of the algorithm, the distances of objects that are later added to *NearestList* can be no smaller than MINDIST(*Queue*) (i.e., due to the lower-bounding property of distances), and the objects in *NearestList* with distances that are less than or equal to MINDIST(*Queue*) constitute the first objects in the result of a ranking query. Thus, we obtain an incremental ranking algorithm by simply outputting objects in *NearestList* once their distance is no larger than MINDIST(*Queue*).

The algorithm presented in Figure 10 is essentially based on this principle. In effect, it merges the functionality of *NearestList* with that of the priority queue, thereby allowing an unlimited result size. Using a single priority queue makes for a simple algorithm: when an object reaches the front of the queue, it gets output as the next object in ranking order. Alternatively, it is possible to use a separate priority queue for the objects, in which case the algorithm would in each iteration dequeue from the priority queue whose front element has the smaller distance. In either case, we can gain an intuition about the progress of the ranking algorithm by considering the geometric case. For a two-dimensional query point q, as in Figure 3, the search can be viewed as expanding a circular query region with q as its center, similar as for the best-first nearest neighbor algorithm. An element e reaching the front of the priority queue is then essentially analogous to the event that the expanding query region hits e.

This algorithm was introduced independently by Henrich [1994] and Hjaltason and Samet [1995, 1999], in the context of spatial databases. The main difference in their algorithms was that Henrich used two priority queues.

The trace of the ranking query on the sample objects and query point in Figure 3 is essentially the same as described for the best-first nearest neighbor algorithm above. The difference is that, here, when b_2 is visited, o_2 is inserted into the priority queue (line 10) rather than *NearestList*, which results in $Queue = \{(o_2, 57), (b_1, 63), (n_3, 80), (b_5, 85), (b_4, 88), (b_3, 203)\}$. Thus, in the next iteration of the **while**-loop, o_2 is reported as the first nearest neighbor (line 7). At this point, if no more neighbors are needed, the algorithm is terminated; otherwise, the traversal continues, visiting b_1 next, and eventually determining that o_7 is the second nearest neighbor (but only after also visiting n_3 and b_7; see Figure 6(b) for the associated distance values).

The fact that the algorithm in Figure 10 is able to rank all N objects in S opens it up to the liability that the priority queue may become very large, and in, the worst case, on the order of N. For example, this would be true if all the objects in S are at approximately the same distance away from q (e.g., in the two-dimensional geometric case, the data objects may be arranged on a circle around the query point). Of course, likelihood of the occurrence of this event is very rare from a probabilistic standpoint (i.e., the probability of the events that all the objects being along a circle and that the query point is at their center), especially for low-dimensional data; nevertheless, we still cannot rule out the possibility of the size of the priority queue being on the order of N. To avoid this situation, we can impose a limit on the size of the priority queue, and discard the elements with the largest distances when it becomes too large, in a similar way that *NearestList* is handled in the nearest neighbor algorithms. When this occurs, we effectively put an upper bound on the distance to the farthest object that can be produced by the ranking. In particular, if D is the smallest distance of an element that has been discarded, the algorithm will only produce a ranking for objects $o \in S$ with $d(q, o) \leq D$. If that does not produce enough objects to satisfy the stopping condition of the ranking query, the algorithm must be restarted, but this time with D being a minimum distance of produced objects (see below).

3.4 Variants

Several useful variants can be defined of the queries and algorithms presented above. For example, we can impose a maximum distance $D_{\max}$ to the nearest neighbor and ranking queries, which has the same meaning as the radius ϵ in the range query. For the nearest neighbor algorithms, this simply requires making the original value of MaxDist(*NearestList*) be $D_{\max}$ instead of ∞, while it requires additional distance tests in the ranking algorithm. A minimum distance bound $D_{\min}$ can also be exploited in all the algorithms, but doing so requires the additional upper-bounding distance functions $\hat{d}_{t[e]}(q, e)$. In particular, in each case, we discard elements that satisfy $\hat{d}_{t[e]}(q, e) < D_{\min}$.

Another useful variant of the nearest neighbor query is the farthest neighbor query, and, similarly, for the ranking query the reverse ranking query. This

simply requires replacing $d_{t[e]}(q, e)$ for ordering elements for processing with $-\hat{d}_t(q, e_t)$ (i.e., ordering by decreasing values of $\hat{d}_t(q, e_t)$). Unfortunately, this is likely to be inefficient and requires visiting a large portion of the search hierarchy before the farthest object from q is identified, since many objects tend to be at a similar distance from q as the farthest one, relatively speaking.

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 algorithms can be easily extended. For approximate nearest neighbor search [Arya et al. 1998], a proposed criterion is that the distance between q and the resulting candidate nearest neighbor o' is within a factor of $1+\epsilon$ of the distance to the actual nearest neighbor o, that is, $d(q, o') \leq (1+\epsilon)d(q, o)$. All the algorithms can be made approximate in this sense by "inflating" the distances of non-object elements by a factor of $1+\epsilon$. For the ranking algorithm, the implication of this modification is that the objects are not necessarily produced in distance order. Instead, for any successive objects o_1 and o_2 that are reported, we have $d(q, o_1) \leq (1+\epsilon)d(q, o_2)$. Other strategies of making search algorithms approximate can often be integrated into the algorithms by similarly manipulating the distances of non-object elements (e.g., the method of Chen et al. [2000], 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. [1999] suggest the criterion that o' is an approximate farthest neighbor of q if $d(q, o') \geq d(q, o) - \epsilon D$, 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).[1] This criterion can also be incorporated into the nearest neighbor and ranking algorithms in a straightforward manner.

Observe that what causes the algorithms to become approximate when the distance functions are modified is that the lower-bounding property of the d_t distance functions (or the upper-bounding property of the $\hat{d}_t$ functions) is no longer satisfied. If the original distance functions do not satisfy the lower-bounding property, the algorithms will also not return correct results, but possibly in unpredictable ways.

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

[1]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 [Duncan et al. 1999].

The indexing methods discussed in later sections 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,[2] but this leads to approximate query results.

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 nonnegativity 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 2.1). 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 $S \subset \mathbb{U}$, 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 4.1. *Given any three objects $q, p, o \in \mathbb{U}$, we have*

$$|d(q, p) - d(p, o)| \leq d(q, o) \leq d(q, p) + d(p, o). \tag{1}$$

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 11(a) illustrates the situation where the lower bound $|d(q, p) - d(p, o)|$ established in Lemma 4.1 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 11(b) illustrates the situation where the upper bound $d(q, p) + d(p, o)$ is nearly attained.

[2] Distance functions for DNA data that are based on edit distance are usually of this nature [Smith and Waterman 1981].

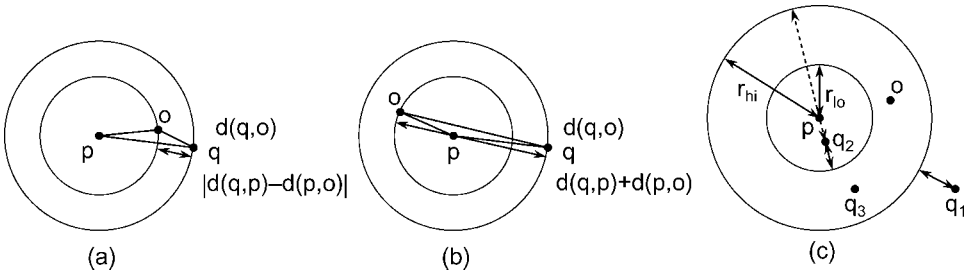


Fig. 11. Illustration of distance bounds. Given d(q,p) and d(p,o), a lower bound (a) and an upper bound (b) can be established for d(q,o). (c) Given that $r_{lo} \leq d(p, o) \leq r_{hi}$, we can establish lower and upper bounds on d(q,o). Three positions are shown for q, demonstrating three cases that can arise.

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 11(c), 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. Note that the lower bound on $d(q_3, o)$ is zero, since q_3 is inside the "shell" around p.

LEMMA 4.2. *Let o and p be objects in $\mathbb{U}$ such that $r_{lo} \leq d(o, p) \leq r_{hi}$. The distance $d(q, o)$ from $q \in \mathbb{U}$ to o can be bounded as follows, given $d(q, p)$:*

$$\max\{d(q, p) - r_{hi}, r_{lo} - d(q, p), 0\} \leq d(q, o) \leq d(q, p) + r_{hi}. \tag{2}$$

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 $d(o, p) \leq r_{hi}$, we obtain $d(q, p) - d(q, o) \leq d(o, p) \leq r_{hi}$, which implies that $d(q, p) - r_{hi} \leq d(q, o)$ (e.g., see q_1 in Figure 11(c)). Similarly, we can combine the triangle inequality and the lower bound on $d(o, p)$ to obtain $r_{lo} \leq d(o, p) \leq d(q, o) + d(q, p)$, which implies that $r_{lo} - d(q, p) \leq d(q, o)$ (e.g., see q_2 in Figure 11(c)). Either or both of these lower bounds can be negative (e.g., for q_3 in Figure 11(c)), whereas distance values are required to be nonnegative. Thus, the overall lower bound in Eq. (2) is obtained by taking the maximum of zero and these two lower bounds. The upper bound in Eq. (2) is obtained by a straightforward application of the triangle inequality and the upper bound on $d(o, p)$, i.e., $d(q, o) \leq d(q, p) + d(o, p) \leq d(q, p) + r_{hi}$ (e.g., see the broken arrow from q_2 through p to the outer boundary in Figure 11(c)). □

In some situations, the distance $d(q, p)$ in Lemma 4.2 may not be known exactly. The next lemma establishes bounds on the distance from q to o in such circumstances:

LEMMA 4.3. *Let o, p, and q be objects in $\mathbb{U}$ for which $d(o, p)$ is known to be in the range $[r_{lo}, r_{hi}]$ and $d(q, p)$ is known to be in the range $[s_{lo}, s_{hi}]$. The distance $d(q, o)$ can be bounded as follows:*

$$\max\{s_{lo} - r_{hi}, r_{lo} - s_{hi}, 0\} \leq d(q, o) \leq r_{hi} + s_{hi}. \tag{3}$$

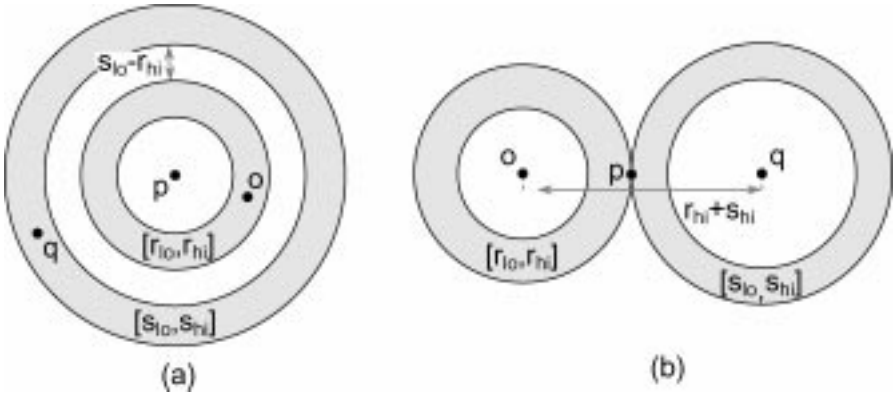


Fig. 12. (a) The lower bound on d(q,o) is illustrated for the case when d(p,o) is in the range $[r_{lo}, r_{hi}]$ and d(p,q) is in the range $[s]_{lo}, s_{hi}]$. (b) Illustration of how the upper bound on d(q,o) can be attained when d(p,o) and d(p,q) are in these ranges.

PROOF. Substituting s_{lo} for the first instance of $d(q, p)$ in Eq. (2) can only reduce the lower bound. Thus, we find that $s_{lo} - r_{hi} \leq d(q, o)$. 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 $r_{lo} - s_{hi} \leq d(q, o)$. Similarly, substituting s_{hi} for the last instance of $d(q, p)$ in the equation increases the upper bound, so we obtain $d(q, o) \leq r_{hi} + s_{hi}$. □

Clearly, the roles of the two ranges in Lemma 4.3 are symmetric. For an intuitive understanding of the lower bound, imagine two shells around p, one with radius range $[r_{lo}, r_{hi}]$ (where o is allowed to reside) and the other with radius range $[s_{lo}, s_{hi}]$ (where q is allowed to reside). As illustrated by the shaded arrow in Figure 12(a), the minimum distance between q and o is equal to the space between the shells, if any. 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 12(b). Observe that Lemmas 4.1–4.3 are increasingly general and that the earlier ones can be proved based on the later ones; we chose to present the lemmas in this manner for didactic reasons.

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 4.4. *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 $d(q, o)$:*

$$\max\left\{\frac{d(q, p_1) - d(q, p_2)}{2}, 0\right\} \leq d(q, o). \tag{4}$$

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

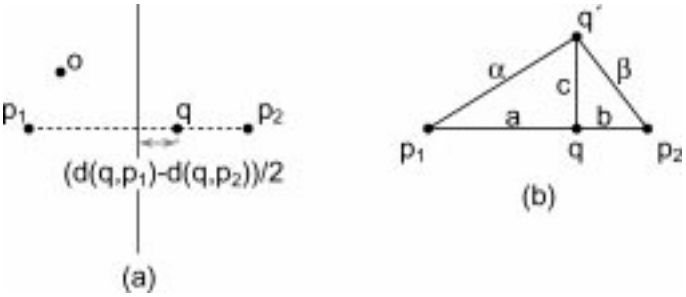


Fig. 13. (a) Lower bound on d(q,o), illustrated in a two-dimensional Euclidean space when q is on the line between p_1 and p_2, closer to p_2, while o is closer to p_1. (b) The lower bound can be shown to decrease when q is moved off the line (e.g., to q').

$d(q, p_1) - d(q, p_2) \leq 2d(q, o)$, which yields the first component of the lower bound in Eq. (4), the second component being furnished by nonnegativity. □

One way to get some intuition about this result is to consider the situation shown in Figure 13(a), 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 13(a)), 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 distance values increase,[3] 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 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.

5. BALL PARTITIONING METHODS

In this section we describe a number of ball partitioning methods. Section 5.1 presents the vp-tree in great detail. Section 5.2 reviews variants of the vp-tree, while Section 5.3 discusses a few other search structures that employ some form of ball partitioning.

[3]Figure 13(b) 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 $(d(q', p_1) - d(q', p_2))/2 < (d(q, p_1) - d(q, p_2))/2$.

5.1 The VP-Tree

The vp-tree (Vantage Point Tree) [Yianilos 1993] is an example of an indexing method that uses ball partitioning (and thus is a variant of the metric tree [Uhlmann 1991a, 1991b]). In this method, we pick a pivot p from S (termed a vantage point in Yianilos [1993]), 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:

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

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 1(a). In fact, we can define *bounding values* for each subset S_1 and S_2. In particular, for $o \in S_i$, we have $d(p, o) \in [r_{i,\text{lo}}, r_{i,\text{hi}}]$, for some bounding values $r_{i,\text{lo}}$ and $r_{i,\text{hi}}$. Given only the radius r, the known bounds are $[r_{1,\text{lo}}, r_{1,\text{hi}}] = [0, r]$ and $[r_{2,\text{lo}}, r_{2,\text{hi}}] = [r, \infty]$ (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. Doing this can yield improved search performance, but perhaps at the price of excessive storage cost.

5.1.1 *Pivot Selection.* The simplest method of picking pivots is to simply select at random. Yianilos [1993] 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 [1977]). 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 14(a) is greater than that in Figure 14(b)), which Yianilos [1993] 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 14(c)). 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 14). Observe also that the length l of the partitioning

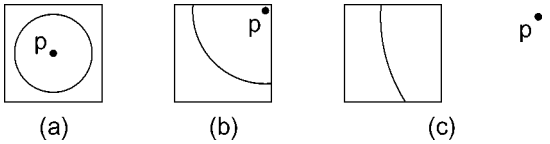


Fig. 14. Depiction of partitions of a set of points in a two-dimensional Euclidean space, assumed to be uniformly distributed in a cube c, for pivot points (a) in the center of c, (b) in a corner of c, and (c) outside c.

arc decreases even more as the pivot is moved further away from c (e.g., see Figure 14(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 $S\backslash\{p\}$ that are closest and farthest from p, as proposed by Chávez et al. [2001b] (and inspired by Burkhard and Keller [1973]). This yields a partition into equal-width "shells" around p. Chávez et al. [2001b] 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 [Chávez et al. 2001b]. 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. Nevertheless, even when splitting at the median, the tree can become unbalanced if we support dynamic insertions, but this can be alleviated with periodic reorganization.

5.1.2 *Search.* Clearly, search algorithms are fundamentally the same regardless of how the pivot and ball radius are determined, since the basic structure is the same. Let us examine how a range query algorithm similar to that in Section 3.1 would proceed. 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. To do so, we must 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 15(a), the left subtree (for the objects inside the ball) need not be visited, while in Figure 15(b), the left subtree must be visited. The lower bounds are provided by Eq. (2) in Lemma 4.2. In particular, by applying the equation with $r_{\text{lo}} = 0$ and $r_{\text{hi}} = r$, we find 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 find 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 $\max\{d(q, p) - r, 0\} \leq \epsilon$ and the right child if and only if $\max\{r - d(q, p), 0\} \leq \epsilon$.

The above description of performing range search on a vp-tree immediately suggests how to derive a search hierarchy according to the framework presented in Section 2.1, thus making it possible to apply all the algorithms in Section 3. The elements of the search hierarchy correspond to the objects (type 0) and the nodes (type 1) in the vp-tree. Figure 16(b) is the resulting search hierarchy for the small sample vp-tree of Figure 16(a). In Figure 16(b), the circles represent

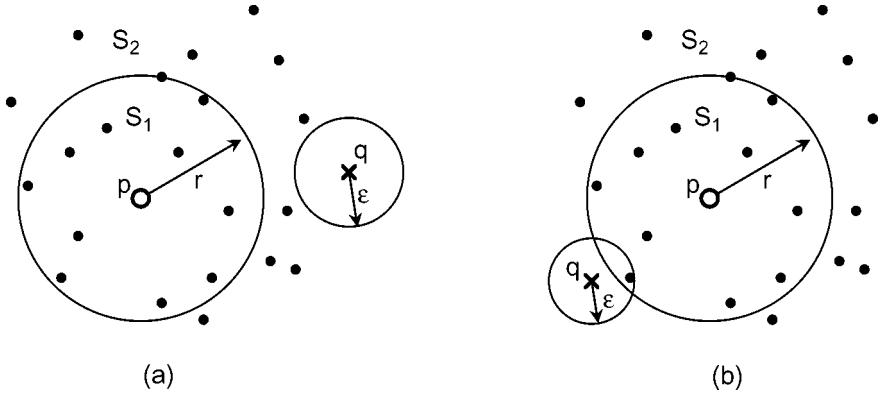


Fig. 15. During a range query with query radius ϵ, the subtree corresponding to the inside of the ball need not be visited in (a) while it must be visited in (b).

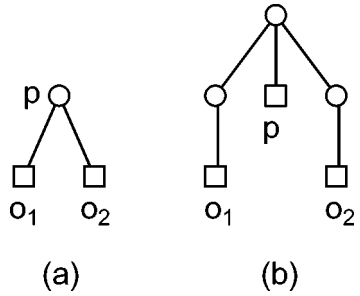


Fig. 16. (a) An example vp-tree for three objects p, o_1, and o_2, and (b) the search hierarchy induced by the vp-tree.

elements of type 1, and the squares represent elements of type 0. Thus, a node element (i.e., of type 1) produces an element of type 0 corresponding to its pivot, and two elements of type 1 corresponding to its two children, if present.

Having defined the structure of the search hierarchy, we must specify the distance functions d_t for each element type t (i.e., 0 and 1, in this case). 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. As we showed above for the range query, such lower bounds are readily derived based on Lemma 4.2. In particular, for a node element e with pivot p and a child element e' with bounding values $[r_{\text{lo}}, r_{\text{hi}}]$, the distance function for e' (of type 1) is:

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

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 $r_{\text{hi}} = \infty$).[4] Furthermore, for either case, it also accounts for q being either

[4]As we pointed out before, tight distance bounds for each subtree could be stored in each vp-tree node instead of just the median [Yianilos 1993], thereby causing d_1 and $\hat{d}_1$ to yield improved bounds.

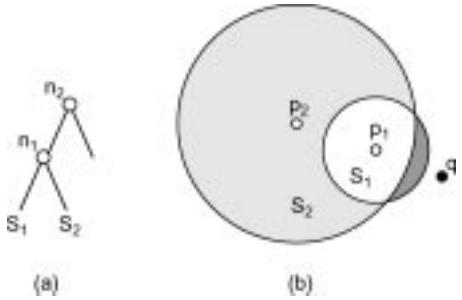


Fig. 17. An example of pivots p_1 and p_2 for two nodes n_1 and n_2, respectively, in a vp-tree (a), where n_1 is the left child of n_2. The regions for the left and right child of n_1 are denoted S_1 and S_2, respectively, and shown in (b).

inside or outside the region for e_1 (i.e., inside or outside the ball around p of radius r).

The upper-bound distance function $\hat{d}_1$ can also be derived from the result of Lemma 4.2 (i.e., by Eq. (2)):

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

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 3.4).

In the vp-tree, it is quite possible for the "ball" around the pivot p_1 for a node n_1 to not be completely inside the "ball" around the pivot p_2 of its parent node n_2, as depicted in Figure 17(b), which shows the ball regions corresponding to the vp-tree in Figure 17(a). Thus, it is possible for the lower-bound distance from q to a child node to be smaller than the lower-bound distance to its parent node, which is true in the figure, since $d(q, p_1) - r_1 < d(q, p_2) - r_2$. However, this does not affect the correctness of the algorithms presented in Section 3, since the objects in the subtree rooted at n_1 are still no closer to q than $d(q, p_2) - r_2$, even though the lower bound based on the pivot p_1 and its ball radius r_1 is smaller. In other words, the objects in the left subtree of n_1 must be somewhere in the white region inside the ball for p_1 (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_2.

5.2 Variants of the VP-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 [1993] 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

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).

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 4.1 allows us to bound the distance between q and o, that is, $|d(q, p) - d(p, o)| \leq d(q, o) \leq d(q, p) + d(p, o)$. Thus, when performing a range query with radius ϵ, we can safely discard o if $|d(q, p) - d(p, o)| > \epsilon$ or directly include it in the result if $d(q, p) + d(p, o) \leq \epsilon$.

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 [1993], 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 [1997; 1999]. 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 18.

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 18(a)). Bozkaya and Ozsoyoglu [1997, 1999] 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^{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

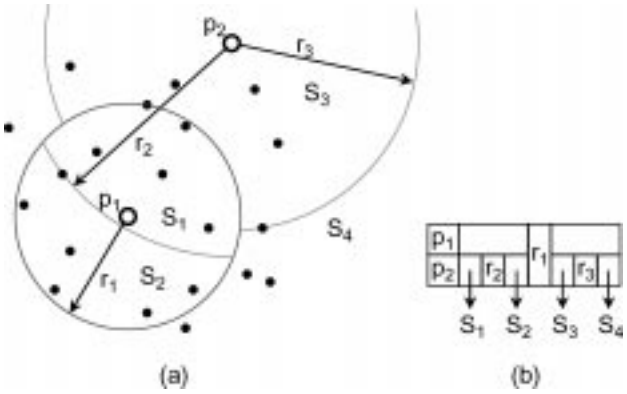


Fig. 18. (a) Possible top-level partitionings of a set of objects (depicted as two-dimensional points) in an mvp-tree where two pivots are used in each node, and (b) a depiction of the corresponding mvp-tree node. The second pivot, p_2, partitions the inside of the ball for p_1 into subsets S_1 and S_2, and the outside of the ball into subsets S_3 and S_4.

objects are stored in the node (a version of the mvp-tree in which pivots are not used in leaves is also considered in Bozkaya and Ozsoyoglu [1999]). Thus, the leaf node pivots essentially function like the ancestral pivots.

Search hierarchies for the above vp-tree variants are readily suggested by their structure, similar as we saw for the vp-tree in Section 5.1. Thus, we would have elements for the nodes and objects. As described above for the range query, in algorithms where a maximum distance bound is defined, the distance information for ancestral pivots can often be used to discard objects without computing their actual distance. However, for the ranking query, a maximum distance bound is usually not available. Thus, to exploit the ancestral pivots in this case, we can define "approximate object" elements that would be produced for objects that appear in leaf nodes, and whose lower-bound distance function is defined on the basis of the ancestral pivots. For a further discussion, please refer to Hjaltason and Samet [2000].

5.3 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.

The vp-tree, one of the more popular 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 [Knuth 1998, 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 $|d(o, p) - r| \leq \delta$.[5]

Among the earliest published work on distance-based indexing was that of Burkhard and Keller [1973]. 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 $S_1, S_2, \ldots, S_m$ 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 5.2). In fact, as pointed out by Chávez et al. [2001b], 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. [1994] proposed a variant of Burkhard and Keller's approach that they termed the *fixed-queries tree* (also known as an *fq-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 (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 h. 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* (also known as an *fq-array*) [Chávez et al. 2001a], 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. Thus, each bit string is the result of encoding the h distances from an object to the h pivots, by concatenating b-bit representations of each distance, so that the most significant digits in the bit string correspond to the pivots closer to the root. In the fixed query array, movements in the equivalent fixed-height fixed-queries tree are simulated with binary search.

Yianilos [1998] proposed a variant of vp-trees termed the *excluded middle vantage point forest* (see also Gennaro et al. [2001]) that is intended for

[5]The idea of a loose partition so that the children of a node are not disjoint is also used in the os-tree [Maneewongvatana and Mount 2001a, 2001b], KD2-tree [van Oosterom 1990; van Oosterom and Claassen 1990], spatial k-d tree [Ooi et al. 1987], and hybrid tree [Chakrabarti and Mehrotra 1998, 1999].

radius-limited nearest neighbor search, that is, 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^* [Yianilos 1998], which is of limited value in most similarity search applications.

6. GENERALIZED HYPERPLANE PARTITIONING METHODS

In this section, we describe a number of generalized hyperplane partitioning partitioning methods. Section 6.1 presents the gh-tree in great detail. Section 6.2 reviews GNAT (Geometric Near-neighbor Access Tree). Section 6.3 describes the bisector tree and the mb-tree, while Section 6.4 discusses a few other search structures that employ some form of generalized hyperplane partitioning.

6.1 The GH-Tree

Uhlmann [1991b] defined a metric tree using generalized hyperplane partitioning, which has been termed a gh-tree by later authors [Bozkaya and Ozsoyoglu 1999; Brin 1995; Fu et al. 2000]. 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 as in Faloutsos and Lin [1995], McNames [1998], McNames et al. [1999], and Merkwirth et al. [2000]) and splits the set of remaining objects based on the closest pivot (see Figure 1(b)):

$$S_1 = \{o \in S \setminus \{p_1, p_2\} \mid d(p_1, o) \leq d(p_2, o)\}, \quad \text{and}$$
$$S_2 = \{o \in S \setminus \{p_1, p_2\} \mid d(p_2, o) < d(p_1, o)\}.$$

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 [1991b] 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

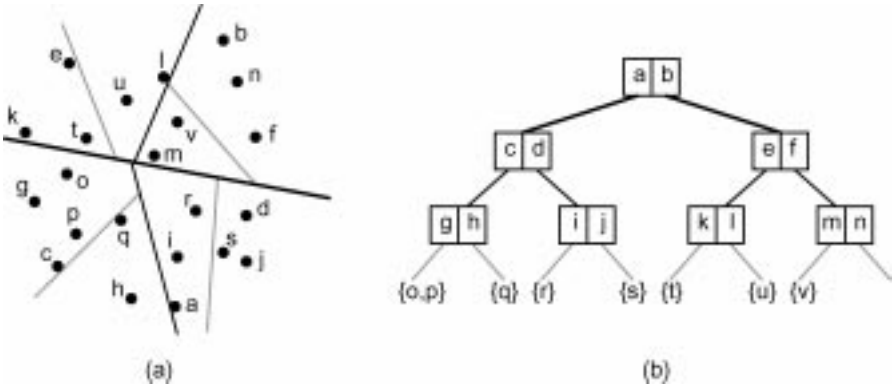


Fig. 19. (a) A possible space partitioning produced by the gh-tree for a set of points in a two-dimensional Euclidean space, and (b) its tree representation.

m is fixed at 0; the discussion is easily generalized to other values. A sample gh-tree is shown in Figure 19.

The term "generalized hyperplane partitioning" is derived from the fact that if the objects are points in an n-dimensional Euclidean space, the resulting partitioning is equivalent to one 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 4.4 shows that it is possible to derive a lower bound on the distance from q to some object in a partition; an upper bound cannot be determined, on the other hand, since objects can be arbitrarily far from from p_1 and p_2. In particular, for a range query with query radius ϵ, the left subtree of a node n with pivots p_1 and p_2 must be visited if and only if $\frac{d(q,p_1)-d(q,p_2)}{2} \leq \epsilon$ and the right one must be visited if and only if $\frac{d(q,p_2)-d(q,p_1)}{2} \leq \epsilon$. Observe that the lower bound obtained from Lemma 4.4 is much weaker than would be obtained in a Euclidean space by using the hyperplane directly, since the bound decreases as q moves parallel to the hyperplane away from p_1 or p_2 (recall Figure 13).

To augment the information given by the closeness relationship to the two pivots in each node, it is common in gh-tree variants to include for each pivot the maximum distance to an object in its subtree (the result is termed a *bisector tree (BST)* [Kalantari and McDonald 1983]). Thus, for pivot p_i $(i = 1, 2)$, we define the radius $r_i = \max_{o \in S_i}\{d(p_i, o)\}$. Clearly, these radii can be used during search in the same way that the ball radii are used in the vp-tree, by using Lemma 4.2.

A number of researchers have independently invented structures similar to this "augmented" gh-tree. For example, this is the case with the work of Noltemeier et al. [1993] (further described in Section 6.3) and Merkwirth et al. [2000]. The latter authors also describe a best-first nearest neighbor algorithm that is equivalent to the algorithm presented in Section 3.2.2 when applied on the search hierarchy defined below.

A search hierarchy for the gh-tree naturally arises from its structure, similar as for the vp-tree (see Section 5.1), with elements for objects and nodes, and where node elements produce an object element for each pivot and a node element for each child. For a node element e with pivots p_1 and p_2, a lower-bound distance function d_1 is defined for the two node elements e' that are children of e, corresponding to S_1 and S_2, respectively. In particular, in defining d_1, we can make use of both the lower bound that follows from Lemma 4.4 as well as the lower bound based on the radii r_i of the balls around pivots p_i using Lemma 4.2, taking the maximum of the two:

$$d_1(q, e') = \begin{cases} \max\left\{\frac{d(q,p_1)-d(q,p_2)}{2}, d(q, p_1) - r_1, 0\right\}, \\ \qquad \text{if } e' \text{ is the child of } e \text{ corresponding to } S_1, \\ \max\left\{\frac{d(q,p_2)-d(q,p_1)}{2}, d(q, p_2) - r_2, 0\right\}, \\ \qquad \text{if } e' \text{ is the child of } e \text{ corresponding to } S_2. \end{cases}$$

Unfortunately, we cannot use Lemma 4.4 to define an upper bound distance function $\hat{d}_1$ since Lemma 4.4 does not provide an upper bound on the distance from q to any point o in a hyperplane bounded region e'. However, we can make use of the radii r_i of the balls around pivots p_i, as defined above again based on Lemma 4.2, to yield:

$$\hat{d}_1(q, e') = \begin{cases} d(q, p_1) + r_1, & \text{if } e' \text{ is the child of } e \text{ corresponding to } S_1, \\ d(q, p_2) + r_2, & \text{if } e' \text{ is the child of } e \text{ corresponding to } S_2. \end{cases}$$

6.2 GNAT

GNAT (Geometric Near-neighbor Access Tree) [Brin 1995] is a generalization of the gh-tree, where more than two pivots (termed *split points* in Brin [1995]) 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 [1995] 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 [1995] describes for choosing the pivot objects is based on a philosophy similar to that of Yianilos [1993] for the vp-tree (and also suggested by others [Bozkaya and Ozsoyoglu 1999; Shapiro 1977]). 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.

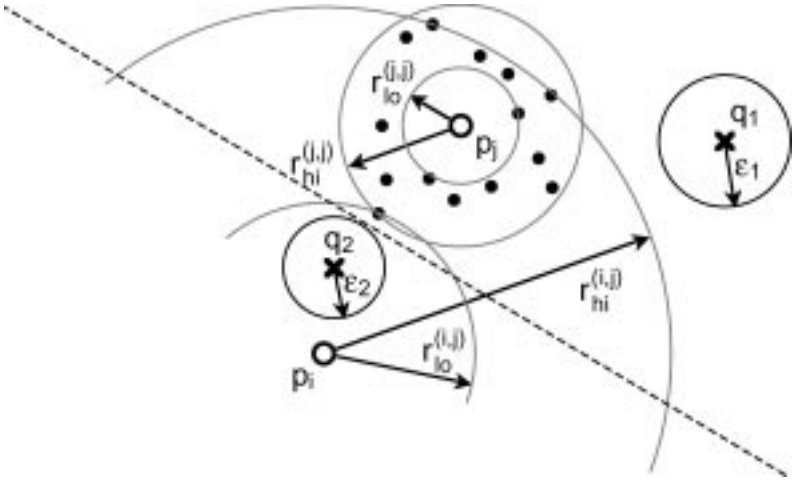


Fig. 20. Depiction of the bounds for two pivots p_i and p_j in a GNAT. The broken line indicates the hyperplane between the pivots (here, it is a line, but, in general, there may be no such simple characterization), and the filled circles denote objects associated with p_j. Query object q_1 with query radius ϵ_1 results in eliminating p_j and its subset from the search since $d(q, p_i) - \epsilon > r_{hi}^{(i,j)}$. Similarly, query object q_2 with query radius ϵ_2 eliminates p_j from the search since $d(q_2, p_i) + \epsilon < r_{lo}^{(i,j)}$.

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 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 $o \in S_j \cup \{p_j\}$; that is, $r_{\text{lo}}^{(i,j)} = \min_{o \in S_j \cup \{p_j\}}\{d(p_i, o)\}$ and $r_{\text{hi}}^{(i,j)} = \max_{o \in S_j \cup \{p_j\}}\{d(p_i, o)\}$. Although not mentioned by Brin [1995], 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 (since there are $m \cdot (m-1) + m$ ranges altogether).[6] Figure 20 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.

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 Brin [1995]). Clearly, we could simply

[6]This approach of storing m^2 ranges should be contrasted with the approach of Kamgar-Parsi and Kanal [1985] and Larsen and Kanal [1986] who only store the m ranges formed by $r_{\text{lo}}^{i,i}$ and $r_{\text{hi}}^{i,i}$ while in the augmented gh-tree [Kalantari and McDonald 1983; Merkwirth et al. 2000; Noltemeier et al. 1993], described in Section 6.1, only $r_{\text{hi}}^{i,i}$ is stored.

apply Lemma 4.4, 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 4.2), 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 other pivots p_j (i.e., $j \neq i$). 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 [1995], 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 $d(q, p_i) \leq \epsilon$, we add p_i to the query result. Next, for all $p_j \in P$, we discard p_j from P if $d(q, p_i) - \epsilon > r_{\text{hi}}^{(i,j)}$ or $d(q, p_i) + \epsilon < r_{\text{lo}}^{(i,j)}$ (or, equivalently, if $\max\{d(q, p_i) - r_{\text{hi}}^{(i,j)}, r_{\text{lo}}^{(i,j)} - d(q, p_i)\} > \epsilon$, based on Lemma 4.2). Figure 20 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 $d(q_1, p_i) - \epsilon_1 > r_{\text{hi}}^{(i,j)}$ and $d(q_2, p_i) + \epsilon_2 < r_{\text{lo}}^{(i,j)}$. 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.

In the range query algorithm above, the query radius ϵ plays a crucial role in pruning pivots in P. The nearest neighbor algorithms in Section 3.2 can be adapted for GNAT in a similar way, by using MaxDist(*NearestList*) for this pruning. However, it is difficult to derive a generic search hierarchy that makes use of all the inter-pivot distances $r^{(i,j)}$, on which we can apply the incremental ranking algorithm of Section 3.3. Some possible strategies are outlined in Hjaltason and Samet [2000].

6.3 Bisector Trees and mb-Trees

As pointed out in Section 6.1, it is often common to augment the gh-tree by including for each pivot the maximum distance to an object in its subtree yielding what are, in effect, *covering balls*. The resulting data structure is called a bisector tree (BST) [Kalantari and McDonald 1983]. The motivation for adding the covering balls is to speed up the search by enabling the pruning of elements whose covering balls are farther from the query object than the current

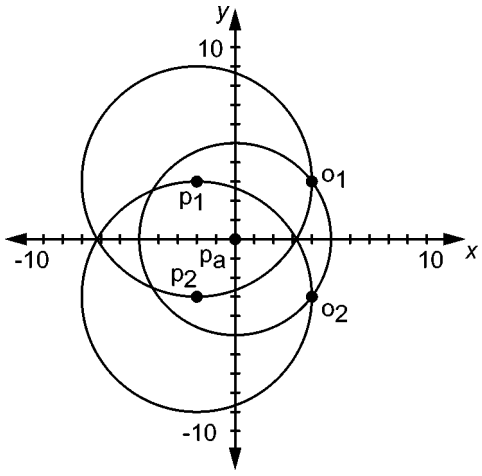


Fig. 21. Example where the radius of the covering ball around pivots p_1 and p_2 in element e_a are be greater than the radius of the covering ball around the pivot p_a in the ancestor element e of e_a.

candidate nearest neighbor (the farthest of the k candidate nearest neighbors) or are outside the range for a range query. Naturally, the utility of the covering balls for pruning increases as their radii decrease. Thus, as the search hierarchy is descended, it is desirable for the covering balls to become smaller thereby leading to more pruning. Unfortunately, the radii of the covering balls of the children are not necessarily smaller than the radii of the covering balls of their ancestors. For example, consider a two-dimensional space with a Euclidean distance metric as shown in Figure 21. Let p_a be at the origin, p_1 be at $(-2, 3)$, p_2 be at $(-2, -3)$, o_1 be at $(4, 3)$, and o_2 be at $(4, -3)$. Let o_1 and o_2 be the farthest objects from p_a. Moreover, let o_1 be the farthest object from p_1 and let o_2 be the farthest object from p_2. We now see that the radii of the covering balls around p_1 and p_2 are both 6 and are larger than the radius of the covering ball of p_a which is 5.

Dehne and Noltemeier [1987] characterize a child element in the bisector tree as *eccentric* when the radius of its covering ball is larger than the radius of the covering ball of its ancestor element. Eccentricity of children is disadvantageous for pruning because the radii of the covering balls increase as the search hierarchy is descended. The potential for having eccentric children is viewed by some (e.g., Dehne and Noltemeier [1987] and Noltemeier et al. [1992, 1993]) as a drawback of the bisector tree. Hence, it has been proposed to modify the definition of the bisector tree so that one of the two pivots in each nonleaf node n, except for the root, is inherited from its parent node—that is, of the two pivots in the parent of n, the one that is inherited is the one that is closer to each object in the subtree rooted at n. In other words, each pivot will also be a pivot in the child node corresponding to that pivot. Since this strategy leads to fewer pivot objects, its use can be expected to reduce the number of distance computations during search (provided the distances of pivot objects are propagated downward during search), at the possible cost of a worse partitioning and a deeper tree if the decomposition process is only halted when each mb-tree leaf

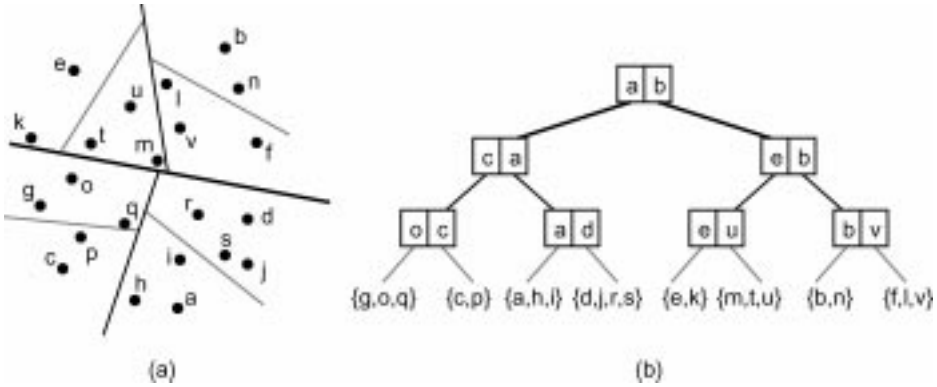


Fig. 22. (a) A possible space partitioning produced by the mb-tree for a set of points in a two-dimensional Euclidean space of Figure 19, and (b) its tree representation.

node contains just one object. Of course, the radius of the covering ball around the pivots (i.e., the maximum distance to objects in the corresponding subtree) is also stored and used for pruning. The result is termed the *monotonous bisector tree* (abbreviated below as *mb-tree*) and was proposed by Noltemeier et al. [1992; 1993] (and used by Bugnion et al. [1993]).

mb-trees were originally intended for use with point data and Minkowski metrics. However, the mb-tree can be used with arbitrary metrics. An extension of the mb-tree, the *mb*-tree*, accommodates complex objects, such as lines and polygons. The TLAESA method of Micó et al. [1996] 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 9.3 for more details).

Figure 22(a) is an example of the space partitioning produced by an mb-tree corresponding to the set of points in the two-dimensional space stored in the gh-tree in Figure 19, while Figure 22(b) is its tree representation, where the same number of partitions were made at each level as in the corresponding gh-tree. Notice that two points are associated with each nonleaf node of the tree representation thereby defining the partition of the underlying space, while each of the points is repeated in the next level of the tree. However, in the interest of saving storage, all of the points are actually stored in the leaf nodes of the mb-tree. Nonleaf nodes store pointers to their corresponding leaf node entries.

It should be clear that many different configurations are possible when constructing an mb-tree for a set of objects. This is because there are many options for which objects to choose as pivots at each step of the decomposition. In constructing our example mb-tree, we have followed a strategy that tries to associate approximately the same number of objects with each leaf node, while using the same initial partition as in the example gh-tree (i.e., using pivots a and b). Note also that if we would have decomposed the underlying space so that every leaf node would contain just one point, then the resulting mb-tree would be considerably deeper than the corresponding gh-tree. In particular, for a set of $N (N > 1)$ objects, the fully decomposed mb-tree always requires $N - 1$

nonleaf nodes, while the corresponding gh-tree may need as few as $N/2$ nodes in total as each node can contain as many as 2 objects, and there is no need to distinguish between leaf and nonleaf nodes in a gh-tree as they are both used to store the points.

It is interesting to observe that the decomposition rule on which the mb-tree is based is analogous to a classical decomposition rule in applications involving spatial data such as points where a region is decomposed if it contains more than one point. This is the case for a PR quadtree [Orenstein 1982; Samet 1990], PR k-d tree [Orenstein 1982], as well as variants of k-d trees [Bentley 1975] such as the adaptive k-d tree [Friedman et al. 1977] and the BSP tree [Fuchs et al. 1980]. The BSP tree analogy is different from the others as the partitioning hyperplanes are not necessarily orthogonal. The analogy also holds for bucket variants of these structures where the decomposition is based on the region containing $k > 1$ points. In all of these examples, the pivot objects play the same role as the points. The principal difference is that the boundaries of the regions associated with the points are represented explicitly while they are implicit in the mb-tree and thus it is easy to use the mb-tree with data from an arbitrary metric space. In particular, in the latter, each partition is defined by two pivots and the set of objects that are closer to one of them than to the other. Thus the pivot objects play a similar role to control points in Bezier methods in modeling curves and surfaces in CAD (computer-aided design) applications (e.g., Foley et al. [1990]) in the sense that just as the curve in the latter is implicitly defined by the control points, the partitioning hyperplanes in the former are also implicitly defined by the pivot points.

Sharing a pivot with an ancestor, as proposed for the mb-tree, has the effect of guaranteeing that children are not eccentric (i.e., that the radii of the covering balls around pivots p_1 or p_2 in element e_a, which form e_1, are not greater than the radius of the covering ball around the pivot p_a in the ancestor element e of e_a). This non-eccentricity constraint on the radii of the covering balls is also satisfied when the more general stipulation that the distance from q to a nonobject element e_1 (i.e., $d(q, e_1)$) must be greater than or equal to the distance from q to an ancestor of e_1 (i.e., $d(q, e_a)$) holds—that is, they form a containment hierarchy and hence are monotonically nondecreasing.

It is important to note that although it is impossible to have a containment hierarchy when the children are eccentric, a containment hierarchy may also fail to exist when the children are not eccentric and even when the hierarchy is formed by sharing a pivot with an ancestor as in the mb-tree. For example, consider Figure 23 where the radius of the covering ball of child e_1 is smaller than the radius of the covering ball of its ancestor e_a yet the covering ball of e_1 is not completely contained in the covering ball of e_a.

Merkwirth et al. [2000] ignore the issue of eccentricity and instead force a containment hierarchy to exist by basing the lower bound $d(q, e_1)$ on the maximum of the distance of q from the children and the distance of q from the parent (i.e., $d(q, e_a)$). They use a conventional bisector tree (also referred to as an augmented gh-tree in Section 6.1). Nevertheless, it can be shown [Samet 2004] that for queries such as finding the nearest neighbors of q using the best-first method, the fact that the covering balls of the children do not contain

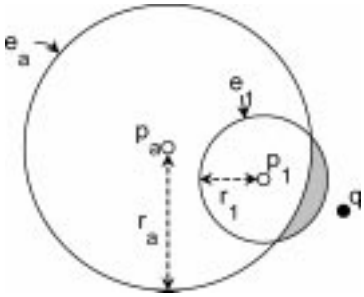


Fig. 23. Example illustrating that the covering ball of a descendant of a node may extend past the ball element of its ancestor.

any objects that are not in the covering balls of the ancestors means that taking the distance of the parent (i.e., $d(q, e_a)$) into account when computing $d(q, e_1)$ does not result in more objects or nodes being pruned from the search process. Equivalently, failing to take the distance of the parent into account, and thus just using the distance from q of the children does not cause additional objects or nodes to be visited during the search process. A similar statement can be made about range searching. The only requirement that must be met is that the distances of the nonobject elements lower bound the distances of the object elements (as pointed out in Section 2.1). Thus we can also safely ignore the issue of whether or not a containment hierarchy exists.

6.4 Other Methods Related to Generalized Hyperplane Partitioning

The gh-tree and GNAT (as well as the M-tree, described in Section 7) can be considered to be special cases of a general class of hierarchical clustering methods, as described by Burkhard and Keller [1973] and Fukunaga and Narendra [1975]. Using the description given by Burkhard and Keller [1973], a set S of objects is clustered into m subsets $S_1, S_2, \ldots, S_m$ using some criterion usually based on proximity. Next, 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.[7] 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 4.2, as was done for the vp-tree in Section 5.1 (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 some value D.[8] The clique property was found to reduce the number of distance computations and allow more pruning during search [Burkhard and Keller 1973], at the price of high preprocessing cost (for determining the cliques).

[7] GNAT [Brin 1995] maintains more comprehensive distance information in each node.

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

The manner in which we described the hierarchies that comprise the gh-tree, GNAT, mb-tree, etc. implies that they are built in a top-down manner. For example, recall that the hierarchy in GNAT is built by initially choosing m of the objects to serve as pivots, and then forming m sets each of which contains the objects that are closest to the corresponding pivot. Each of these m sets are recursively processed to yield m or some other number of subsets. Alternatively, the hierarchy can also be built in a bottom-up manner. In this case, the algorithms generally assume that the objects are initially processed using some clustering method to obtain k clusters with k cluster centers (e.g., k-centers Gonzalez [1985]). These clusters are subsequently processed by applying a hierarchical clustering method (e.g., Dasgupta [2002], Fukunaga and Narendra [1975] and Moore [2000]). These methods are very similar to those used to obtain object hierarchies such as the R-tree [Guttman 1984] and R*-tree [Beckmann et al. 1990] which make use of bounding hyperrectangles, and more specifically object hierarchies that make use of minimum bounding hyperspheres such as the SS-tree [White and Jain 1996], balltree [Omohundro 1989] (the anchors hierarchy [Moore 2000] is similar), and sphere tree [Hubbard 1996; van Oosterom and Claassen 1990]. Clustering is an active area of research in pattern recognition, machine learning, etc. but is beyond the scope of this article.

7. THE M-TREE

The distance-based indexing methods described in Sections 5 and 6 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 [Ciaccia et al. 1997; Ciaccia and Patella 2002] 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 [Guttman 1984] (which, in turn, was inspired by the B-tree) with the capabilities of static distance-based indexes.

7.1 Structure

In the M-tree, as in the R-tree, all the objects being indexed are referenced in the leaf nodes,[9] 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 Ciaccia et al. [1997]) 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 pivots in GNAT (see Section 6). However, unlike GNAT, all objects in S are stored in the leaf nodes of the M-tree, so an

[9]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.

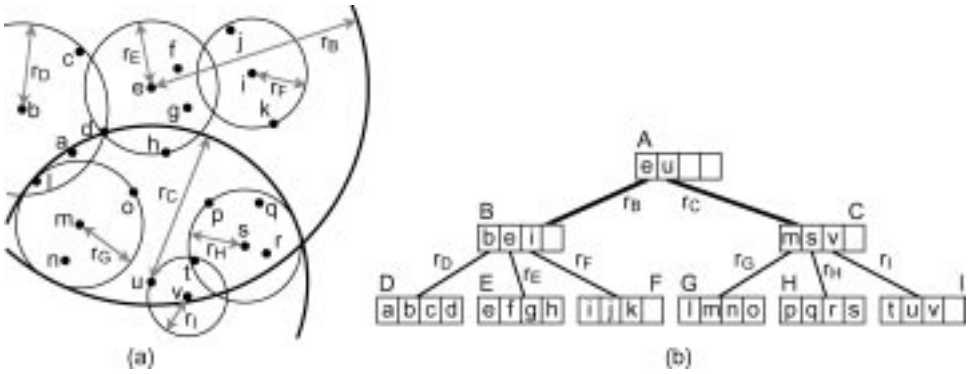


Fig. 24. (a) A possible space partitioning produced by the M-tree for a set of points in a two-dimensional Euclidean space, and (b) the corresponding M-tree node structure.

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 5.2). A sample M-tree is shown in Figure 24.

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 nonroot node, let *parent object* denote its associated pivot, that is, 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 [Ciaccia and Patella 1998]).

Being a dynamic structure, the M-tree can be built gradually as new data arrives [Ciaccia et al. 1997]. 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. [1997] considered a number of heuristics for choosing the child node to route an object into and for splitting overflowing nodes. Bulk-loading strategies [Ciaccia and Patella 1998] have also been developed for use when an M-tree must be built for an existing set of data

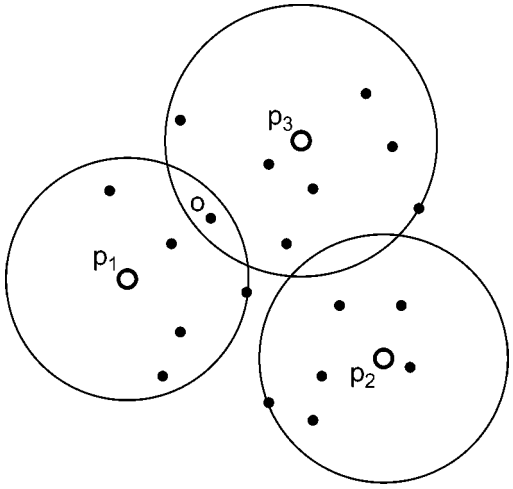


Fig. 25. Possible top-level partitionings of a set of objects (depicted as two-dimensional points) in an M-tree. Objects that fall into more than one "ball," like o, can be inserted into any of the corresponding subtrees.

objects. An example of root node partitioning in an M-tree for a set of objects is shown in Figure 25, 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.

Traina Jr. et al. [2002] introduced the *Slim-tree*, a variant of the M-tree with faster node insertion and node splitting algorithms. More importantly, the Slim-tree also features improved storage utilization, which is achieved, in part, by applying a post-processing step, termed the *Slim-down* algorithm. This algorithm attempts to reduce the overlap among node regions by moving entries between sibling nodes in iterative fashion. In particular, in the case of slimming down leaf nodes, an object o stored in a leaf node n is a candidate for being moved if (1) o is the object in n that is farthest from n's parent object p, and (2) the region of a sibling n' of n also covers o. Having identified such a candidate o, o is moved from n to n', and the covering radius of n is reduced, if possible, depending on the distance between p and the next farthest object in n. An empirical study showed that these modifications led to a reduction in the number of disk accesses as compared to the original M-tree [Traina Jr. et al. 2002].

7.2 Search

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, as in the algorithm of Section 3.1. Let n be a node that is being visited, and let p' be its parent pivot, that is, 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 $|d(q, p') - D| - r > \epsilon$, 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 4.1 and 4.3. Lemma 4.1 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 4.3 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, $|d(q, p') - D| - r \leq \epsilon$. 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 $d(q, p) - r > \epsilon$ and is a direct result of applying Lemma 4.2 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 $|d(q, p') - D| \leq \epsilon$ (since we know from Lemma 4.1 that $|d(q, p') - d(p', o)| = |d(q, p') - D| \leq d(q, o)$, so if $\epsilon < |d(q, p') - D| \leq d(q, o)$, we can immediately discard o without computing its distance), and only for such entries compute $d(q, o)$ and check whether $d(q, o) \leq \epsilon$. 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.

For k-nearest neighbor search, Ciaccia et al. [1997] propose using the distance of the farthest candidate kth nearest neighbor in place of ϵ in the pruning conditions. Thus, their algorithm is a variant of the best-first nearest neighbor algorithm of Section 3.2.2. Unfortunately, the pruning conditions are not applicable for the ranking query (Section 3.3), since the number of result objects is typically unknown in advance, implying that the search radius is unbounded. To overcome this dilemma, we introduce two new element types corresponding to approximate objects and approximate nodes to the search hierarchy. These additional elements provide a simple way to order 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. In fact, we show in Hjaltason and Samet [2000] that applying the algorithm of Section 3.2.2 on the resulting search hierarchy results in a more efficient

solution to the k-nearest neighbor problem than the algorithm of Ciaccia et al. [1997].

In particular, in the search hierarchy for the M-tree, 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. In an analogous manner to their use in defining the range query pruning rules, we can use Lemmas 4.1 and 4.3 to derive the lower-bound distance functions for elements of type 1 through 3:

$$\begin{aligned} d_1(q, e_1) &= \max\{|d(q, p') - D|, 0\}, \\ d_2(q, e_2) &= \max\{d(q, p) - r, 0\}, \quad \text{and} \\ d_3(q, e_3) &= \max\{|d(q, p') - D| - r, 0\} \end{aligned} \tag{5}$$

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

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

Correctness of these definitions is shown in Hjaltason and Samet [2000].

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. Doing this ensures 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, when we apply the range query algorithm of Section 3.1 to the search hierarchy, any elements of types 1 and 3 whose distances exceed ϵ represent objects and nodes, respectively, that we were able to prune without computing their actual distances from q. The same effect is achieved in using the pruning rules in the range query algorithm described above, so the two algorithms are in fact equivalent.

8. THE SA-TREE

Like GNAT, the sa-tree [Navarro 1999, 2002] was inspired by the Voronoi diagram [Voronoi 1909], a widely used method for nearest neighbor search in point data. Hence, to understand the sa-tree, it is important to look at how Voronoi diagrams can be used for performing search. In a Voronoi diagram for point data, for each "site" p, the Voronoi cell of p 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 as the graph where each object is a node and two nodes have an edge between them if their Voronoi cells have a common boundary (in an earlier publication, Navarro [1999] used the term "Voronoi graph"). In other words, the Delaunay graph is simply an explicit representation of neighbor relations that are implicitly represented in the Voronoi diagram; clearly, Delaunay graphs are closely related to Delaunay triangulations, the difference being that in the latter, the edges have an associated geometric shape. Searching a Delaunay graph for the nearest neighbor in S of a query point q in $\mathbb{U}$ starts with an arbitrary point in S, and proceeds to a neighboring point in S that is closer to q as long as this is possible. Once we reach a point o in S where the points in its neighbor set $N(o)$ in S (i.e., the points connected to o by an edge) are all farther away from q than o, 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 point 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, however, that the Delaunay graph is not necessarily unique as there can be several such minimal graphs (possibly even with a different number of edges).

8.1 Definition

In Section 6, 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 6.2, Voronoi cells (or, perhaps more accurately, Dirichlet domains [Brin 1995]) for objects cannot be constructed explicitly (i.e., their boundaries specified) if only interobject distances are available. Moreover, it is possible to show [Navarro 2002] that without more information about the structure of the underlying metric space $(\mathbb{U}, d)$, just knowing the set of interobject distances for a finite metric space (S, d), $S \subset \mathbb{U}$, is not enough to enable the construction of a valid

Delaunay graph for S based on d—that is, we also need information about the distances between the elements of S and the elements of $\mathbb{U}$. In other words, for the 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 $\mathbb{U}'$, (S, d) may have a Delaunay graph D that is not a Delaunay graph for (S', d'), or vice versa.[10] Moreover, for any two objects a and b, a finite metric space (S, d) exists whose Delaunay graph contains the edge between a and b. Hence, given only the interobject distances for a set S, the only way to construct a graph G such that G satisfies the Voronoi property for all potential query objects in $\mathbb{U}$ (i.e., contains all the edges in the Delaunay graph) is for G to be the complete graph—that is, the graph containing an edge between all pairs of nodes (each of which represents an object in S). However, such a graph is useless for search, as deciding on what edge to traverse from the initial object in S 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 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 (see the example in Figure 26 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, a smallest possible set $N(a) \subset S \setminus \{a\}$ is identified, such that x is in $N(a)$ iff for all $y \in N(a) \setminus \{x\}$, $d(x, a) < d(x, y)$. The set $N(a)$ is termed the *neighbor set* of a, by analogy with the Delaunay graph, and the objects in $N(a)$ are said to be the neighbors of a. 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 from the root b of each subtree S_b to the farthest object in S_b can also be stored in b—that is, $d_{\max}(b) := \max_{o \in S_b} d(o, b)$. Figure 26(b) shows a sample

[10] 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. Thus, this graph is a Delaunay graph for (S, d). However, it does not satisfy the Voronoi property 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. Thus, it is not a Delaunay graph for (S', d'). It is interesting to note that the graph with edges (a, b) and (b, c) (i.e., $N(b) = \{a, c\}$ and $N(a) = N(c) = \{b\}$) satisfies the Voronoi property for both (S, d) and (S', d') and thus it is a Delaunay graph for both (S, d) and (S', d'). Of course, this example does not invalidate our observation that knowledge of (S, d) is insufficient to determine the Delaunay graph.

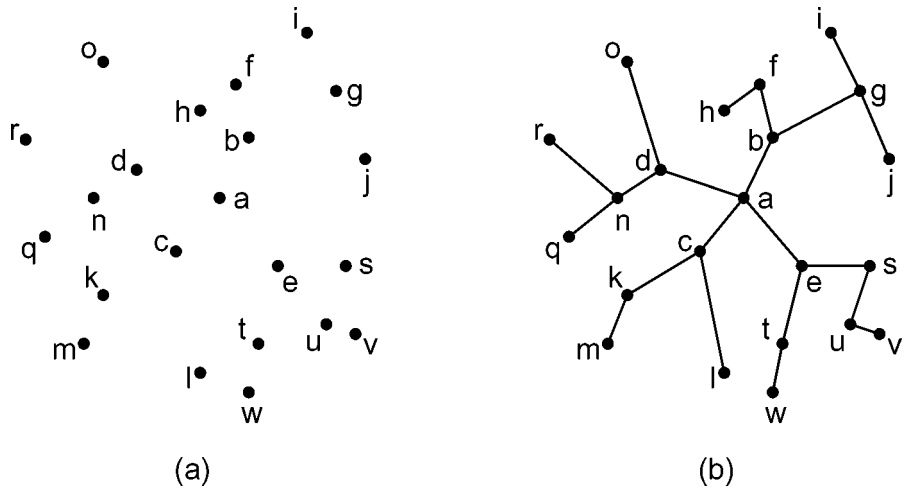


Fig. 26. (a) A set of points in a 2-dimensional Euclidean space, and (b) its corresponding sa-tree constructed using the algorithm of Navarro [2002] when a is chosen as the root.

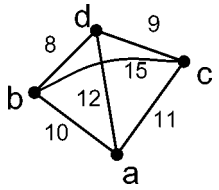


Fig. 27. An example of four points a, b, c, d where the sa-tree construction algorithm does not find the minimal neighbor set N(a).

sa-tree for the two-dimensional points a–w given in Figure 26(a), with a chosen as the root. In this example, $N(a) = \{b, c, d, e\}$. Note that h is not in $N(a)$ as h is closer to b than to a.

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 [2002] argues that its construction is an NP-complete problem. Thus, Navarro [2002] resorts to a heuristic for identifying the neighbor set. 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 26(b) has been constructed using this heuristic with a chosen as the root. An example of a situation where the heuristic would not find the minimal neighbor set is shown in Figure 27, where approximate distances between four points a through d are labeled. The minimum neighbor set of a in this case is $N(a) = \{d\}$ (and $N(d) = \{b, c\}$) whereas use of the heuristic would lead to $N(a) = \{b, c\}$ (and $N(b) = \{d\}$). 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.

8.2 Search

Using the sa-tree, it is easy to perform exact match queries (i.e., to search for an object in S) with the same procedure as in the Delaunay graph as described above. Of course, this is not very useful, as 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 4.4. In particular, if a is the object corresponding to the root node of an sa-tree, 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 4.4 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 a is not the root node for the sa-tree, it is possible to show [Hjaltason and Samet 2000, 2003a] that c can be chosen from the larger set $\bigcup_{a' \in A(b)}(\{a'\} \cup N(a'))$, where $A(b)$ is the set of ancestors of b and b is in $N(a)$. In other words, c is the closest to q among b's ancestors, and all of their immediate children.

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, the search is realized with a depth-first traversal of the tree, starting at the root. When at node a which is not a root, we first determine the object $c \in \bigcup_{a' \in A(b)}(\{a'\} \cup N(a'))$ where b is in $N(a)$ such that $d(q, c)$ is minimized. When at node a which is a root, then c is set to a. Next, the search traversal visits each child $b \in N(a)$, except those for which $(d(q, b) - d(q, c))/2 > \epsilon$, since, in this case, we know that $d(q, o) > \epsilon$ for any object o in the subtree associated with b. A similar argument can be used in deriving a search hierarchy, on which we can apply the other query algorithms presented in Section 3 (see Hjaltason and Samet [2000, 2003a]).

The sa-tree, as described above, is a static structure, in that the entire data set must be known in order to build it. Navarro and Reyes [2002] introduce a dynamic version of the sa-tree, that supports both insertions and deletions. In order to yield adequate performance, the dynamic version must relax the definition of neighbor sets. In particular, for a given object o inserted into the tree at time t, the properties described above for ancestors and siblings of ancestors of o only hold for those inserted before time t. The implication of this relaxed definition of neighbor sets is that when evaluating the lower-bound distances for objects in the subtree rooted at a node b, we can only make use of those ancestors and ancestor siblings of b that were inserted before b.

9. DISTANCE MATRIX METHODS

The distance-based indexing 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. A number of methods have been proposed that instead precompute some or all of the distances between the objects in S,

typically stored in a matrix, and use these distances when evaluating queries. In Sections 9.1 and 9.2, we describe AESA and LAESA, which are "pure" distance matrix methods, while in Section 9.3, we discuss other related methods, some of which are hybrid in that they use both a distance matrix and a hierarchy.

9.1 AESA

AESA (Approximating and Eliminating Search Algorithm) [Vidal Ruiz 1986; Vidal 1994][11] is a nearest neighbor algorithm that requires all $O(N^2)$ inter-object distances are precomputed 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 Vidal Ruiz [1986], 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 [Vidal Ruiz 1986]. 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 if 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

[11] The difference between Vidal Ruiz [1986] and Vidal [1994] 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 14 below).

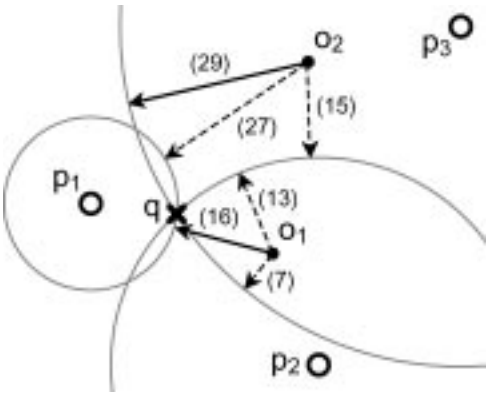


Fig. 28. An example of the computation of the lower bound distance d_{lo} for two objects o_1 and o_2 based on three pivots p_1, p_2, and p_3. The directed lines emanating from each object o_i to the different pivots $p_j (j = 1 \cdots 3)$ indicate the distance value $|d(q, p_j) - d(p_j, o_i)|$ (in parentheses). For each object o_i, the solid directed line is the longest and its corresponding distance value is the one that is used in computing the lower bound distance $d_{lo}(q, o_i)$.

computations, while the distance matrix has to be accessed many times for each query ($\Omega(N)$ for each nearest neighbor query[12]), although the distance computations are usually many orders of magnitude more complex than the operation of accessing the distance matrix.

The key to the search strategy employed by AESA [Vidal Ruiz 1986] in determining the nearest neighbor (and which also can be used for range searching) is the property described in Lemma 4.1: 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 $d_{lo}(q, o)$ on $d(q, o)$ for any object $o \in S \setminus S_c$ is

$$d_{lo}(q, o) = \max_{p \in S_c} \{|d(q, p) - d(p, o)|\} \tag{6}$$

An example of this computation is illustrated in Figure 28 for two objects o_1 and o_2 based on distances to three objects p_1, p_2, p_3 in S_c, which serve as pivots. There are three directed lines emanating from each object to the different pivots where the solid line is the longest and its distance is the one that is used to compute the lower bound distance $d_{lo}(q, o)$ where o is one of o_1 and o_2. Observe that, in this example, the lower bound for both objects is quite close to the actual distance of the objects (16.5 and 31 for objects o_1 and o_2, respectively).

For finding nearest neighbors, 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, that is, $d_{lo}(q, o) > d(q, o_n)$ (for range

[12]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.

search with query radius ϵ, the elimination criterion is $d_{\text{lo}}(q, o) > \epsilon$).[13] 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.

Initially, the algorithm sets S_c to $\emptyset$, S_u to S, and $d_{\text{lo}}(q, p)$ to ∞ for all $p \in S$. 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, with ties in lower-bound distances broken arbitrarily. Next, the algorithm computes $d(q, p)$, updates the nearest neighbor candidate o_n if necessary, and then eliminates objects from S_u that cannot be the nearest neighbor 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 $d(q, o_n)$. (In the case of range searching, we instead add p to the result set if $d(q, p) \leq \epsilon$ and eliminate objects from S_u for which $d_{\text{lo}}(q, o) > \epsilon$. Furthermore, when the algorithm terminates, we know that $d_{\text{lo}}(q, o) > \epsilon$ for all objects $o \in S \setminus S_c$.)

Observe that in the above algorithm, the lower-bound distance, $d_{\text{lo}}(q, o)$, for an object $o \in S_u$ need not be computed from scratch based on all $p \in S_c$ each time the algorithm makes use of 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 [1994], 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.[14] 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 [1990] explored several different choices which are described briefly in Section 9.3.

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. Unfortunately, this is not applicable for the ranking query, since its search radius is typically not bounded. There are a number of ways to define a search hierarchy based on the AESA framework, allowing the ranking algorithm in Section 3.3 to be applied.

[13] 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, $d_{\text{lo}}(q, o) > \epsilon$ means that $d(q, o) > \epsilon$.

[14] In the original formulation of AESA [Vidal Ruiz 1986], 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 [1994] 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.

Of course, applying the range query algorithm of Section 3.1 to such a hierarchy should be equivalent to the algorithm presented above, in terms of the number of distance computations. In Hjaltason and Samet [2000], we outline the definition of three search hierarchies, each of which has somewhat different characteristics. Conceptually, all three approaches involve the maintenance of 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. They differ in the way in which S_u is represented, which has implications in the cost of maintaining the lower-bound distance, $d_{\text{lo}}(q, o)$, associated with each object $o \in S_u$. Notice that in the case of a ranking query, S_u is larger than when it was used for range searching as no objects can be eliminated; instead, their reporting is deferred.

9.2 LAESA

Recall that AESA is impractical for all but the smallest data sets due to the large preprocessing and storage costs. LAESA (Linear AESA) [Micó et al. 1992, 1994] alleviates this drawback by choosing a fixed number M of pivots (termed *base prototypes* by Micó et al. [1992; 1994]), 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. [1994]. Essentially, this algorithm attempts to choose the pivots such that they are *maximally separated*, that is, as far away from each other as possible (a similar procedure was suggested by Brin [1995] for GNAT; see Section 6.2).

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. [1994] suggest treating the pivot objects in S_u differently than nonpivot 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. [1994] are simple, and call for

(1) selecting a pivot object in S_u over any nonpivot 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 defining a search hierarchy within the framework of LAESA, as shown in Hjaltason and Samet [2000]. However, since LAESA is practical for much larger data sets than AESA, some of the approaches that are feasible for AESA are too inefficient. Nevertheless, it is possible to define search hierarchies in which the use of the ranking algorithm of Section 3.3, without an *a priori* knowledge of the result size, has the same cost as the LAESA nearest neighbor search strategy [Hjaltason and Samet 2000]. See Chávez et al. [1999] for another search strategy that is in the spirit of the method of Nene and Nayar [1997].

9.3 Other Distance Matrix Methods

Shapiro [1977] 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 4.1). 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.[15]

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

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

to use next and to decide when to compute the 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 [1990] 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 interobject 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 Wang and Shasha [1990]), 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.[16] 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 [1990] that utilizes a generalized version of the triangle inequality[17] 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 9.1.

The nearest neighbor algorithm presented by Wang and Shasha [1990] 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.

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

[17] 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 $d(o_1, o_4) \geq d(o_1, o_2) - d(o_2, o_3) - d(o_3, o_4)$.

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 [1990] 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 [1990]. Wang and Shasha [1990] 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 9.1) 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 $D_{\text{hi}}[i, N+1]$.

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.

Micó et al. [1996] 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 [Micó et al. 1994] (see Section 9.2) and the mb-tree [Noltemeier et al. 1993] (see Section 6.3). 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. [1996]). 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 4.1 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 4.3, 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., Ramasubramanian and Paliwal [1992] and Vilar [1995]). For example, Ramasubramanian and Paliwal [1992] 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 [1995] 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 9.1), 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).[18] Note that ROAESA 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_{lo} values being greater than ϵ.

Interestingly, some of the search hierarchies that we devised for AESA and LAESA (see Hjaltason and Samet [2000]) 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.

10. ALTERNATIVE CHARACTERIZATION OF THE DIFFERENT DISTANCE-BASED INDEXING METHODS

An alternative way of distinguishing between some of the different distance-based indexing methods is on the basis of whether they are pivot-based or clustering-based (e.g., Chávez and Navarro [2000]). Pivot-based methods choose a subset of the objects in the data set to serve as distinguished objects, termed *pivot objects* (or more generally *pivots*), and classify the remaining objects in terms of their distances from the pivot objects. Pivot-based similarity searching

[18]Vilar [1995] 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.

algorithms make use of the known distances from the objects to different pivot objects to reduce the number of distance computations involving the query object that will be needed to respond to the query. The pivot objects, assuming without loss of generality that there are k of them, can often be viewed as coordinates in a k-dimensional space and the result of the distance computation for object x is equivalent to a mapping of x to a point $(x_0, x_1, \ldots, x_{k-1})$ where coordinate value x_i is the distance $d(x, p_i)$ of x from pivot p_i. The result is very similar to embedding methods discussed briefly in Section 1.

In this case, similarity search usually makes use of Lemma 4.1 which enables pruning an object x from further consideration as being within ϵ of query object q when $|d(q, p_i) - d(x, p_i)| > \epsilon$ for some coordinate corresponding to pivot p_i $(0 \leq i \leq k-1)$. Its use is analogous to the application of the method of Friedman et al. [1975] as well as Nene and Nayar [1997] for vector spaces who eliminate a k-dimensional object $x = (x_0, x_1, \ldots, x_{k-1})$ from consideration as being within ϵ of $q = (q_0, q_1, \ldots, q_{k-1})$ if $|x_i - q_i| > \epsilon$ for one of x_i where $0 \leq i \leq k-1$. A variant of this method is also advocated by Chávez et al. [1999]. It is interesting to note that this search strategy makes an implicit assumption that the k-dimensional space is indexed with a set of inverted lists, one for the distance of the objects from each of the pivots. This is in contrast with a search strategy that assumes the existence of a more general multidimensional point access method (e.g., Gaede and Günther [1998] and Samet [1990, 1995]) on the k-dimensional space.

Ball partitioning methods (Section 5) are all examples of pivot-based methods. In particular, the fixed-queries tree [Baeza-Yates et al. 1994], fixed-height fixed-queries tree [Baeza-Yates et al. 1994], and fixed-queries array [Chávez et al. 2001a] methods (see Section 5.3) can be viewed as variations of embedding methods. In addition, methods that make use of distance matrices which contain precomputed distances between some or all of the objects in the data set such as AESA [Vidal Ruiz 1986; Wang and Shasha 1990] and LAESA [Micó et al. 1994] (Section 9) are also examples of pivot-based methods. Note that the distance matrix methods differ from the ball partitioning methods in that they do not form a hierarchical partitioning of the data set.

Clustering-based methods partition the underlying data set into spatial-like zones called *clusters* that are based on proximity to a distinguished object known as the *cluster center*. In particular, once a set of cluster centers has been chosen, the objects that are associated with each cluster center c are those that are closer to c than to any other cluster center. Although the cluster centers play a similar role as the pivot objects, the principal difference is that an object o is associated with a particular pivot p on the basis of the distance from o to p and not because p is the closest pivot to o, which would be the case if p was a cluster center. This means that in pivot-based methods an object o is not necessarily associated with the element whose pivot is closest to o. Generalized-hyperplane partitioning methods (Section 6) are examples of clustering-based methods. The *sa-tree* [Navarro 2002] (Section 8), inspired by the Voronoi diagram, is another example of a clustering-based method. It records a portion of the Delaunay graph of the data set, which is a graph whose vertices are the Voronoi cells, with

edges between adjacent cells. Although many of the clustering-based methods are hierarchical, this need not necessarily be the case.

It is interesting to observe that both pivot-based and clustering-based methods achieve a partitioning of the underlying data set into spatial-like zones. However, the difference is that the boundaries of the zones are more well defined in the case of pivot-based methods in the sense that they can be expressed explicitly using a small number of objects and a known distance value. In contrast, in the case of clustering-based methods, the boundaries of the zones are usually expressed implicitly in terms of the cluster centers, instead of explicitly, which may require quite a bit of computation to determine. In fact, very often, the boundaries cannot be expressed explicitly as, for example, in the case of an arbitrary metric space (in contrast to a Euclidean space) where we do not have a direct representation of the "generalized hyperplane" that separates the two partitions.

11. CONCLUDING REMARKS

We have surveyed a number of different methods for performing similarity search in metric spaces. The main focus was on distance-based indexing methods, with a short discussion of the alternative method of mapping into a vector space. We introduced a framework for performing search based on distances and presented algorithms for common types of queries. These algorithms can be applied to the indexing methods that we presented, given that a suitable search hierarchy is defined. We sketched such a hierarchy for several selected methods.

An important future task in this area is to develop cost models for the algorithms presented in Section 3 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. For the ranking query (Section 3.3), for example, there would be 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 kth 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, for example, for high-dimensional vector spaces [Berchtold et al. 1997] and for M-trees [Ciaccia et al. 1998]. 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 experiments using various distance-based indexes and mapping methods and 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) in a selected set of applications.

REFERENCES

ARYA, S., MOUNT, D. M., NETANYAHU, N. S., SILVERMAN, R., AND WU, A. 1994. An optimal algorithm for approximate nearest neighbor searching. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms* (Arlington, Va.). 573–582. (journal version: [Arya et al. 1998]).

ARYA, S., MOUNT, D. M., NETANYAHU, N. S., SILVERMAN, R., AND WU, A. Y. 1998. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM 45*, 6 (Nov.), 891–923.

BAEZA-YATES, R. A., CUNTO, W., MANBER, U., AND WU, S. 1994. Proximity matching using fixed-queries trees. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching* (Asilomar, Calif.). 198–212.

BECKMANN, N., KRIEGEL, H.-P., SCHNEIDER, R., AND SEEGER, B. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference* (Atlantic City, N.J.). ACM, New York, 322–331.

BENTLEY, J. L. 1975. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (Sept.), 509–517.

BERCHTOLD, S., BÖHM, C., KEIM, D. A., AND KRIEGEL, H.-P. 1997. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)* (Tucson, Az.). ACM, New York, 78–86.

BÖHM, C., BERCHTOLD, S., AND KEIM, D. A. 2001. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv. 33*, 3 (Sept.), 322–373.

BOZKAYA, T. AND OZSOYOGLU, M. 1997. Distance-based indexing for high-dimensional metric spaces. In *Proceedings of the ACM SIGMOD Conference* (Tucson, Az.), J. Peckham, Ed. ACM, New York, 357–368.

BOZKAYA, T. AND OZSOYOGLU, M. 1999. Indexing large metric spaces for similarity search queries. *ACM Trans. Datab. Syst. 24*, 3 (Sept.), 361–404.

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

BRINKHOFF, T., KRIEGEL, H.-P., AND SEEGER, B. 1993. Efficient processing of spatial joins using R-trees. In *Proceedings of the ACM SIGMOD Conference* (Washington, D.C.). ACM, New York, 237–246.

BUGNION, E., FEI, S., ROOS, T., WIDMAYER, P., AND WIDMER, F. 1993. A spatial index for approximate multiple string matching. In *Proceedings of the 1st South American Workshop on String Processing (WSP'93)* (Belo Horizonte, Brazil), R. Baeza-Yates and N. Ziviani, Eds. 43–53.

BURKHARD, W. A. AND KELLER, R. 1973. Some approaches to best-match file searching. *Commun. ACM 16*, 4 (Apr.), 230–236.

CAREY, M. J. AND KOSSMANN, D. 1997. On saying "enough already!" in SQL. In *Proceedings of the ACM SIGMOD Conference* (Tucson, Az.), J. Peckham, Ed., ACM, New York, 219–230.

CHAKRABARTI, K. AND MEHROTRA, S. 1998. High dimensional feature indexing using hybrid trees. Tech. Rep. TR-MARS-98-14. Department of Information and Computer Science, University of California, Irvine, Calif., July.

CHAKRABARTI, K. AND MEHROTRA, S. 1999. The hybrid tree: an index structure for high dimensional feature spaces. In *Proceedings of the 15th IEEE International Conference on Data Engineering*. (Sydney, Australia). IEEE Computer Society Press, Los Alamitos, Calif., 440–447.

CHÁVEZ, E., MARROQUÍN, J., AND BAEZA-YATES, R. 1999. Spaghettis: an array-based algorithm for similarity queries in metric spaces. In *Proceedings String Processing and Information Retrieval and International Workshop on Groupware (SPIRE/CRIWG 1999)* (Cancun, Mexico). 38–46.

CHÁVEZ, E., MARROQUÍN, J., AND NAVARRO, G. 2001a. Fixed queries array: A fast and economical data structure for proximity searching. *Multimed. Tools Appl. 14*, 2 (June), 113–135. (Expanded version of Overcoming the curse of dimensionality. In *European Workshop on Content-Based Multimedia Indexing*, pages 57–64, Toulouse, France, October 1999).

CHÁVEZ, E. AND NAVARRO, G. 2000. An effective clustering algorithm to index high dimensional spaces. In *Proceedings String Processing and Information Retrieval (SPIRE 2000)* (A Coruña, Spain). 75–86.

CHÁVEZ, E., NAVARRO, G., BAEZA-YATES, R., AND MARROQUÍN, J. 2001b. Searching in metric spaces. *ACM Comput. Surv. 33*, 3 (Sept.), 273–322. (Also Technical Report TR/DCC-99-3, DCC, University of Chile, Santiago, Chile, June 1999).

CHEN, J.-Y., BOUMAN, C. A., AND DALTON, J. C. 2000. Hierarchical browsing and search of large image databases. *IEEE Trans. Image Processing 9*, 3 (Mar.), 442–455.

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

CIACCIA, P. AND PATELLA, M. 1998. Bulk loading the M-tree. In *Proceedings of the 9th Australasian Database Conference (ADC'98)* (Perth, Australia). 15–26.

CIACCIA, P. AND PATELLA, M. 2002. Searching in metric spaces with user-defined and approximate distances. *ACM Trans. Datab. Syst. 27*, 4 (Dec.), 398–437.

CIACCIA, P., PATELLA, M., AND ZEZULA, P. 1997. 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)* (Athens, Greece), M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, Eds. 426–435.

CIACCIA, P., PATELLA, M., AND ZEZULA, P. 1998. A cost model for similarity queries in metric spaces. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)* (Seattle, Wash.). ACM, New York, 59–68.

DASGUPTA, S. 2002. Performance guarantees for hierarchical clustering. In *Proceedings of the 15th Annual Conference on Computational Learning Theory (COLT 2002)* (Sydney, Australia), J. Kivinen and R. H. Sloan, Eds. ACM, New York, 351–363.

DEHNE, F. AND NOLTEMEIER, H. 1987. Voronoi trees and clustering problems. *Inf. Syst. 12*, 2, 171–175.

DUNCAN, C. A., GOODRICH, M., AND KOBOUROV, S. 1999. Balanced aspect ratio trees: Combining the advantages of *k*-d trees and octrees. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms* (Baltimore, Md.). ACM, New York, 300–309.

FAGIN, R., LOTEM, A., AND NAOR, M. 2001. Optimal aggregation algorithms for middleware. In *Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)* (Santa Barbara, Calif.). ACM, New York, 102–113.

FALOUTSOS, C. AND LIN, K.-I. 1995. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proceedings of the ACM SIGMOD Conference* (San Jose, Calif.). ACM, New York, 163–174.

FOLEY, J. D., VAN DAM, A., FEINER, S. K., AND HUGHES, J. F. 1990. *Computer Graphics: Principles and Practice*, 2nd ed. Addison-Wesley, Reading, Mass.

FRIEDMAN, J. H., BASKETT, F., AND SHUSTEK, L. J. 1975. An algorithm for finding nearest neighbors. *IEEE Trans. Comput. 24*, 10 (Oct.), 1000–1006.

FRIEDMAN, J. H., BENTLEY, J. L., AND FINKEL, R. A. 1977. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw. 3*, 3 (Sept.), 209–226.

FU, A. W.-C., CHAN, P. M.-S., CHEUNG, Y.-L., AND MOON, Y. S. 2000. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances. *VLDB J. 9*, 2 (June), 154–173.

FUCHS, H., KEDEM, Z. M., AND NAYLOR, B. F. 1980. On visible surface generation by a priori tree structures. *Comput. Graph. 14*, 3 (July), 124–133. (Also *Proceedings of the SIGGRAPH'80 Conference*, Seattle, WA, July 1980.)

FUKUNAGA, K. AND NARENDRA, P. M. 1975. A branch and bound algorithm for computing *k*-nearest neighbors. *IEEE Trans. Comput. 24*, 7 (July), 750–753.

GAEDE, V. AND GÜNTHER, O. 1998. Multidimensional access methods. *ACM Comput. Surv. 20*, 2 (June), 170–231.

GENNARO, C., SAVINO, P., AND ZEZULA, P. 2001. Similarity search in metric databases through hashing. In *Proceedings of the 3rd International Workshop on Multimedia Information Retrieval (MIR'01)* (Ottawa, Ont., Canada). 1–5.

GONZALEZ, T. F. 1985. Clustering to minimize the maximum intercluster distance. *Theoret. Comput. Sci. 38*, 293–306.

GUTTMAN, A. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference* (Boston, Mass.). ACM, New York, 47–57.

HENRICH, A. 1994. A distance-scan algorithm for spatial access structures. In *Proceedings of the 2nd ACM Workshop on Geographic Information Systems* (Gaithersburg, Md.), N. Pissinou and K. Makki, Eds. ACM, New York, 136–143.

HJALTASON, G. R. AND SAMET, H. 1995. Ranking in spatial databases. In *Advances in Spatial Databases—4th International Symposium, SSD'95* (Portland, Me.), M. J. Egenhofer and J. R. Herring, Eds. 83–95.

HJALTASON, G. R. AND SAMET, H. 1998. Incremental distance join algorithms for spatial databases. In *Proceedings of the ACM SIGMOD Conference* (Seattle, Wash.), L. Hass and A. Tiwary, Eds. ACM, New York, 237–248.

HJALTASON, G. R. AND SAMET, H. 1999. Distance browsing in spatial databases. *ACM Trans. Datab. Syst. 24*, 2 (June), 265–318. (Also Computer Science TR-3919, University of Maryland, College Park, Md).

HJALTASON, G. R. AND SAMET, H. 2000. Incremental similarity search in multimedia databases. Computer Science Department TR-4199, Univ. Maryland, College Park, Md., Nov.

HJALTASON, G. R. AND SAMET, H. 2003a. Improved search heuristics for the SA-tree. *Patt. Rec. Lett. 24*, 15 (Nov.), 2785–2795.

HJALTASON, G. R. AND SAMET, H. 2003b. Properties of embedding methods for similarity searching in metric spaces. *IEEE Trans. Patt. Anal. Mach. Intell. 25*, 5 (May), 530–549. (Also University of Maryland Computer Science TR-4102.)

HRISTESCU, G. AND FARACH–COLTON, M. 1999. Cluster-preserving embedding of proteins. Tech. Rep., Rutgers University, Piscataway, N.J.

HUBBARD, P. M. 1996. Approximating polyhedra with spheres for time-critical collision detection. *ACM Trans. Graph. 15*, 3 (July), 179–210.

KALANTARI, I. AND MCDONALD, G. 1983. A data structure and an algorithm for the nearest point problem. *IEEE Trans. Softw. Eng. 9*, 5 (Sept.), 631–634.

KAMGAR-PARSI, B. AND KANAL, L. N. 1985. An improved branch and bound algorithm for computing *k*-nearest neighbors. *Patt. Rec. Lett. 3*, 1 (Jan.), 7–12.

KNUTH, D. E. 1998. *The Art of Computer Programming: Sorting and Searching*, 2nd ed. Vol. 3. Addison-Wesley, Reading, Mass.

KORN, F., SIDIROPOULOS, N., FALOUTSOS, C., SIEGEL, E., AND PROTOPAPAS, Z. 1996. 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. Mumbai (Bombay), India, 215–226.

LARSEN, S. AND KANAL, L. N. 1986. Analysis of k-nearest neighbor branch and bound rules. *Patt. Rec. Lett. 4*, 2 (Apr.), 71–77.

LINIAL, N., LONDON, E., AND RABINOVICH, Y. 1995. The geometry of graphs and some of its algorithmic applications. *Combinatorica 15*, 215–245.

MANEEWONGVATANA, S. AND MOUNT, D. M. 2001a. The analysis of a probabilistic approach to nearest neighbor searching. In *Algorithms and Data Structures—7th International Workshop, WADS 2001* (Providence, R.I.), F. K. H. A. Dehne, J. R. Sack, and R. Tamassia, Eds. 276–286.

MANEEWONGVATANA, S. AND MOUNT, D. M. 2001b. An empirical study of a new approach to nearest neighbor searching. In *Proceedigns of the 3rd Workshop on Algorithm Engineering and Experimentation (ALENEX'01)* (Washington, D.C.). 172–187.

MCNAMES, J. 1998. A nearest trajectory strategy for time series prediction. In *Proceedings of the International Workshop on Advanced Black-Box Techniques for Nonlinear Modeling*. Katholieke Universiteit Leuven, Leuven, Belgium, 112–128.

MCNAMES, J., SUYKENS, J. A. K., AND VANDERWALLE, J. 1999. Winning entry of the K. U. Leuven time series prediction competition. *Int. J. Bifur. Chaos 9*, 8 (Aug.), 1485–1500.

MERKWIRTH, C., PARLITZ, U., AND LAUTERBORN, W. 2000. Fast exact and approximate nearest neighbor searching for nonlinear signal processing. *Phys. Rev. E (Statistical Physics, Plasmas, Fluids, and Related Interdisciplinary Topics) 62*, 2 (Aug.), 2089–2097.

MICÓ, L., ONCINA, J., AND CARRASCO, R. 1996. A fast branch & bound nearest neighbor classifier in metric spaces. *Patt. Rec. Lett. 17*, 7 (June), 731–739.

MICÓ, L., ONCINA, J., AND VIDAL, E. 1992. 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. The Hague, The Netherlands, 557–560.

MICÓ, L., ONCINA, J., AND VIDAL, E. 1994. A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing-time and memory requirements. *Patt. Rec. Lett. 15*, 1 (Jan.), 9–17.

MOORE, A. 2000. The anchors hierarchy: Using the triangle inequality to survive high-dimensional data. In *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence* (San Francisco, Calif.). 397–405.

NAVARRO, G. 1999. Searching in metric spaces by spatial approximation. In *Proceedings String Processing and Information Retrieval and International Workshop on Groupware (SPIRE/CRIWG 1999)* (Cancun, Mexico). 141–148.

NAVARRO, G. 2002. Searching in metric spaces by spatial approximation. *VLDB J. 11*, 1, 28–46.

NAVARRO, G. AND REYES, N. 2002. Fully dynamic spatial approximation trees. In *String Processing and Information Retrieval—9th International Symposium (SPIRE 2002)* (Lisbon, Portugal), A. H. F. Laender and A. L. Oliveira, Eds. 254–270.

NENE, S. A. AND NAYAR, S. K. 1997. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Trans. Patt. Anal. Mach. Intel. 19*, 9 (Sept.), 989–1003.

NOLTEMEIER, H., VERBARG, K., AND ZIRKELBACH, C. 1992. Monotonous bisector* trees—a tool for efficient partitioning of complex scenes of geometric objects. In *Data Structures and Efficient Algorithms* (Berlin, West Germany). 186–211.

NOLTEMEIER, H., VERBARG, K., AND ZIRKELBACH, C. 1993. A data structure for representing and efficient querying large scenes of geometric objects: Mb*-trees. In *Geometric Modelling 1992* (Vienna, Austria), G. E. Farin, H. Hagen, H. Noltemeier, and W. Knödel, Eds. Springer-Verlag, New York, 211–226.

OMOHUNDRO, S. M. 1989. Five balltree construction algorithms. Tech. Rep. TR-89-063, International Computer Science Institute, Berkeley, Calif. Dec.

OOI, B. C., MCDONELL, K. J., AND SACKS-DAVIS, R. 1987. Spatial *k-d*-tree: an indexing mechanism for spatial database. In *Proceedings of the 11th International Computer Software and Applications Conference (COMPSAC)* (Tokyo, Japan). 433–438.

ORENSTEIN, J. A. 1982. Multidimensional tries used for associative searching. *Inf. Proc. Lett. 14*, 4 (June), 150–157.

RAMASUBRAMANIAN, V. AND PALIWAL, K. K. 1992. An efficient approximation-elimination algorithm for fast nearest neighbour search based on a spherical distance coordinate formulation. *Patt. Rec. Lett. 13*, 7 (July), 471–480.

ROUSSOPOULOS, N., KELLEY, S., AND VINCENT, F. 1995. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD Conference* (San Jose, Calif.). ACM, New York, 71–79.

RUSSEL, S. AND NORVIG, P. 1994. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Englewood Cliffs, N.J.

SAMET, H. 1990. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, Mass.

SAMET, H. 1995. Spatial data structures. In *Modern Database Systems, The Object Model, Interoperability and Beyond*, W. Kim, Ed. ACM Press and Addison-Wesley, New York, 361–385.

SAMET, H. 2004. *Foundations of Multidimensional Data Structures*. To appear.

SEIDL, T. AND KRIEGEL, H.-P. 1998. Optimal multi-step *k*-nearest neighbor search. In *Proceedings of the ACM SIGMOD Conference* (Seattle, Wash.), L. Hass and A. Tiwary, Eds. ACM, New York, 154–165.

SHAFER, J. C. AND AGRAWAL, R. 1997. Parallel algorithms for high-dimensional proximity joins. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)* (Athens, Greece), M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, Eds. 176–185.

SHAPIRO, M. 1977. The choice of reference points in best-match file searching. *Commun. ACM 20*, 5 (May), 339–343.

SHIN, H., MOON, B., AND LEE, S. 2000. Adaptive multi-stage distance join processing. In *Proceedings of the ACM SIGMOD Conference* (Dallas, Tex.), W. Chen, J. Naughton, and P. A. Bernstein, Eds. ACM, New York, 343–354.

SMITH, T. F. AND WATERMAN, M. S. 1981. Identification of common molecular subsequences. *J. Mol. Biol. 147*, 1, 195–197.

TRAINA JR., C., TRAINA, A. J. M., FALOUTSOS, C., AND SEEGER, B. 2002. Fast indexing and visualization of metric data sets using slim-trees. *IEEE Trans. Knowl. Data Eng. 14*, 2 (Mar./Apr.), 244–260.

UHLMANN, J. K. 1991a. Metric trees. *Appl. Math. Lett. 4*, 5, 61–62.

UHLMANN, J. K. 1991b. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett. 40*, 4 (Nov.), 175–179.

VAN OOSTEROM, P. 1990. Reactive data structures for geographic information systems. Ph.D. dissertation, Department of Computer Science, Leiden University, Leiden, The Netherlands.

VAN OOSTEROM, P. AND CLAASSEN, E. 1990. Orientation insensitive indexing methods for geometric objects. In *Proceedings of the 4th International Symposium on Spatial Data Handling*. Vol. 2. Zurich, Switzerland, 1016–1029.

VIDAL, E. 1994. New formulation and improvements of the nearest-neighbour approximating and eliminating search algorithm (AESA). *Patt. Rec. Lett. 15*, 1 (Jan.), 1–7.

VIDAL RUIZ, E. 1986. An algorithm for finding nearest neighbours in (approximately) constant average time. *Patt. Rec. Lett. 4*, 3 (July), 145–157.

VILAR, J. M. 1995. Reducing the overhead of the AESA metric-space nearest neighbour searching algorithm. *Inf. Process. Lett. 56*, 5 (Dec.), 265–271.

VORONOI, G. 1909. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxiême mémoire: Recherches sur les parallèlloèdres primitifs. seconde partie. *J. Reine Angew. Math. 136*, 2, 67–181.

WANG, J. T.-L., WANG, X., LIN, K.-I., SHASHA, D., SHAPIRO, B. A., AND ZHANG, K. 1999. Evaluating a class of distance-mapping algorithms for data mining and clustering. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Diego, Calif.). ACM, New York, 307–311.

WANG, T. L. AND SHASHA, D. 1990. Query processing for distance metrics. In *Proceedings of the 16th International Conference on Very Large Databases (VLDB)* (Brisbane, Australia), D. McLeod, R. Sacks-Davis, and H.-J. Schek, Eds. 602–613.

WHITE, D. A. AND JAIN, R. 1996. Similarity indexing with the SS-tree. In *Proceedings of the 12th IEEE International Conference on Data Engineering* (New Orleans, La.), S. Y. W. Su, Ed. IEEE Computer Society Press, Los Alamitos, Calif., 516–523.

YIANILOS, P. N. 1993. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms* (Austin, Tex.). ACM, New York, 311–321.

YIANILOS, P. N. 1998. Excluded middle vantage point forests for nearest neighbor search. Tech. rep., NEC Research Institute, Princeton, NJ. July. (Presented at the *First Workshop on Algorithm Engineering and Experimentation (ALENEX'99)*, Baltimore, MD, January 1999.)

Received January 2003; accepted July 2003