Indexing Large Metric Spaces for Similarity Search Queries

TOLGA BOZKAYA
Oracle Corporation
and
MERAL OZSOYOGLU
Case Western Reserve University

One of the common queries in many database applications is finding approximate matches to a given query item from a collection of data items. For example, given an image database, one may want to retrieve all images that are similar to a given query image. Distance-based index structures are proposed for applications where the distance computations between objects of the data domain are expensive (such as high-dimensional data) and the distance function is metric. In this paper we consider using distance-based index structures for similarity queries on large metric spaces. We elaborate on the approach that uses reference points (vantage points) to partition the data space into spherical shell-like regions in a hierarchical manner. We introduce the multivantage point tree structure (mvp-tree) that uses more than one vantage point to partition the space into spherical cuts at each level. In answering similarity-based queries, the mvp-tree also utilizes the precomputed (at construction time) distances between the data points and the vantage points.

We summarize the experiments comparing mvp-trees to vp-trees that have a similar partitioning strategy, but use only one vantage point at each level and do not make use of the precomputed distances. Empirical studies show that the mvp-tree outperforms the vp-tree by 20% to 80% for varying query ranges and different distance distributions. Next, we generalize the idea of using multiple vantage points and discuss the results of experiments we have made to see how varying the number of vantage points in a node affects search performance and how much is gained in performance by making use of precomputed distances. The results show that, after all, it may be best to use a large number of vantage points in an internal node in order to end up with a single directory node and keep as many of the precomputed distances as possible to provide more efficient filtering during search operations. Finally, we provide some experimental results that compare mvp-trees with M-trees, which is a dynamic distance-based index structure for metric domains.

A preliminary version of this paper appeared in Proceedings of the 1997 ACM-SIGMOD [Bozkaya and Ozsoyoglu 1997].
This research is partially supported by the National Science Foundation grant IRI 92-24660, and the National Science Foundation FAW award IRI-90-24152.
Authors' addresses: T. Bozkaya, Oracle Corporation, 500 Oracle Parkway, Redwood Shores, CA 94065; email: tbozkaya@us.oracle.com; M. Ozsoyoglu, Department of Computer Engineering and Science, Case Western Reserve University, Cleveland, OH 44106; email: ozsoy@ces.cwru.edu.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*Trees*; H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*

General Terms: Algorithms, Experimentation, Measurement, Performance, Verification

1. INTRODUCTION

In many database applications it is desirable to answer queries based on proximity, such as asking for data items that are similar to a query item, or that are closest to a query item. We face such queries in the context of many database applications such as genetics, text matching, image/picture databases, time-series analysis, information retrieval, and so on. In genetics, the goal is to find DNA or protein sequences that are similar in a genetic database. In time-series analysis, we would like to find similar patterns among a given collection of sequences. Image databases can be queried to find and retrieve images in the database that are similar to the query image with respect to specified criteria.

Similarity between images can be measured in a number of ways. Features such as shape, color, and texture can be extracted from images in the database for use as content information where the distance calculations are based on them. Images can also be compared on a pixel by pixel basis by calculating the distance between two images as the accumulation of the differences between the intensities of their pixels.

In all the applications above, the problem is to find data items similar to a given query item where the similarity between items is computed by some distance function defined on the application domain. Our objective is to provide an efficient access mechanism to answer these similarity queries. In this paper we consider the applications where the distance function employed is *metric* and computation of distances are expensive. It is important for an application to have a metric distance function to filter distant data items for a similarity query by using the *triangle inequality* property (Section 2). As the distance computations are assumed to be expensive, an efficient access mechanism should certainly minimize the number of distance calculations for similarity queries and improve the speed in answering them. This is usually done by employing techniques and index structures to filter out distant (nonsimilar) data items quickly, avoiding expensive distance computations for each of them.

Data items that are in the result of a similarity query can be further filtered out by the user through visual browsing. This happens in image database applications where the user picks those semantically related images that are most similar to a query image by examining the images retrieved in the result of a similarity query. This is mostly inevitable because it is impossible to extract and represent all the semantic information for an image by simply extracting features in the image. The best an image database can do is present the images that are related or close to the

query image and leave further identification and semantic interpretation of images to users.

In this paper, the number of distance computations required in a similarity search query is taken as the efficiency measure. We do not incorporate the I/O operations required during the evaluation of queries into the cost measure. This can be justified in part, since our target applications are the ones where the distance computations are very expensive. In such applications, the distance-computation measure, to a degree, also reflects the I/O costs (or other costs, such as network costs) because a distance computation requires retrieving a database object from secondary memory (though it does not reflect the I/O operations required by the index structure). As an example, consider a www site with an index on a large number of pages on some other www sites. For a similarity query, the cost of searching the index for a similarity query is directly related to the number of www pages retrieved during distance computations, making the I/O costs at the index site negligible. However, the role of I/O costs should be incorporated in general, and comparisons of mvp-tree performance with other access structures, for the general case where I/O costs can not be neglected, remains as future research.

We introduce the mvp-tree (multivantage point tree) as a general solution to the problem of efficiently answering similarity-based queries for high-dimensional metric spaces. The mvp-tree is similar to the vp-tree (vantage point tree) [Uhlmann 1991], in the sense that both structures use relative distances from a vantage point to partition the domain space. At every node of the vp-tree, a vantage point is chosen among the data points, and the distances of this vantage point from all other points (the points that will be indexed below that node) are computed. These points are then sorted into an ordered list with respect to their distances from the vantage point. Next, the list is partitioned to create sublists of equal cardinality. The order of the tree corresponds to the number of partitions made. Each of these partitions keeps the data points that fall into a spherical cut, with inner and outer radii being the minimum and the maximum distances of these points from the vantage point.

The mvp-tree behaves more cleverly in making use of the vantage-points by employing more than one at each level of the tree to increase the fanout of each node of the tree. In vp-trees, for a given similarity query, most of the distance computations are between the query point and the vantage points. Because it uses more than one vantage point in a node, the mvp-tree has fewer vantage points compared to a vp-tree. The distances of data points at the leaf nodes from the vantage points at higher levels (which were already computed at construction time) are kept in mvp-trees, and these distances are used for efficient filtering at search time. More efficient filtering at the leaf level is utilized by making the leaf nodes have higher node capacities. This way, the major filtering step during the search is delayed to the leaf level.

We present experiments with high-dimensional Euclidean vectors and gray-level images to compare vp-trees to mvp-trees to demonstrate the

efficiency of mvp-trees. In these experiments we use an mvp-tree that has two vantage points in a node. The distance distribution of data points plays an important role in the efficiency of the index structures; so we experimented with different sets of Euclidean vectors with different distance distributions. Our experiments with Euclidean vectors show that mvp-trees require 40% to 80% fewer distance computations compared to vp-trees for small query ranges. For higher query ranges, the percentage-wise difference decreases gradually, yet mvp-trees still perform better, making up to 30% fewer distance computations for the largest query ranges.

Our experiments on gray-level images using L_1 and L_2 metrics (see Section 5.1) also reveal the fact that mvp-trees perform better than vp-trees. For this data set we had only 1151 images to experiment with (and so had rather shallow trees), and the mvp-trees performed up to 20-30% fewer distance computations.

We explore the issue of choosing better vantage points, preferably without introducing too much overhead into the construction step. We test a simple heuristic that chooses points that are far away from most of the data points for Euclidean vectors, and compare it to the results where the vantage points are chosen randomly.

We generalize the mvp-tree structure so that it can use any number of vantage points in an internal node and conduct experiments to see how using more than one vantage point in a node scales up. In these experiments, Euclidean vectors are used to observe and compare the performance of mvp-trees with more than two vantage points in a node. In the ultimate case, all the vantage points are kept in a single directory node, creating a two-level tree structure (one internal node and the leaves), where only the vantage points in this single directory node are used hierarchically to partition the whole data space. Interestingly, the two-level mvp-tree that keeps all vantage points in a single directory is the most efficient structure in terms of minimizing the number of distance computations in answering similarity queries for high-dimensional Euclidean vectors we used in our experiments. As a final step, we compare the generalized mvp-trees with M-trees, which is one of the state-of-the-art index structures for metric spaces.

The rest of the paper is organized as follows. Section 2 gives the definitions for metric spaces and similarity queries. Section 3 presents the problem of indexing in large spaces and also previous approaches to this problem. The related work on distance-based index structures is also given in Section 3. Section 4 introduces the mvp-tree structure. The experimental results for comparing the mvp-trees with vp-trees are given in Section 5. Section 6 elaborates on how to choose better vantage points. Section 7 explains how the mvp-tree structure can be generalized so that more than two vantage points can be kept in any node. Section 8 presents the experimental results for the generalized version of mvp-trees with different numbers of vantage points, and the results of experiments conducted for comparing mvp-trees to M-trees. The conclusions are given in Section 9.

2. METRIC SPACES AND SIMILARITY QUERIES

In this section we briefly give the definitions for metric distance functions and different types of similarity queries.

A metric distance function *d(x,y)* for a metric space is defined as follows:

(i) $d(x, y) = d(y, x)$
(ii) $0 < d(x, y) < \infty, x \neq y$
(iii) $d(x, x) = 0$
(iv) $d(x, y) \leq d(x, z) + d(z, y)$ (triangle inequality)

The above conditions are the only ones we can assume when designing an index structure based on distances between objects in a metric space. No geometric information can be utilized for a metric space, unlike the case for a Euclidean space. Thus, we only have a set of objects from a metric space and a distance function $d()$ that can be used to compute the distance between any two objects.

Similarity-based queries can be posed in a number of ways. The most common type asks for all data objects within some specified distance from a given query object. These queries require retrieval of the *near neighbors* of the query object. The formal definition for this type of query is as follows:

Near-neighbor query. From a given set of data objects $X = \{X_1, X_2, \ldots, X_n\}$ from a metric space with a metric distance function $d()$, retrieve all data objects that are within distance r of a given query point Y. The resulting set is $\{X_i \mid X_i \in X \text{ and } d(X_i, Y) \leq r\}$. Here, r is generally referred to as the similarity measure, or the tolerance factor.

Some variations of the near-neighbor query are also possible. The *nearest-neighbor query* asks for the object (or objects) that has the minimum distance to a given query object. Similarly, k-closest objects may be requested as well. Though not very common, objects that are farther than a given range from a query object can also be asked, as well as the farthest or the k-farthest objects from the query object. The formulations for all these queries are similar to the formulation of the near-neighbor query given above.

Here we are mainly concerned about distance-based indexing for large metric spaces. We also concentrate on the near-neighbor queries when we introduce our index structure. Our main objective is to minimize the number of distance calculations for a given similarity query, as the distance computations are assumed to be very expensive for the applications we target. In the next section we discuss the indexing problem for large metric spaces and review previous approaches to the problem.

3. INDEXING IN LARGE METRIC SPACES

The problem of indexing large metric spaces can be approached in different ways. One approach is to use distance transformations to Euclidean spaces, which is discussed in Section 3.1. Another is to use distance-based index structures. In Section 3.2, we discuss distance-based index structures and

briefly review the previous work. In Section 3.3, vp-tree structure is discussed in more detail.

3.1 Distance Transformations to Euclidean Spaces

For low-dimensional Euclidean domains, the conventional index structures [Samet 1989] such as R-trees (and its variations) [Guttman 1984 ; Sellis et al. 1987; Beckmann et al. 1990] can be used effectively to answer similarity queries. In such cases, a near-neighbor search query asks for all the objects in (or that intersect) a spherical search window where the center is the query object and the radius is the tolerance factor r. There are some special techniques for other forms of similarity queries, such as nearest-neighbor queries. For example, in Roussopoulos et al. [1995], some heuristics are introduced to efficiently search the R-tree structure to answer nearest-neighbor queries. However, the conventional spatial structures stop being efficient if the dimensionality is high. Experimental results [Otterman 1992] show that R-trees become inefficient for n-dimensional spaces where n is greater than 20.

It is possible to make use of conventional spatial index structures for some high-dimensional Euclidean domains. One way is to apply a mapping of objects from the original high-dimensional space to a low-dimensional (Euclidean) space by using a distance transformation and then conventional index structures (such as R-trees) as a major filtering mechanism in the transformed space. For a distance transformation from a high-dimensional domain to a lower-dimensional domain to be effective, the distances between objects before the transformation (in the original space) should be greater than or equal to the distances after the transformation (in the transformed space); otherwise the transformation may impose some false dismissals during similarity search queries. That is, the distance transformation function should underestimate the actual distances between objects in the transformed space. For efficiency, the distances in the transformed space should be close estimates of the distances in the actual space. Such transformations have been successfully used to index high-dimensional data in many applications such as time sequences [Agrawal et al. 1993; Faloutsos et al. 1994a], and images [Faloutsos et al. 1994b].

Although it is possible to make use of general transformations such as DFT, Karhunen-Loeve for any Euclidean domain, it is also possible to come up with application-specific distance transformations. In the QBIC (Query By Image Content) system [Faloutsos et al. 1994b], the color content of images was used to compute similarity between images. The differences between the color content of two images are computed from their color histograms. Computation of a distance between the color histograms of two images is quite expensive because the color histograms are high-dimensional (the number of different colors is generally 64 or 256) vectors, and *crosstalk* (as some colors are similar) between colors have to be considered. To increase speed in color distance computation, QBIC keeps an index on the average color of images. The average color of an image is a three-

dimensional vector with the average red, blue, and green values of the pixels in the image. The distance between average color vectors of images is proven to be less than or equal to the distance between their color histograms; that is, the transformation underestimates the actual distances. Similarity queries on the color content of images are answered by first using the index on average color vectors as the major filtering step and then refining the result by actual computations of histogram distances.

Note that, although the idea of using a distance transformation works fine for many applications, it makes the assumption that such a transformation exists and is applicable to the domain of interest. Transformations such as DFT or Karhunen-Loeve are not effective in indexing high-dimensional vectors where the values of dimensions are uncorrelated in any given vector. Therefore, unfortunately, it is not always possible or cost effective to employ a distance transformation. Yet, there are distance-based indexing techniques that are applicable to all domains where metric distance functions are employed. These techniques can be used directly for high-dimensional spatial domains, since the conventional distance functions (such as Euclidean, or any L_p distance) defined on these domains are metric. Sequence matching, time-series analysis, and image databases are some example applications having such domains. Distance-based techniques are also applicable for domains where the data is nonspatial (that is, data objects can not be mapped to points in a multidimensional space), such as text databases which generally use the *edit distance* (which is metric) for computing similarity data items (lines of text, words, etc.). We review a few of the distance-based indexing techniques in Section 3.2.

3.2 Distance-Based Index Structures

There are a number of research results for efficiently answering similarity search queries in different contexts. Burkhard and Keller [1973] suggest the use of three different techniques for finding best-matching (closest) key words in a file to a given query key. They employ a metric distance function on the key space, which always returns discrete values (i.e., the distances are always integers). Their first method is a hierarchical multiway tree decomposition. At the top level, they pick an arbitrary element from the key domain and group the rest of the keys with respect to their distances to that key. Keys that are of the same distance from that key get into the same group. Note that this is possible because the distance values are always discrete. The same hierarchical composition goes on for all the groups recursively, creating a tree structure.

The second method in Burkhard and Keller [1973] partitions the data space into a number of sets of keys. For each set, a *center* key is picked arbitrarily, and the *radius*, which is the maximum distance between the *center* and any other key in the set, is calculated. The keys in each set are partitioned in the same way recursively, creating a multiway tree. Each node in the tree keeps the *centers* and the *radii* for the sets of keys indexed

below. The strategy for partitioning the keys into sets was not discussed and was left as a parameter.

The third method in Burkhard and Keller [1973] is similar to the second one, but there is the requirement that the *diameter* (the maximum distance between any two points in a group) of any group is less than a given constant k, where the value of k is different at each level. The group satisfying this criterion is called a *clique*. This method relies on finding the set of maximal cliques at each level and keeping their representatives in the nodes to direct or trim the search. Note that keys may appear in more than one clique; so the aim is to select as representative keys the ones that appear in as many *cliques* as possible.

In another approach, Shasha and Wang [1990] suggest using precomputed distances between data elements to efficiently answer similarity search queries. The aim is to minimize the number of distance computations as much as possible, since they are assumed to be very expensive. Search algorithms of $O(n)$ or even $O(n\log n)$ (where n is the number of data objects) are acceptable if they minimize the number of distance computations. In Shasha and Wang's method [Shasha and Wang 1990], a table of size $O(n^2)$ keeps the distances between data objects if they are precomputed. The other pairwise distances are estimated (by specifying an interval) by making use of the precomputed distances. The technique of storing and using precomputed distances may be effective for data domains with small cardinality; however, space requirements and search complexity become overwhelming for larger domains.

Uhlmann [1991] introduced two hierarchical index structures for similarity search. The first is the vp-tree (*vantage-point tree*). The vp-tree basically partitions the data space into spherical cuts around a chosen *vantage point* at each level. This approach, referred to as *ball decomposition* in the paper, is similar to the first method in Burkhard and Keller [1973]. At each node the distances between the vantage point for that node and the data points to be indexed below that node are computed. The median is found and the data points are partitioned into two groups, one of them accommodating the points whose distances to the vantage point are less than or equal to the median distance, and the other group accommodating the points whose distances are larger than or equal to the median. These two groups of data points are indexed separately by the left and right subbranches below that node, which are constructed in the same way recursively.

Although the vp-tree was introduced as a binary tree, it is also possible to generalize it to a multiway tree for larger fanouts. Yiannilos [1993] provided some analytical results on vp-trees and suggested ways to pick better vantage points. Chiueh [1994] proposed an algorithm for the vp-tree structure to answer nearest-neighbor queries. We talk about vp-trees in detail in Section 3.3.

The gh-tree (a *generalized hyperplane tree)* structure was also introduced in Uhlmann [1991]. A gh-tree is constructed as follows. At the top level, two points are picked and the remaining points are divided into two groups,

depending on which of these two points they are closer to. This partitioning descends down recursively to create a tree structure. Unlike the vp-trees, the branching factor can only be two. If the two *pivot* points are well-selected at every level, the gh-tree tends to be a well-balanced structure.

The FQ-tree (*fixed queries tree*) is another tree structure that uses the idea of partitioning the data space around reference points [Baeza-Yates et al. 1994]. The main difference from the vp-tree is that the FQ-tree uses the same reference point for all internal nodes at the same level. So the total number of reference points (vantage points) used is equal to the height of the tree. The partitioning in FQ-trees is similar to the first approach in Burkhard and Keller [1973]. A discrete (or discretized) distance function is assumed, and the data space is partitioned with respect to every possible distance value from the reference point. A performance analysis of FQ-trees is also given in Baeza-Yates et al. [1994]. The idea of using a single reference point for all nodes in the same level is an interesting one. We use a similar technique in the design of mvp-trees.

The GNAT (*geometric near-neighbor access tree*) structure [Brin 1995] is another mechanism for answering near-neighbor queries. A k number of *split points* are chosen at the top level. Each one of the remaining points is associated with one of the k data sets (one for each *split point*), depending on which *split point* they are closest to. For each *split point*, the minimum and the maximum distances from the points in the data sets of other *split points* are recorded. The tree is built recursively for each data set at the next level. The number of *split points*, k, is parameterized and chosen to be a different value for each data set, depending on its cardinality. The GNAT structure is compared to the binary vp-tree, and it is shown that the preprocessing (construction) step of GNAT is more expensive than the vp-tree, but its search algorithm makes less distance computations in the experiments for different data sets.

More recently, Ciaccia et al. [1997] introduced the M-tree structure, which differs from the other distance-based index structures in being able to handle dynamic operations. The M-tree is constructed bottom-up (in contrast to the other structures such as the vp-tree, GNAT, and the gh-tree, which are constructed top-down), and it can handle dynamic operations with reasonable cost and without requiring periodical restructuring. An M-tree stores a given set of objects $\{o_1, \ldots, o_2\}$ into fixed-size leaf nodes, which correspond to sphere-like regions of the metric space. Each leaf node entry contains the id of a data object, its feature values (used in a distance computation), and its distance from a routing object, which is kept at the parent node. Each internal node entry keeps a child pointer, a routing object, and its distance from its parent routing object (except for the root), and the radius of the sphere-like region that accommodates all the objects indexed below that entry (called the covering radius). The search is pruned by making use of the covering radii and the distances from objects to their routing objects in their parent nodes. Experimental results for M-trees are provided in Ciaccia et al. [1997]; Ciaccia and Patella [1998]; Ciaccia et al.

[1998a; 1998b]. An analytical cost model based on distance distribution of the objects is derived in Ciaccia et al. [1998b] for M-trees. Evaluation of complex similarity queries (with multiple similarity predicates) using M-trees is discussed in Ciaccia et al. [1998a]. Ciaccia and Patella [1998] provide an algorithm for creating an M-tree from a given set of objects via *bulkloading*. We provide some experimental results with M-trees in Section 8.2.

3.3 Vantage Point-Tree Structure

Let us briefly discuss the vp-tree to explain the idea of partitioning the data space around selected points (vantage points) at different levels to form a hierarchical tree structure and using it for effective filtering in similarity search queries.

The structure of a binary vp-tree is very simple. Each internal node is of the form $(S_v, M, R_{ptr}, L_{ptr})$, where S_v is the vantage point, M is the median distance among the distances of all the points (from S_v) indexed below that node, and R_{ptr} and L_{ptr} are pointers to the left and right branches. The left branch of the node indexes the points whose distances from S_v are less than or equal to M, and the right branch of the node indexes the points whose distances from S_v are greater than or equal to M. References to the data points are kept in leaf nodes, instead of pointers to the left and right branches.

Given a finite set $S = \{S_1, S_2, .., S_n\}$ of n objects and a metric distance function $d(S_i, S_j)$, a binary vp-tree V on S is constructed as follows.

(1) If $|S| = 0$, then create an empty tree.
(2) Else, let S_v be an arbitrary object from S. (S_v is the vantage point)
$M =$ median of $\{d(S_i, S_v) \mid S_i \in S\}$
Let $S_l = \{S_i \mid d(S_i, S_v) \leq M,$ where $S_i \in S$ and $S_i \neq S_v\}$
$S_r = \{S_j \mid d(S_j, S_v) \geq M,$ where $S_j \in S\}$
(The cardinality of S_l and S_r should be equal)
Recursively create vp-trees on S_l and on S_r as the left and right branches of the root of V.

The binary vp-tree is balanced, and can therefore be easily paged for storage in secondary memory. The construction step requires $O(n \log_2 n)$ distance computations, where n is the number of objects.

For a given query object Q, the set of data objects that are within distance r of Q is found using the search algorithm given below.

(1) If $d(Q, S_v) \leq r$, then S_v (the vantage point at the root) is in the answer set.
(2) If $d(Q, S_v) + r \geq M$ (median), then recursively search the right branch.
(3) If $d(Q, S_v) - r \leq M$, then recursively search the left branch.
(note that both branches can be searched if both search conditions are satisfied.)

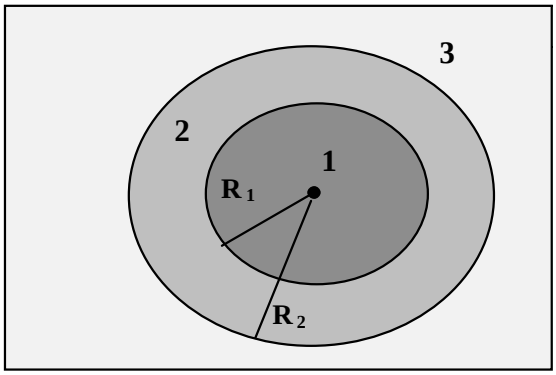

Fig. 1. The root level partitioning of a vp-tree with branching factor 3. The three different regions are labeled 1, 2, 3, and they are all shaded differently.

The correctness of this simple search strategy can be proven easily by using the *triangle inequality* of distances among any three objects in a metric data space (see Appendix).

Generalizing binary vp-trees into multiway vp-trees. The binary vp-tree can be easily generalized into a multiway tree structure for larger fanouts at every node, hoping that the decrease in the height of the tree would also decrease the number of distance computations. The construction of a vp-tree of order m is very similar to that of a binary vp-tree. Instead of finding the median of the distances between the vantage point and the data points, the points are ordered with respect to their distances from the vantage point and partitioned into m groups of equal cardinality. The distance values used to partition the data points are recorded in each node. We refer to those values as *cutoff* values. There are $m - 1$ cutoff values in a node. The m groups of data points are indexed below the root node by its m children, which are themselves vp-trees of order m created in the same way recursively. The construction of an m-way vp-tree requires $O(n \log_m n)$ distance computations. That is, creating an m-way vp-tree decreases the number of distance computations by a factor of $\log_2 m$ at the construction stage compared to binary vp-trees.

However, there is one problem with high-order vp-trees. A vp-tree partitions the data space into spherical cuts (see Figure 1). These spherical cuts become too thin for high-dimensional domains, leading the search regions to intersect with many of them, and thus to more branching during a similarity search. As an example, consider an N-dimensional Euclidean space where N is a large number, and a vp-tree of order three is built to index the uniformly distributed data points in that space. At the root level, the N-dimensional space is partitioned into three spherical regions, as shown in Figure 1. The three different regions are colored differently and labeled 1, 2, and 3. Let R_1 be the radius of region 1 and R_2 be the radius of the sphere enclosing regions 1 and 2. Due to the uniform distribution

assumption, the N-dimensional volumes of regions 1 and 2 can be considered equal. The volume of an N-dimensional sphere is directly proportional to the N^{th} factor of its radius, so we can deduce that $R_2 = R_1*(2)^{1/N}$. The thickness of the spherical shell of region 2 is $R_2 - R_1 = R_1*(2^{1/N} - 1)$. To give an idea, for $N = 100$, $R_2 = 1.007R_1$.

So when the spherical cuts are very thin, the chances of a search operation descending down to more than one branch becomes higher. If a search path descends down to k out of m children of a node, then k distance computations are needed at the next level, where the distance between the query point and the vantage point of each child node has to be found. This is because the vp-tree keeps a different vantage point for each node at the same level. Each child of a node is associated with a region that is like a spherical shell (other than the innermost child, which has a spherical region), and the data points indexed below that child node all belong to that region. Those regions are disjoint for the siblings. Since the vantage point for a node has to be chosen among the data points indexed below a node, the vantage points of the siblings are all different.

4. MULTIVANTAGE-POINT TREES

In this section we present the mvp-tree (multivantage point tree). Similar to the vp-tree, the mvp-tree partitions the data space into spherical cuts around vantage points. However, it creates partitions with respect to more than one vantage point at each level and keeps extra information in the leaf nodes for effective filtering of distant points in a similarity search operation.

4.1 Motivation

Before introducing the mvp-tree, we discuss a few useful observations that can be used as heuristics for designing a better search structure using vantage points.

Observation 1. It is possible to partition a spherical shell-like region using a vantage point chosen from outside the region. This is shown in Figure 2, where a vantage point outside of a region is used to partition it into three parts, which are labeled as 1, 2, 3 and shaded differently (region 2 consists of two disjoint parts).

This means that the same vantage point can be used to partition the regions associated with the nodes at the same level. When the search operation descends down to several branches, we do not have to make a different distance computation at the root of each branch. Also, if the same vantage point can be used for all the children of a node, that vantage point can be kept in the parent just as well. This way we would keep more than one vantage point in the parent node. We can avoid creating the children nodes by incorporating them in the parent. This could be done by increasing the fanout of the parent node. The mvp-tree takes this approach and uses more than one vantage point in the nodes for higher utilization.

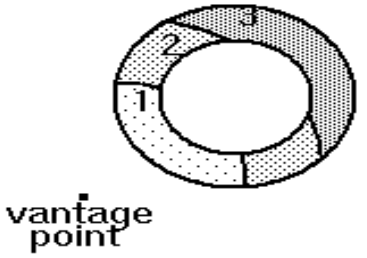

Fig. 2. Partitioning a spherical shell-like region using a vantage point from outside.

Observation 2. In the construction of the vp-tree structure, for each data point in the leaves, we compute the distances between that point and all the vantage points on the path from the root node to the leaf node that keeps that data point. So for each data point, $(\log_m(n))$ distance computations (for an vp-tree of order m) are made (which is equal to the height of the tree). In vp-trees, such distances (other than the distance to the vantage point of the leaf node) are not kept. However, *it is possible to keep these distances for the data points in the leaf nodes to provide further filtering at the leaf level during search operations.* We use this idea in mvp-trees. In mvp-trees, for each data point in a leaf, we also keep the first p distances (here, p is a parameter) that are computed in the construction step between that data point and the vantage points at the upper levels of the tree. The search algorithm is modified to make use of these distances.

Figure 3 illustrates how the precomputed distances could be helpful in filtering distant objects. In this figure, a shallow vp-tree is shown with two internal nodes having vantage points $vp1$ and $vp2$, and a leaf node with data point $p1$. Consider a near-neighbor query where the query point is Q and the similarity range is r, as depicted in Figure 3. The data space is partitioned into two regions with respect to vp1 where the boundary is shown with the bold circle around $vp1$. The outer region is partitioned using $vp2$. The similarity search proceeds down to the leaf node where $p1$ is kept. By considering only the distance between $p1$ and $vp2$ (which is the way done in vp-trees), we would not be able to filter out $p1$, and therefore would have to compute $d(Q, p1)$ (see inequality (1)). However, if the distance $d(vp1, p1)$ (which is computed at construction time) is also considered, then $p1$ can be filtered out due to inequality (2).

Having shown the motivation behind the mvp-tree structure, we explain the construction and search algorithms below.

4.2 Mvp-Tree Structure

The mvp-tree uses two vantage points in every node. Each node of the mvp-tree can be viewed as two levels of a vantage point tree (a parent node

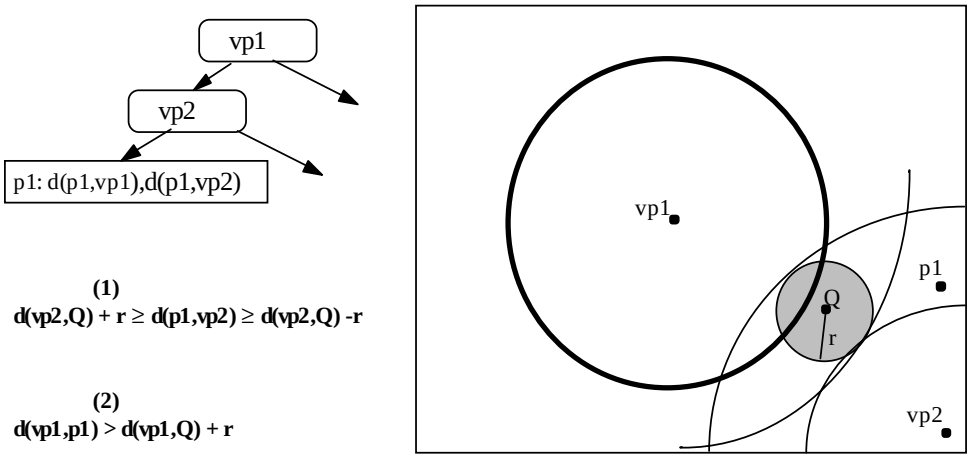

Fig. 3. A vp-tree decomposition where the use of precomputed (at construction time) distances make it possible to filter out a distant point ($p1$).

and all its children) where all the children nodes at the lower level use the same vantage point. This makes it possible for an mvp-tree node to have large fanouts and a smaller number of vantage points in nonleaf levels.

In this section we show the structure of mvp-trees and present the construction algorithm for binary mvp-trees. In general, an mvp-tree has three parameters:

- the number of partitions created by each vantage point (m);
- the maximum fanout for the leaf nodes (k); and
- the number of distances for the data points to be kept at the leaves (p).

In binary mvp-trees, the first vantage point (referred to as S_{v1}) divides the space into two parts, and the second vantage point (referred to as S_{v2}) divides each of these partitions into two. So the fanout of a node in a binary mvp-tree is four. In general, the fanout of an internal node is denoted by the parameter m^2, where m is the number of partitions created by a vantage point. The first vantage point creates m partitions and the second point creates m partitions from each of the partitions created by the first vantage point, making the fanout of the node m^2.

In every internal node, we keep the median M_1 for the partition with respect to the first vantage point, and medians $M_2[1]$ and $M_2[2]$ for partitions with respect to the second vantage point.

The exact distances between the data points and vantage points of that leaf are kept in a leaf node. $D_1[i]$ and $D_2[i](i = 1,2, \ldots, k)$ are the distances from the first and second vantage points, respectively, where k is the maximum fanout for the leaf nodes, which may be chosen larger than the fanout m^2 of internal nodes.

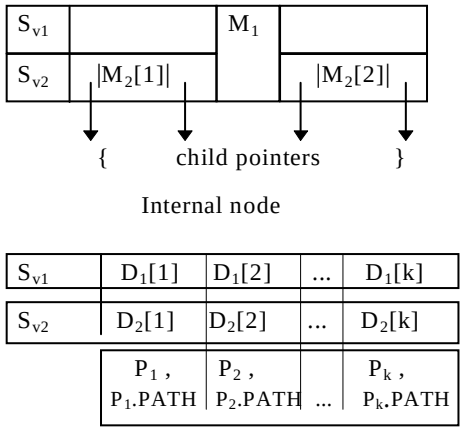

Fig. 4. Node structure for a binary mvp-tree.

For each data point x in the leaves, the array x.PATH[p] keeps the precomputed distances between the data point x and the first C vantage points along the path from the root to the leaf node that keeps x. The parameter p can not be bigger than the maximum number of vantage points along a path from the root to any leaf node. Figure 4 shows the structure of internal nodes and the leaf nodes of a binary mvp-tree.

Having given the explanation for the parameters and the structure, we present the construction algorithm next. Note that, for simplicity, in presenting the algorithm we took $m = 2$.

Constructing mvp-trees. Given a finite set $S = \{S_1, S_2, .., S_n\}$ of n objects and a metric distance function $d(S_i, S_j)$, an mvp-tree with parameters $m = 2$, k, and p is constructed on S as follows.

(Here we use the notation given in Figure 4. The variable *level* is used to keep track of the number of vantage points used along the path from the current node to the root. It is initialized to 1.)

(1) If $|S| = 0$, then create an empty tree and quit
(2) If $|S| \leq k + 2$, then
  (2.1) Select an arbitrary object S_{v1} from S. S_{v1} is the first vantage point.
  (2.2) Delete S_{v1} from S.
  (2.3) Calculate all $d(S_i, S_{v1})$ where $S_i \in S$, and store in array D_1.
  (2.4) Let S_{v2} be the farthest point from S_{v1} in S; S_{v2} is the second vantage point.
  (2.5) Delete S_{v2} from S.
  (2.6) Calculate all $d(S_j, S_{v2})$ where $S_j \in S$, and store in array D_2.
  (2.7) Quit
(3) Else if $|S| > k + 2$, then
  (3.1) Let S_{v1} be an arbitrary object from S. S_{v1} is the first vantage point.

(3.2) Delete S_{v1} from S.

(3.3) Calculate all $d(S_i, S_{v1})$ where $S_i \in S$, if $(level \leq p)$, *then* S_i.PATH $[level] = d(S_i, S_{v1})$

(3.4) Order the objects in S with respect to their distances from S_{v1}.
M_1 = median of $\{d(S_i, S_{v1}) \mid S_i \in S\}$. Break this list into 2 lists of equal cardinality at the median.
Let SS_1 and SS_2 be these two sets in order, that is, SS_2 keeps the farthest objects from S_{v1}.

(3.5) Let S_{v2} be an arbitrary object from SS_2. S_{v2} is the second vantage point.

(3.6) Let $SS_2 := SS_2 - \{S_{v2}\}$ (Delete S_{v2} from SS_2)

(3.7) Calculate all $d(S_j, S_{v2})$ where $S_j \in SS_1$ or $S_j \in SS_2$, if $(level < p)$, *then* S_j.PATH$[level + 1] = d(S_j, S_{v2})$

(3.8) $M_2[1]$ = median of $\{d(S_j, S_{v2}) \mid S_j \in SS_1\}$.
$M_2[2]$ = median of $\{d(S_j, S_{v2}) \mid S_j \in SS_2\}$.

(3.9) Break the list SS_1 into two sets of equal cardinality at $M_2[1]$
Similarly, break SS_2 into two sets of equal cardinality at $M_2[2]$
Let $level := level + 2$, and recursively create the mvp-trees on these four sets.

The mvp-tree construction can be modified easily so that more than two vantage points can be kept in one node. We talk about this generalization in Section 7. Also, higher fanouts at the internal nodes are also possible, and may be more favorable in some cases.

Observe that we chose the second vantage point to be one of the farthest points from the first vantage point. If the two vantage points were close to each other, they would not be able to effectively partition the data set. Actually, the farthest point may very well be the best candidate for the second vantage point. This is why we chose the second vantage point in a leaf node to be the farthest point from the first vantage point of that leaf node. Note that any optimization technique (such as a heuristic to choose the best vantage point) for vp-trees can also be applied to mvp-trees. We briefly discuss better ways of choosing vantage points in Section 6.

The construction step requires $O(n \log_m n)$ distance computations for the mvp-tree. There is an extra storage requirement for mvp-trees, as we keep p distances for each data point in every leaf node.

A full mvp-tree with parameters (m, k, p) and height h has $2*(m^{2h} - 1)/(m^2 - 1)$ vantage points. This is actually twice the number of nodes in the mvp-tree, since two vantage points are kept at every node. The number of data points that are not used as vantage points is $(m^{2(h-1)})*k$, which is the number of leaf nodes times the capacity (k) of a leaf node.

It is a good idea to have k large, so that most of the data items are kept in the leaves. If k is large, the ratio of the number of vantage points versus the number of points in the leaf nodes becomes smaller, meaning that most of the data points are accommodated in the leaf nodes. This makes it possible to filter out many distant (out of the search region) points from

further consideration by making use of the p precomputed distances for each point in a leaf node. In other words, instead of making many distance computations with the vantage points in the internal nodes, we delay the major filtering step of the search algorithm to the leaf level where we have more effective ways of avoiding unnecessary distance computations.

4.3 Search Algorithm for mvp-Trees

The search algorithm proceeds depth-first for mvp-trees. We keep the distances between the query object and the first p vantage points along the current search path as we will be using these distances for filtering data points in the leaves (if possible). An array, PATH[ ], of size p, is used to keep these distances.

Similarity search in mvp-trees. For a given query object Q, the set of data objects that are within distance r of Q are found using the following search algorithm:

(1) Compute the distances $d(Q, S_{v1})$ and $d(Q, S_{v2})$. (S_{v1} and S_{v2} are first and second vantage points)
  If $d(Q, S_{v1}) \leq r$ then S_{v1} is in the answer set.
  If $d(Q, S_{v2}) \leq r$ then S_{v2} is in the answer set.
(2) If the current node is a leaf node,
  For all data points (S_i) in the node,
  (2.1) Find $d(S_i, S_{v1})$ and $d(S_i, S_{v2})$ from the arrays D_1 and D_2 respectively.
  (2.2) If $[d(Q, S_{v1}) - r \leq d(S_i, S_{v1}) \leq d(Q, S_{v1}) + r]$ and
    $[d(Q, S_{v2}) - r \leq d(S_i, S_{v2}) \leq d(Q, S_{v2}) + r]$, then
    if for all $i = 1 \ldots p$
    (PATH[i] $- r \leq S_i$.PATH[i] $\leq$ PATH[i] $+ r$) holds,
    then compute $d(Q, S_i)$. If $d(Q, S_i) \leq r$, then S_i is in the answer set.
(3) Else if the current node is an internal node
  (3.1) If ($level \leq p$) then PATH[$level$] $= d(Q, S_{v1})$
    If ($level < p$) then PATH[$level$] $= d(Q, S_{v2})$
  (3.2) If $d(Q, S_{v1}) + r \leq M_1$ then
    if $d(Q, S_{v2}) + r \leq M_2[1]$ then recursively search the first branch with $level = level + 2$
    if $d(Q, S_{v2}) - r \geq M_2[1]$ then recursively search the second branch with $level = level + 2$
  (3.3) If $d(Q, S_{v1}) - r \geq M_1$ then
    if $d(Q, S_{v2}) + r \leq M_2[2]$ then recursively search the third branch with $level = level + 2$
    if $d(Q, S_{v2}) - r \geq M_2[2]$ then recursively search the fourth branch with $level = level + 2$

The efficiency of the search algorithm very much depends on the distribution of distances among the data points, the query range, and selection of vantage points. In the worst case, most data points are relatively far away from each other (such as randomly generated vectors in a high-dimensional

domain, as in Section 5). In this case the search algorithm can make $O(N)$ distance computations, where N is the cardinality of the data set. However, even in the worst case, the number of distance computations made by the search algorithm is far less than N, making it a significant improvement over linear search. Note that the claim on worst-case complexity is true for all distance-based index structures simply because all of them use the triangle inequality to filter out data points that are distant from the query point.

In the next section we present the results of our experimental study for the evaluation of performance of mvp-trees.

5. IMPLEMENTATION

We implemented the main memory model of the mvp-trees to test and compare it with the vp-trees. The mvp-tree and the vp-trees are both implemented in C under UNIX operating system. Since the distance computations are assumed to be expensive for the metric spaces we consider, the number of distance computations was used as the cost measure. The mvp-tree structure is not a paged structure, so we do not discuss the I/O performance here. We counted the number of distance computations required for similarity search queries by both mvp and vp-trees for comparison.

5.1 Data Sets

Two types of data, high-dimensional Euclidean vectors and gray-level MRI images (where each image has 256*256 pixels) were used for empirical study.

High-dimensional Euclidean vectors. We used two sets of Euclidean vectors with two different distributions. A Euclidean distance metric was used as the distance metric for all the experiments. Note that the dimensionality of the Euclidean data sets or the choice of Euclidean distance L_2 metric is not of particular significance here. There are many other indexing techniques such as TV-trees [Lin et al. 1994] and X-trees [Berchtold et al. 1996] that are particularly designed for high-dimensional Euclidean data. For general metric spaces, we only use the pairwise distances between objects in the data space for both index construction and search, as we assume that no geometric information is available. The only characteristics that would affect the query performance is the pairwise distance distribution of the objects in the metric space [Ciaccia et al. 1998b]. So although mvp-trees and vp-trees were not specifically designed for the Euclidean vectors we experimented with, using them in the experiments provides us a convenient test bed for relating search performance with different distance distributions.

Our first set of experiments were conducted on uniformly distributed Euclidean vectors. We used 50,000 uniformly distributed vectors in 10-dimensional Euclidean space. For this set, all vectors were chosen ran-

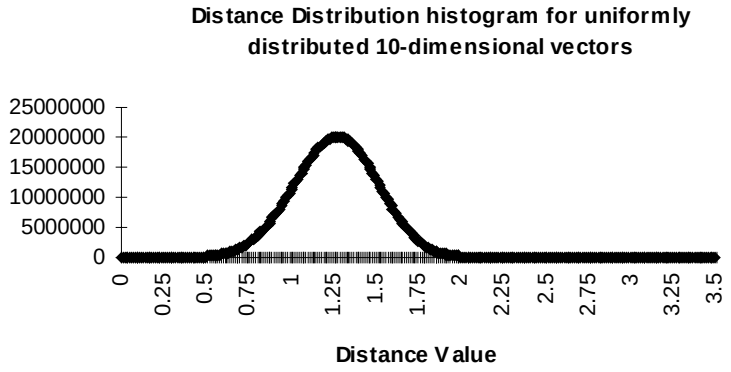

Fig. 5. Distance distribution for uniformly distributed vectors in 10–dimensional Euclidean space. (Y axis shows the number of data object pairs that have the corresponding distance value. The distance values are sampled at intervals of length 0.01).

domly from the 10-dimensional unit hypercube. The pairwise distance distribution of these uniformly distributed vectors are shown in Figure 5. The distance values are sampled at intervals of length 0.01.

The distance distribution of randomly generated 10-dimensional vectors is similar to a Gaussian curve where the distances between any two points fall mostly within the interval [0.5, 2.0], concentrating around the midpoint 1.25. For this vector set, we tried query (similarity) ranges from 0.2 to 0.5 in our experiments. The reason we chose 10-dimensional vectors to test our structures on uniformly distributed data is based on the fact that it is very hard to get meaningful results in higher dimensional spaces. We initially tried randomly generated 20-dimensional Euclidean vectors, but the selectivity for the query ranges we tried was very low. We were not able to find more than one near neighbor (mostly none at all), even for the highest query range (0.5) used in the experiments. Although 10-dimensional vectors may not be considered as high-dimensional data, the selectivity of queries is much higher in 10 dimensions, which allows us to relate the selectivity factor with query performance in the experiments for uniformly distributed vectors. The selectivity for different query ranges for 10-dimensional vectors is presented in Section 5.2.

Another set of experiments were conducted on 20-dimensional Euclidean vectors generated in clusters of equal size. The clusters were generated as follows. First, a random vector is generated from the 20–dimensional unit hypercube with each side of size 1. This random vector becomes the seed for the cluster. Then the other vectors in the cluster are generated from this vector, or a previously generated vector in the same cluster, simply by altering each dimension of that vector with the addition of a random value chosen from the interval $[-\varepsilon, \varepsilon]$, where ε is a small constant (between 0.1 to 0.2).

Since most of the points are generated from previously generated points, the accumulation of differences may become large, and therefore, there are many points that are distant from the seed of the cluster (and from each

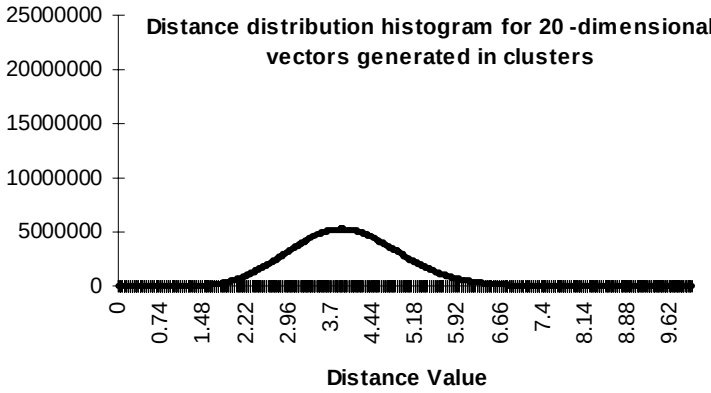

Fig. 6. Distance distribution for 20-dimensional Euclidean vectors generated in clusters. (Y axis shows the number of data object pairs that have the corresponding distance value. The distance values are sampled at intervals of length 0.01).

other), and many are outside of the unit hypercube. We call these groups of points clusters because of the way they are generated, not because they are a bunch of points that are physically close in the Euclidean space. In Figure 6, the distance distribution histogram for a set of clustered data is shown, where each cluster is of size 1000, and ε is 0.15. Again the distance values are sampled at intervals of size 0.01. One quickly realizes that this data set has a different distance distribution where the possible pairwise distances have a wider range. The distribution is not as sharp as it was for random vectors. For this data set, we tested similarity queries with r ranging from 0.2 to 1.0 for different query sets. We did not try the same experiments on 10-dimensional vectors (generated in the same way), as we were able to observe the query range/selectivity relationship on 20-dimensional space. The selectivity of the queries for this data set is discussed in Section 5.2 as well.

Gray-level MRI images. We have also experimented on 1151 MRI images with 256*256 pixels and 256 values of gray level. These images are a collection of MRI head scans of several people. Since we did not have any content information on these images, we simply used L_1 and L_2 metrics to compute the distances between images. Remember that the L_p distance between any two N-dimensional Euclidean vectors X and Y (denoted $D_p(X, Y)$) is calculated as follows:

$$D_p(X, Y) = \sqrt[p]{\sum_{i=1}^{N} (|X_i - Y_i|)^p}$$

the L_2 metric is the Euclidean distance metric. An L_1 distance between two vectors is simply found by accumulating absolute differences for each dimension.

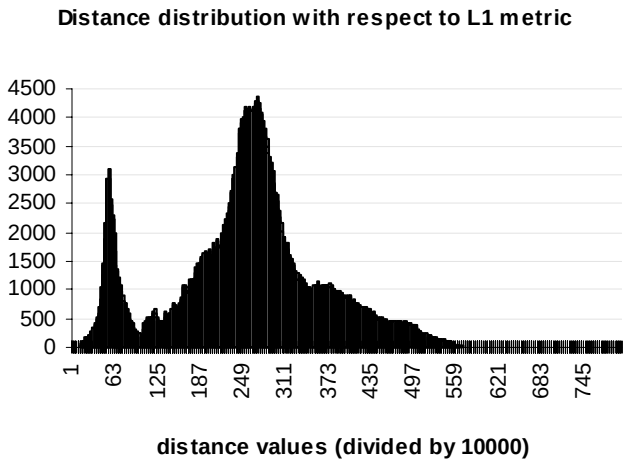

Fig. 7. Distance histogram for images when the L_1 metric is used.

When calculating distances, these images are simply treated as 256*256 = 65,536-dimensional Euclidean vectors, and the pixel by pixel intensity differences are accumulated using L_1 or L_2 metrics. This data set is a good example where it is very desirable to decrease the number of distance computations by using an index structure. The distance computations not only require a large number of arithmetic operations, but also require considerable I/O time, since the images are stored on disk using around 61K per image (images are in binary PGM format using one byte per pixel).

The distance distributions of the MRI images for L_1 and L_2 metrics are shown in the two histograms in Figures 7 and 8. There are (1150*1151)/2 = 658,795 different pairs of images, and hence as many computations. The L_1 distance values are normalized by 10,000 to avoid large values in all distance calculations between images. The L_2 distance values are similarly normalized by 100. After the normalization, the distance values are sampled at intervals of length 1 in each case.

The distance distribution for the images is much different than the one for Euclidean vectors. There are two peaks, indicating that while most of the images are distant from each other, some of them are quite similar, probably forming several clusters. This distribution also gives us an idea about choosing meaningful tolerance factors for similarity queries, in the sense that we can see what distance ranges can be considered similar. If the L_1 metric is used, a tolerance factor (r) around 500,000 is quite meaningful, while the if L_2 metric is used, the tolerance factor should be around 3000.

It is also possible to use other distance measures as well. Any L_p metric can be used just like L_1 or L_2. An L_p metric can also be used in a weighted fashion where each pixel position is assigned a weight that would be used to multiply intensity differences of two images at that pixel position when

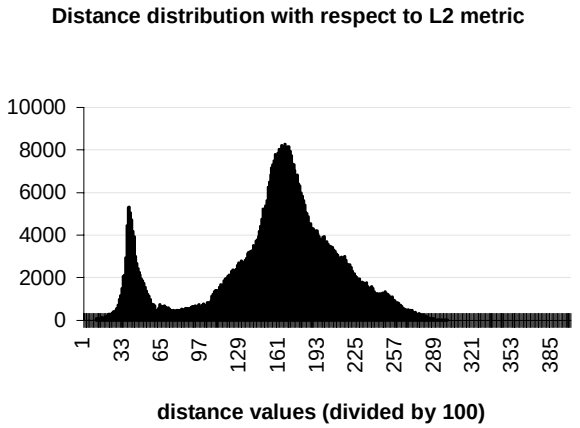

Fig. 8. Distance histogram for images when the L_2 metric is used.

computing the distances. Such a distance function can be easily shown to be metric. It can be used to give more importance to particular regions (for example, center of the images) in computing distances.

5.2 Experimental Results

High-dimensional Euclidean vectors. For Euclidean vectors, we present the search performances of four tree structures: the vp-trees of order 2 and 3 and two mvp-trees with the (m, k, p) values (3,9,5) and (3,80,5), respectively. In the experiments with vp-trees of higher order, we observed that higher order vp-trees give similar or worse performances, hence those results are not presented here. We have also tried several mvp-trees with different parameters; however, we observed that order 3 (m) gives slightly better (but very close) results compared to order 2 or any value higher than 3. We kept 5 (p) reference points for each data point in the leaf nodes of the mvp-trees. The two mvp-trees for which we display the results have different k (leaf capacity) values to see how it affects search efficiency. We do not take into account how the leaf and internal nodes of an mvpt-tree are paged because we do not consider I/O behavior in this study. In the following figures, the mvp-tree with (m, k, p) values (3,9,5) is referred to as mvpt(3,9) and the other mvp-tree is referred to as mvpt(3,80), since both trees have the same p values. The vp-trees of order 2 and 3 are referred to as vpt(2) and vpt(3), respectively.

We discuss the results on uniformly distributed data sets first. In all the experiments, the query points are generated in the same way as the data points are; that is, they conform to the same uniform distribution. The results in Figure 9 are obtained by taking the average of four different runs for each structure where a different seed (for the random function used to pick vantage points) is used in each run. The result of each run is obtained by averaging the results of 100 search queries. For this set of experiments, selectivity versus query range information is given in Table I, where the

Table I. Total Number of Near Neighbors in Experiments with 10-Dimensional Random Vectors

Query range (r)	**0.2**	**0.3**	**0.4**	**0.5**
Number of near neighbors found	1	38	503	3431

total number of near neighbors found in 100 queries is shown for query ranges 0.2 thru 0.5.

Figure 9 shows the performance results for 10-dimensional uniformly distributed Euclidean vectors. As shown in the figure, the mvp-trees perform better than vp-trees, especially for small query ranges. Between the vp-trees, vpt(2) was the one with superior performance, compared to vpt(3) for all query ranges. Compared to vpt(2), mvpt(3,9) performed 25%, 20%, 10%, and 4% fewer distance computations for query ranges 0.2, 0.3, 0.4, and 0.5 (respectively), where mvpt(3,80) performed 65%, 40%, 20% and 3% fewer distance computations for the same query ranges. We see that the performances are very close for high query ranges because the selectivity of queries jump quickly for high query ranges (see Table I), making it harder to filter out non-near-neighbor data points during the search for all structures.

For Euclidean vectors generated in clusters, we use two different query sets. In the first query set, which we refer to as Q_1, all query objects are generated randomly from the 20-dimensional unit hypercube, as in the previous test case. In the second query set, which we refer to as Q_2, query objects are generated by slightly altering randomly chosen data objects, so that we are guaranteed finding some near neighbors during query evaluation. Note that this set of query points actually conforms to data distribution, as the data points are generated in the same way. The experimental results for the two query sets are shown in Figures 10 and 11. Each query set contains 100 objects, and results are obtained by averaging two different runs (with different seeds). Table II shows the total number of near neighbors found during the evaluation of these queries for different query ranges.

Figure 10 shows the performance results for the data set where the vectors are generated in clusters and the query objects are generated randomly (query set Q_1). For these data and query sets, vpt(3) performs slightly better than vpt(2) (around 10%). The mvp-trees again perform much better than vp-trees. The mvpt(3,80) makes around 70%–80% fewer distance computations than vpt(3) for small query ranges (up to 0.4), where mvpt(3,9) makes around 45%–50% fewer computations for the same query ranges. For higher query ranges, the gain in efficiency decreases slowly as the query range increases. For the query range 1.0, mvpt(3,80) requires 25% fewer distance computations compared to vpt(3), and mvpt(3,9) requires 20% fewers.

Figure 11 shows performance results for the data set where the vectors are generated in clusters (query set Q_2). For this data set, the perfor-

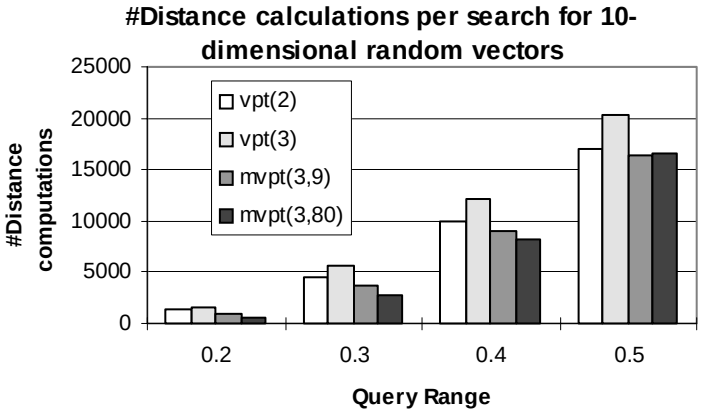

Fig. 9. Search performance of vp and mvp trees for 10-dimensional randomly generated Euclidean vectors.

Table II. Total Number of Near Neighbors for Query Sets Q_1 and Q_2 with Respect to Different Query Ranges (for Euclidean vectors generated in clusters)

Query range (r)	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	**1.0**
# Near neighbors found (Q_1)	0	0	0	0	0	1	3	20	105
# Near neighbors found (Q_2)	2	95	101	132	246	344	464	647	875

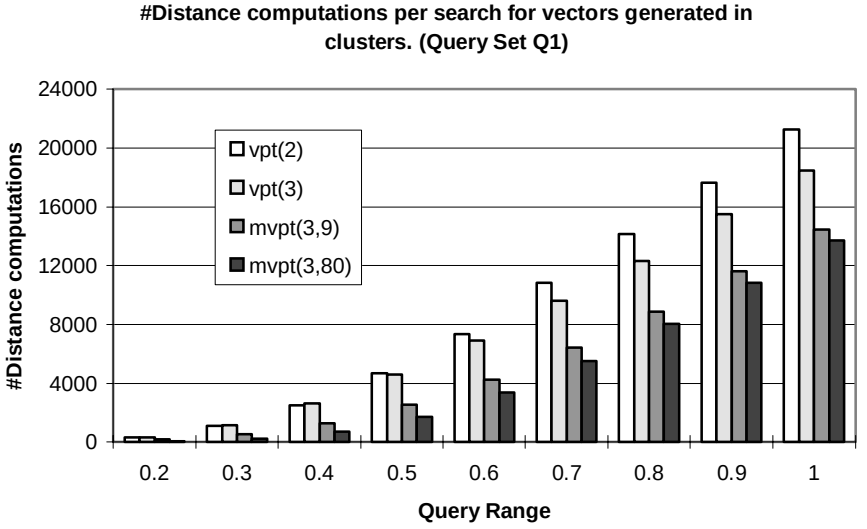

Fig. 10. Search performances of vp and mvp trees for Euclidean vectors generated in clusters (query set Q_1).

mances of vpt(3) and vpt(2) were very close. Other than that, the relative performances of the index structures were very similar to the previous case, although the absolute number of distance computations is less than for randomly generated query points (query set Q_1). Again, mvpt(3.80)

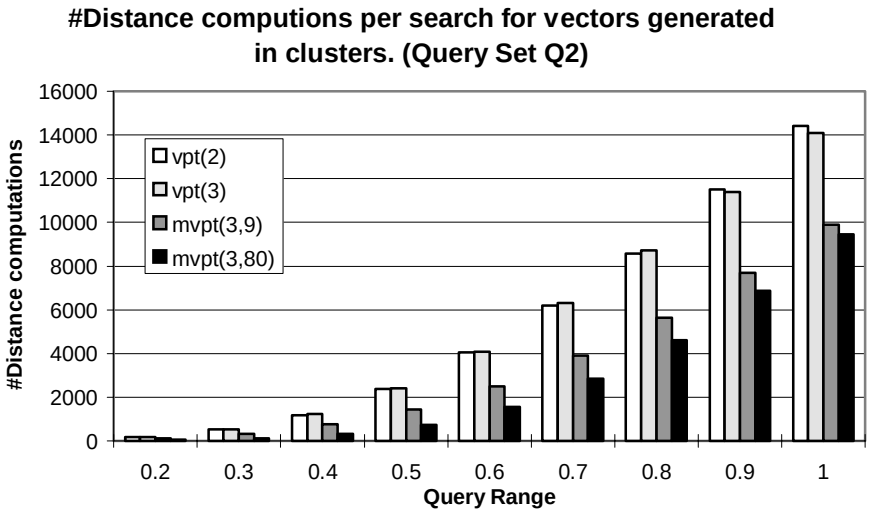

Fig. 11. Vp and mvp tree search performance for Euclidean vectors generated in clusters (query set Q_2).

makes around 70% fewer distance computations compared to vpt(3) for small query ranges, and around 30% for highest query ranges. Mvpt(3,9) makes around 45% to 30% fewer distance computations compared to vpt(3).

We can summarize our observations as follows:

- Higher order vp-trees perform slightly better for wider distance distributions; although the difference is very small. For data sets with narrow distance distributions, low-order vp-trees are better.
- mvp-trees perform much better than vp-trees. The idea of increasing leaf capacity pays off, since it decreases the number of vantage points by shortening the height of the tree and delays the major filtering step to the leaf level.
- For both random and clustered vectors, mvp-trees with high leaf node capacity are a considerable improvement over vp-trees, especially for small query ranges (up to 80%). The efficiency gain (in number of distance computations) is smaller for larger query ranges, but still significant (around 30% for 20-dimensional vectors).

Gray-level MRI images. The experimental results for similarity search performances of vp and mvp trees on MRI images are given in Figures 12 and 13. For this domain, we present the results for two vp-trees and three mvp-trees. The vp-trees are of order 2 and 3, referred to as vpt(2) and vpt(3). All the mvp-trees have the same p paramete, which is 4. The three mvp-trees are mvpt(2,16), mvpt(2,5), and mvpt(3,13) where the first parameter is the order (m) and the second is the leaf capacity (k). We did not try for higher m or k values, as the number of data items in our domain is small (1151). Actually, 4 is the maximum p value common to all three

Table III. Average Number of Near Neighbors for Images Using the L_1 Metric

	Query range (normalized by 10000)					
L_1 Metric	**30**	**40**	**50**	**60**	**80**	**100**
# Near neighbors found	5.8	13.8	38.6	94.9	127.8	134

mvp-tree structures due to the low cardinality of the data domain. The results are averages taken after different runs for different seeds and for 30 different query objects in each run. Query objects are MRI images selected randomly from the data set.

The selectivity of the query ranges is shown in Tables III and IV for L_1 and L_2 metrics, respectively. As we mentioned in Section 5.1, the images seem to form several clusters as the number of near neighbors found seems to be around 100 for moderate to large query ranges for both distance metrics.

The search performances of the five structures we tested (two vp-trees and three mvp-trees) for the L_1 metric are shown in Figure 12. The query range values shown are normalized by 10,000 as for Figure 7. Among the vp-trees, vpt(2) performs around 10–20% percent better than vpt(3). mvpt(2,16) and mvpt(2,5) perform very close to each other, both having around 10% edge over vpt(2). The best one is mvpt(3,13) performing around 20–30% fewer distance computations compared to vpt(2).

The search performances for the L_2 metric are shown in Figure 13. The query range values shown are normalized by 100 as for Figure 8. Similar to the case when the L_1 metric was used, vpt(2) outperforms vpt(3) with a similar approximate 10% margin. mvpt(2,16) performs better than vpt(2) but its performance degrades for higher query range values. This should not be taken as a general result, since the random function used to pick vantage points has a considerable effect on the efficiency of these structures (especially for small cardinality domains). Similar to the previous case, mvpt(3,13) gives the best performance among all the structures, once again making 20–30% fewer distance computations compared to vpt(2).

In summary, the experimental results for the data set of gray-level images support our previous observations about the efficiency of mvp-trees with high leaf-node capacity. Even though our image data set has a very low cardinality (leading to shallow tree structures), we were able to get around 20–30% gain in efficiency. If the experiments were conducted on a larger set of images, we would expect higher performance gains.

6. CHOOSING VANTAGE POINTS

In Yiannilos [1993], it was suggested that, when constructing vantage point trees, choosing vantage points from the corners of the space leads to better partitions, and hence better performance. This heuristic can be used if we have an idea about the geometry of the data space and the distribution of the data points in the data space. In the general case, we simply do not

Table IV. Average Number of Near Neighbors for Images Using the L_2 metric

	Query Range (normalized by 100)						
L_2 Metric	**10**	**20**	**30**	**40**	**50**	**60**	**80**
# Near neighbors found	1	2.43	11.3	73.6	122.1	132.7	153.6

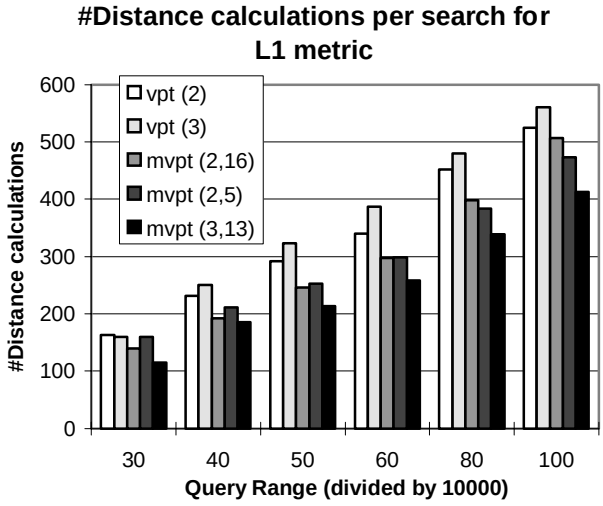

Fig. 12. Similarity search performances of vp and mvp trees on MRI images when the L_1 metric is used for distance computations.

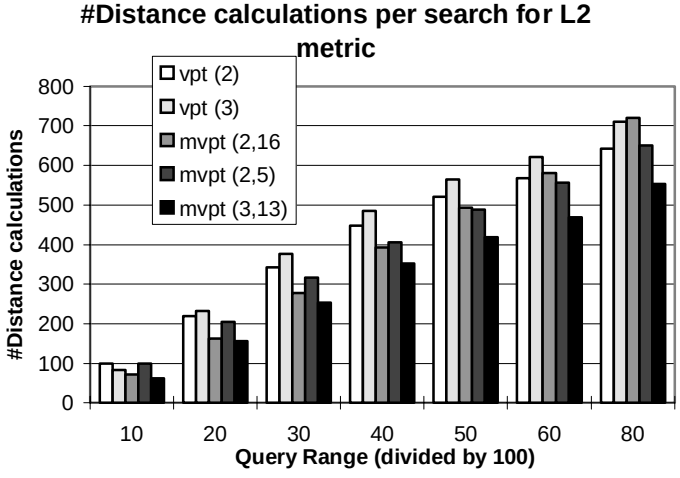

Fig. 13. Similarity search performances of vp and mvp trees on MRI images when the L_2 metric is used for distance computations.

have any idea about where the corners of the data space are. The only information we can make use of is the pairwise distances between objects.

There is a reason why the vantage points chosen from around the corners of the data space provide better partitions. Basically, the distance distribu-

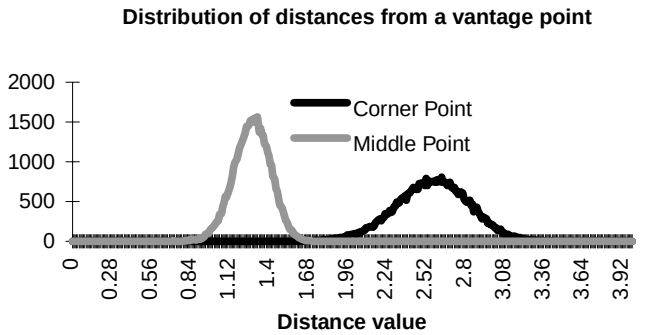

Fig. 14. Distance distributions for two vantage points, one is the center of the 20-dimensional Euclidean hypercube (middle point), the other is one of the corners of the cube (corner point). The data points are distributed uniformly.

tions for the corner points are more *flat* than the distance distributions for the center points. Figure 14 illustrates this point. For a uniformly distributed 20-dimensional Euclidean data space, we computed the distances of all the data points from two vantage points: the first one is the center point, and the second is a corner point. As we can easily see from the figure, the distribution for the center point is sharper, which means that using the center point as a vantage point is less useful in trimming the search during similarity queries.

Most of the data points are far away from the corner points, which is a trivial fact that can also be observed from Figure 14. This simple fact is actually why the vantage points from the corners work better. In the general case, for metric spaces, although we may not be able to choose vantage points from the corners of the space, we may still be able to choose better vantage points. Here, we suggest a simple heuristic.

Choosing a Vantage Point:
(1) Choose a random point.
(2) Compute the distances from this point to all the other points.
(3) Choose the farthest point as the vantage point.

Note that the simple procedure above cannot guarantee choosing the very best vantage point, but it does help in choosing better vantage points compared to those chosen without this heuristic (i.e., randomly). In case of Euclidean spaces, this heuristic is verifiable for some distributions simply because the farthest point from any given point is most likely to be a point that is close to the corner (or sides of the Euclidean hypercube). We tested this simple heuristic to see if it provides better performance on the 20-dimensional Euclidean vector sets generated in clusters. But this time the comparison is between mvp-trees that randomly choose the first vantage point in any internal node and the mvp-trees that choose the first vantage point using the heuristic shown below. We show the results only for randomly generated query objects (referred to as Q_1 in Section 5).

Figure 15 shows the result of this comparison when an mvp-tree with parameters $m = 3$, $k = 80$, $p = 5$ (mvpt(3,80)) is used. The performance

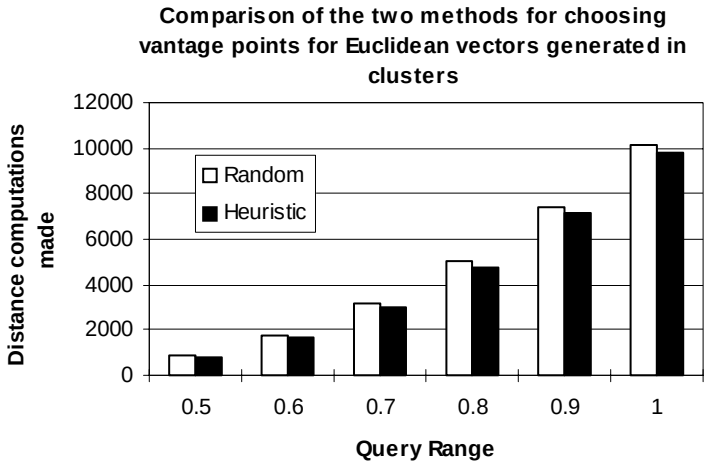

Fig. 15. Query performance of mvpt (3,80) for the two different methods for choosing vantage points for 20-dimensional Euclidean vectors generated in clusters.

gain varied between 5% to 10%, in terms of the average number of distance computations in a query. Note that this performance gain comes at the expense of an increased number of distance computations at construction time. Actually, it is also possible to use the random vantage point (the one picked first at step 1) as the second vantage point, in which case there would not be any extra distance computations made.

In trimming the search during similarity queries, it is also important that the consecutive vantage points seen along a path are not very close to each other. In mvp-trees, this also makes for better utilization of the precomputed distances at search time. We have already adopted this strategy by choosing the second vantage point in an internal node to be one of the farthest points from the first one. In Section 7, when explaining the generalized mvp-tree structure that may have any number of vantage points in an internal node, we use the same strategy.

7. GENERALIZED MVP-TREES

In Section 4, when we introduce the multivantage point tree structure, we only consider the case where two vantage points are used in an internal node to hierarchically partition the data space. As mentioned before, the construction and search algorithms can be modified easily, so that more than two vantage points can be used in an internal node. In this section we change the structure of the mvp-tree a little bit and treat the number of vantage points in an internal node as a parameter. So in addition to the parameters m, k, and p, a fourth parameter, v, is introduced as the number of vantage points used in an internal node.

The leaf node structure is also changed as a minor improvement. The leaf nodes do not contain vantage points any more, but they only accommodate the data points and p precomputed distances for each data point. When the

search proceeds down to a leaf node, only these p precomputed distances will be used to filter out distant data items. Again, the distances kept for a data item are the distances from the first p of the vantage points along the path, starting from the root node to the leaf node that contains the data item.

7.1 Constructing Generalized Mvp-Trees

The vantage points in an internal node are selected in a similar way to that explained in Section 4. The first vantage point, say vp_1, is picked randomly (or using the heuristic in Section 6), and it is used to partition the data space into m spherical shell-like regions, which are referred to as R_1, . . . R_m (R_1 is the partition that keeps the closest points and R_m is the partition that keeps the farthest points). The farthest point from vp_1 is selected as the second vantage point, vp_2. This time, vp_2 is used to partition the regions R_1, . . . R_m into further m regions, creating m^2 regions $\{R_{i,j} \mid i, j = 1, \ldots m\}$. Here $R_{i,1}$. . . $R_{i,m}$ are the partitions of the region R_i . If $v > 2$, the third vantage point is chosen as the farthest point from vp_2 in $R_{m,m}$. This guarantees that the third vantage point is distant from the previous vantage points, namely, vp_1 and vp_2. It is distant from vp_1 because it is one of the data points in partition R_m, which accommodates the farthest points from vp_1. Similarly, vp_3 is distant from vp_2, because it is the farthest point from vp_2 among all the points from $R_{m,m}$. Note that vp_3 may not be the farthest point from vp_2 (the farthest point may be in $R_{i,m}$ where $i \neq m$), but it is still a distant point. This process continues in the same way until all v of the vantage points are chosen, and the data space is partitioned into m^v regions $\{R_{i1,\ldots,iv} \mid i1, \ldots iv = 1, \ldots, m\}$. The construction algorithm is given below.

Constructing an mvp-tree with parameters m, v, k, p. Given a finite set $S = \{S_1, \ldots, S_n\}$ of n objects, and a metric distance function $d(S_i, S_j)$, an mvp tree with parameters m, v, k, and p is constructed as follows. The notation is the same as in Section 4.

(1) If $|S| = 0$, then create an empty tree and quit.
(2) If $|S| \leq k$ then
  (2.1) Create a leaf node L and put all the data items in S to L
  (2.2) Quit.
(3) Else if $|S| > k$ then
  (3.1) Let S_{v1} be an arbitrary object from S; S_{v1} is the first vantage point.
  (3.2) Delete S_{v1} from S.
  (3.3) Calculate all $d(S_i, S_{v1})$ where $S_i \in S$
  (3.4) If ($level \leq p$) then
    S_i.PATH[l] = $d(S_i, S_{v1})$.
    $level := level + 1$
  (3.5) Order the objects in S with respect to their distances from S_{v1}. Break

this list into m lists of equal cardinality, recording all the distance values at cutoff points. Denote these lists as $SR_1, \ldots, SR_m$.

(3.6) Let $j = 2$ (j is just a loop variable)

(3.7) While $j \leq v$ do

(3.7.1) Choose S_{vj} to be the farthest point from $S_{v(j-1)}$ in $SR_{m,\ldots,m}$ ($(j-1)$ m's)

(3.7.2) Delete S_{vj} from $SR_{m,\ldots,m}$

(3.7.3) Calculate all $d(S_j, S_{vj})$ where $S_j \in SR_{i1,i2,\ldots i(j-1)}$ where $i1, i2, \ldots i(j-1) = 1, \ldots, m$

(3.7.4) If $(level \leq p)$ then

S_i.PATH[$level$] = $d(S_j, S_{vj})$

(3.7.5) Use these distances to partition each of the $SR_{i1,i2,\ldots i(j-1)}$ $(i1, i2, \ldots i(j-1) = 1, \ldots, m)$ regions further into m more regions, creating $SR_{i1,i2,\ldots ij}$ $(i1, i2, \ldots ij = 1, \ldots, m)$. Record cutoff values.

(3.7.6) $level = level + 1$;

$j = j + 1$;

(3.8) Recursively create the mvpt-tree on each of the m^v partitions, namely $SR_{i1,i2,\ldots,iv}(i1, i2, \ldots iv = 1, \ldots, m)$.

The search algorithm is similar to the one discussed in Section 4. Starting from the root, all the children whose regions intersect with the spherical query region will be visited during a search query. When a leaf node is reached, the distant data points will be filtered out by looking at the precomputed distances (the first p of them) from the vantage points higher up in the tree.

7.2 Updating Mvp-Trees

Here we briefly discuss the update characteristics of the generalized mvp-tree structure. Since the mvp-tree is created from an initial set of data objects in a top-down fashion, it is a rather static index structure. It is also balanced because of the way it is constructed (the number of objects indexed in the subtrees of a node can differ by at most 1). Note that all the distance-based index structures other than M-trees are created top-down, and are therefore static like mvp-trees. However, it is possible to handle dynamic insertions if it is allowed to violate the balance of the mvp-tree, in which case the tree structure may grow downwards in the direction of the tree branches where the insertions are made. If the distribution of the dynamically inserted data points conforms to the distribution of the initial data set that the index is built on, the mvp-tree grows smoothly, staying balanced, or close to being balanced. If the insertions cause the tree structure to be skewed (that is, the additions of new data points change the distance distribution of the whole data set), global restructuring may have to be done, possibly during off hours of operation. The number of distance computations that have to be done during a restructuring process depends on the number of precomputed distance values kept in the leaf nodes. If all precomputed distance values are kept, they can be reused (via choosing the

same vantage points) during the restructuring process, and the restructuring would be done with minimum distance computation possible. The implementation and evaluation of these strategies for updating the mvp-trees are in our agenda for future work.

In the next section, the results of experiments using different values for the parameters of the mvp-trees are provided, and the query performances are compared for these cases.

8. EXPERIMENTS WITH GENERALIZED MVP-TREES

In these experiments, the same sets of Euclidean vectors are used as in Section 5, where in the first set the vectors are generated randomly, and in the second set the vectors are generated in clusters. Pairwise distance distributions of these data sets are given in Section 5. We use query ranges starting from 0.2 thru 0.5 for randomly generated vectors, and 0.3 thru 1.0 for vectors generated in clusters. Randomly generated vectors are tested using randomly generated query objects, and Euclidean vectors generated in clusters are tested using both query sets Q_1 (randomly generated query objects) and Q_2 (query objects generated from randomly chosen data objects) as in Section 5. In Section 8.1, we investigate the effect of using different numbers of vantage points in the internal nodes, and utilizing precomputed distances at search time. In Section 8.2, we present experimental results done with one of the state of the art distance-based index structures, the M-trees,[1] .

8.1 Tuning Mvp-Tree Parameters

In the first set of experiments we tried to see the effects of changing the number of vantage points used in an internal node of an mvp-tree. Figures 16–18 show the results for six mvp-trees with different values for the parameter v. In all structures, the parameters m, k, and p are the same, having the values 2, 13, and 7, respectively. In Figures 16–18, these structures are referred to by their (m, v, k, p) parameters. The parameters v and k are chosen in such a way that all the trees have the same number of vantage points along any path from the root node to any leaf node. For example, for the structure mvpt($m = 2$, $v = 1$, $k = 13$, $p = 7$), for a set of 50,000 Euclidean vectors, there are 12 vantage points along any path from the root to a leaf, since $\lceil \log_2(50{,}000 / k) \rceil = 12$. The same holds for the other structures as well.

Note that the structure mvpt(2,1,13,7) is not much different than the binary vp-tree (one vantage point in every internal node), except for the fact that precomputed distances are used in this structure during search time, and the leaf size is larger. We should also mention that

[1]The authors thank Paolo Ciaccia and Marco Patella for providing, for purposes of comparison, the code of the M-tree.

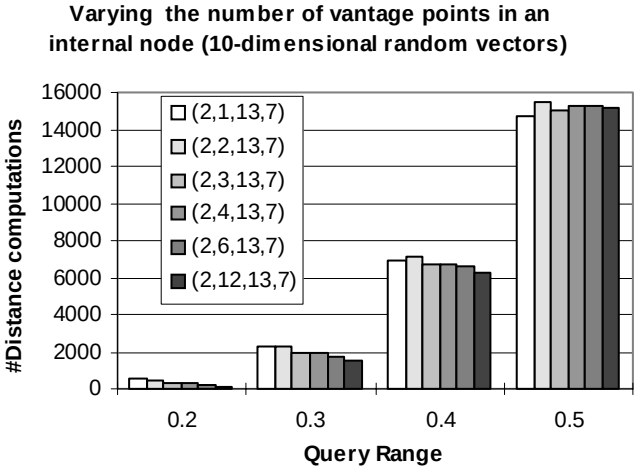

Fig. 16. Performance of mvp-trees with different v values for 10-dimensional uniformly distributed Euclidean vectors.

mvpt(2,12,13,7) is a tree structure with only one internal node, meaning the total number of vantage points in the whole tree is 12.

The performance results shown in Figures 16–18 are quite interesting. For clustered vectors and query set Q_1 (randomly generated query objects) (Figure 17), we see that mvpt(2,1,13,7) catches up with the other structures for moderate query ranges and then actually performs slightly better for large query ranges, except for mvpt(2,12,13,7), which remains the best for all query ranges. This tells us that for large query ranges, when many distance computations have to be made anyway, having a large number of vantage points at the internal levels of the tree sometimes actually helps in trimming the search better. On the other hand, for small query ranges, too many distance computations between the vantage points and the query point have to be made, which makes mvp-trees with smaller v values perform worse than mvp-trees with larger v values. The situation is similar in the case where 10-dimensional random vectors are used; however, mvp-trees with smaller v values close the gap quickly and perform approximately similarly for the largest query range. For query set Q_2 on clustered vectors (Figure 18), mvp-trees with higher v values perform better for all query ranges. The percentage-wise performance difference was quite high, especially for small query ranges.

In the next set of experiments, the parameter p is varied and m, v, and k are kept constant. For this set, we used the mvpt(2,12,13, p) structure (with varying p) that performed the best in our previous test. Again, performance results are obtained for both data sets of Euclidean vectors, and are as in Figures 19–21. The parameter p is varied from 7 to 12 (the maximum).

We can clearly observe that using higher p values improves the search performance significantly. To give an idea, in Figure 19, mvpt(2,12,13,12)

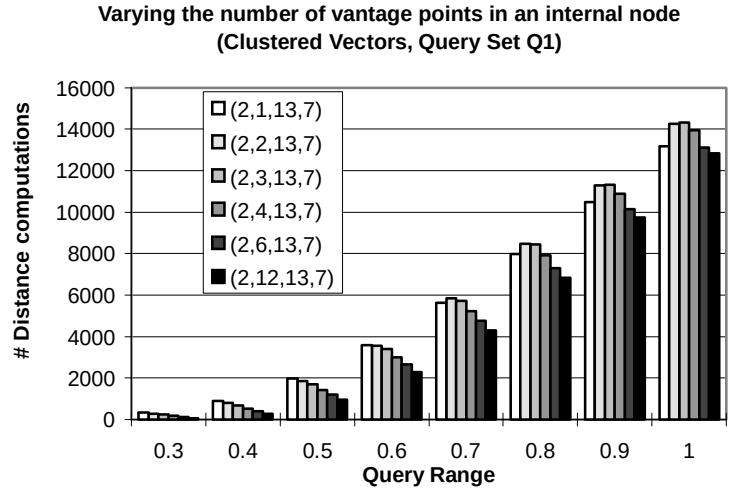

Fig. 17. Performance of mvp-trees with different v values for Euclidean vectors generated in clusters (query set Q_1).

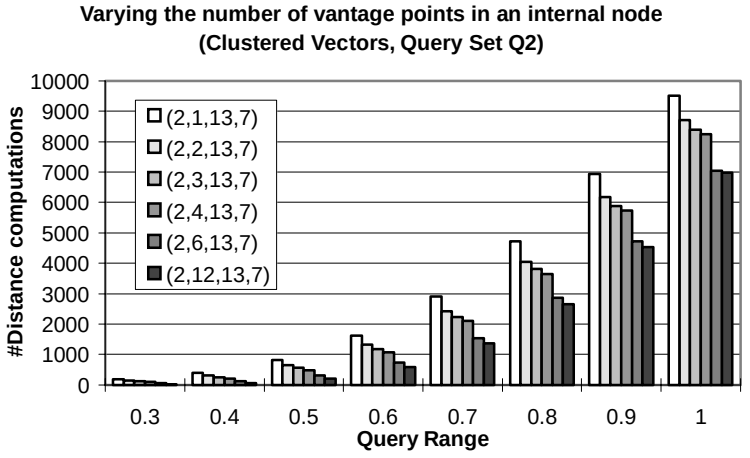

Fig. 18. Performance of mvp-trees with different v values for Euclidean vectors generated in clusters (query set Q_2).

performs 85% fewer distance computations compared to mvpt(2,12,13,7) for the query range 0.2 for 20-dimensional Euclidean vectors. The performance difference gradually decreases, and for query range 0.5 it performs 42% fewer distance computations. Similar behavior is also observed for the experiments with Euclidean vectors generated in clusters. For query set Q_1 (Figure 20), the performance difference ranges from 70% to 27% for query ranges 0.3 to 1.0. For query set Q_2 (Figure 21), the performance difference never goes below 40 %. However, more storage is needed for higher p values, which may affect the I/O time during queries.

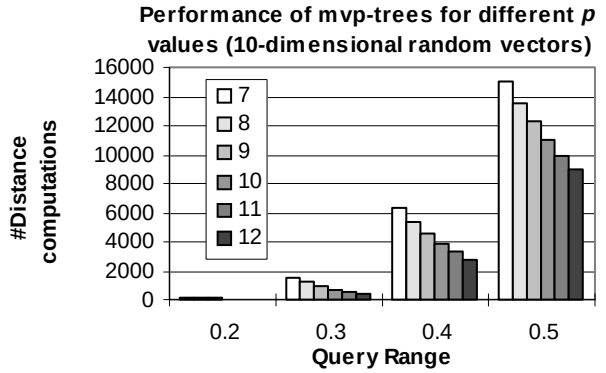

Fig. 19. Performance of mvpt(2,12,13, p) for different p values using 10-dimensional uniformly distributed Euclidean vectors.

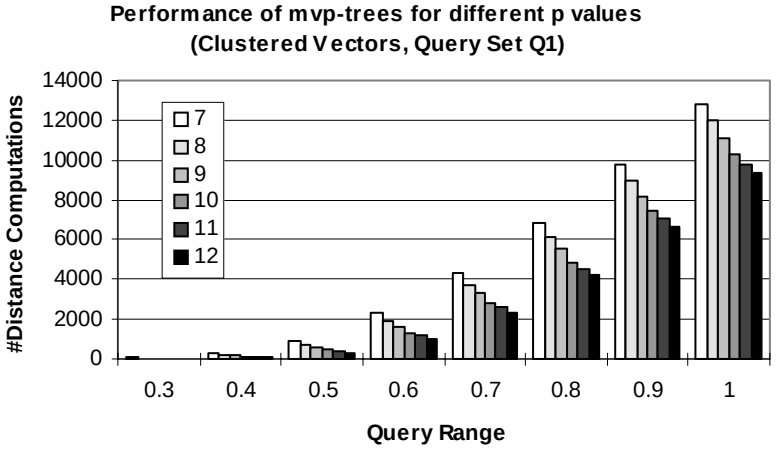

Fig. 20. Performance of mvpt(2,12,13, p) for different p values using Euclidean vectors generated in clusters (query set Q_1).

Our last set of experiments were done to investigate how changing the leaf size together with the number of vantage points in the internal nodes affects performance. We experimented on six mvp-trees where each has only a single directory (internal) node having a different number of vantage points in it, and therefore each has a different leaf size. Here we are basically varying the number of vantage points from the root to any leaf node; this is done by introducing extra levels of decomposition, and hence decreasing the leaf size. The parameter p is chosen to be the same for all structures (it is 7). The performance results are shown in Figures 22–24.

From Figures 22–24, we see that using more vantage points increases the performance by trimming more branches at lower levels of the tree. In all the structures, the number of precomputed distances (p) used in search queries are the same (which is 7), and the v values of all the structures are larger than p. That is, the vantage points used in mvpt(9,2,100,7) (all 9 of

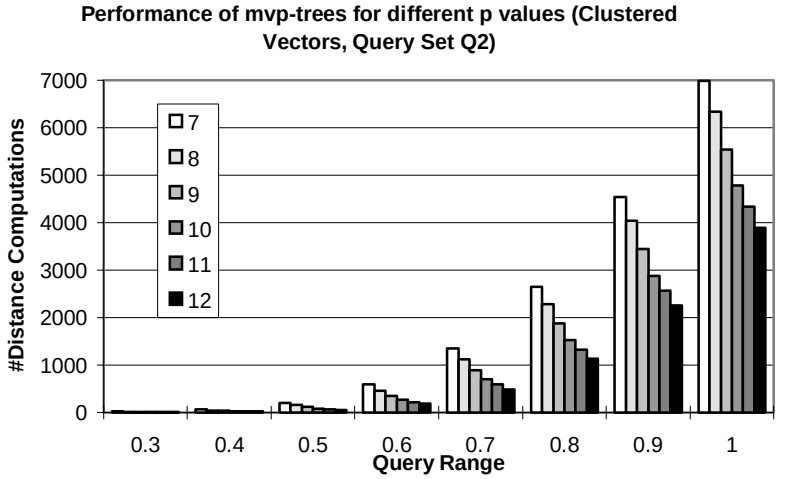

Fig. 21. Performance of mvpt(2,12,13, p) for different p values using Euclidean vectors generated in clusters (query set Q_2).

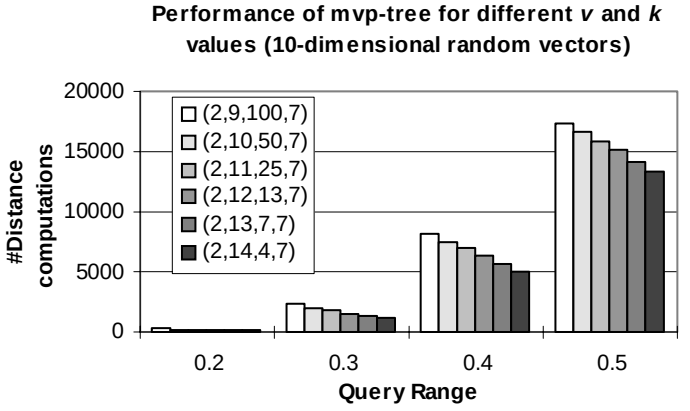

Fig. 22. Performance of mvp-trees with different v and k parameters using 10-dimensional uniformly distributed Euclidean vectors.

them) are equal to the first 9 vantage points used in the other mvp-tree structures. This means that, for all the structures, the same set of precomputed distances are kept for the data points in the leaves. Furthermore, any performance difference between these structures is simply due to the extra trimming that is done in the internal level.

Mvpt(14,2,4,7) uses 5 more vantage points compared to mvpt(9,2,100,7). Employment of 5 extra vantage points in mvpt(14,2,4,7) results in 65% to 25% fewer distance computations (performance difference decreases for increasing query ranges) for 10-dimensional random vectors; 44% to 9% fewer distance computations for Euclidean vectors generated in clusters using query set Q_1; and 50% to 22% using query set Q_2.

Our observations from these experiments can be summarized as follows:

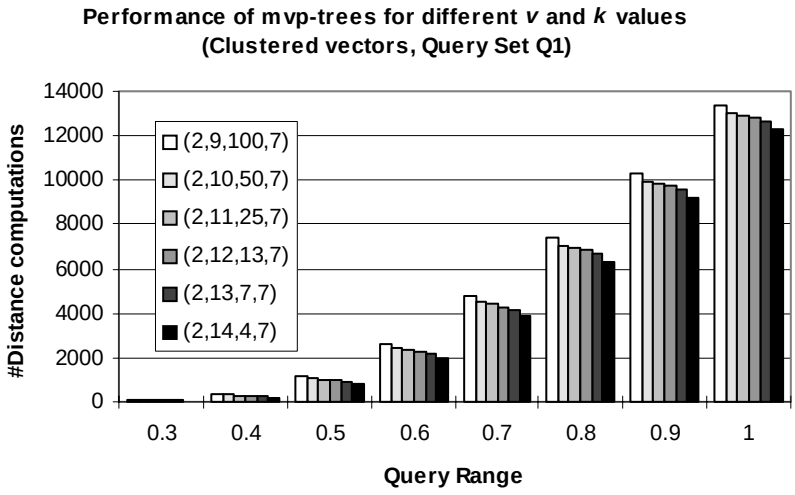

Fig. 23. Performance of mvp-trees with different v and k parameters using Euclidean vectors generated in clusters (query set Q_1).

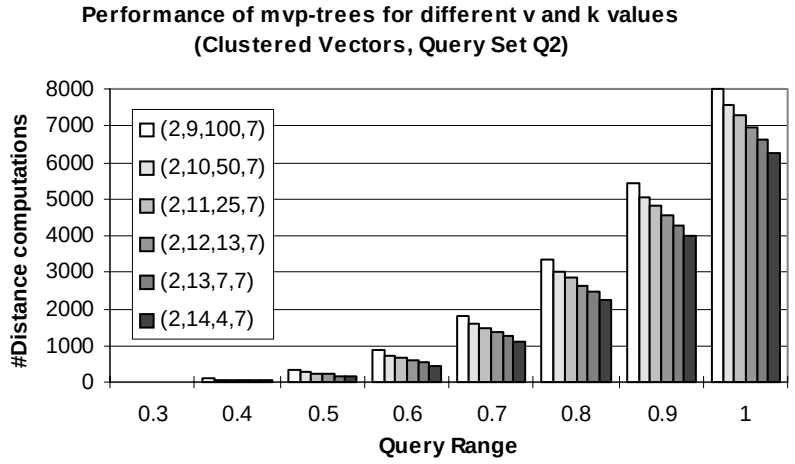

Fig. 24. Performance of mvp-trees with different v and k parameters using Euclidean vectors generated in clusters (query set Q_2).

- The best performance is obtained by using only one internal node and a single set of vantage points, as can be seen from Figures 16–18, where mvpt(2,12,13,7) performed the best. This is an interesting result, as it implies that we do not really need to come up with complex structures after all; just pick a good set of reference points (vantage points), compute the distances of the other points from these points, and create a simple directory structure using these distances to direct the search.
- As expected, using more precomputed distances at search time clearly improves search performance in terms of the number of distance computations made. If all the precomputed distances are kept for the data

points in the leaves, and the mvp-tree structure consists of a single internal node (as discussed above, ex: mvpt(2,12,13,7)), it may further imply that restructuring the tree after a batch of dynamic operations could be handled with only minimum number of distance computations and the cost of sorting and partitioning the data points with respect to their distances from the vantage points.

- Using more vantage points and keeping small leaf sizes also increases trimming in the directory nodes, provided that there is only one internal level. If there is more than one internal level, the mvp-tree seems to suffer from making too many distance computations between the query objects and the vantage points, although it provides better filtering for the data points in the leaves.

8.2 Comparison with M-Trees

We also made some experiments comparing the M-tree structure [Ciaccia et al. 1997] to mvp-trees, in terms of the average number of distance computations made in similarity queries. We used the *BulkLoading* algorithm [Ciaccia and Patella 1998] in creating the M-tree. We remind readers that the M-tree, unlike the other distance-based index structures, including mvp-trees, is a dynamic index structure and is created bottom-up. It is designed as a paged index structure to minimize required I/O operations as well as distance computations. There are two parameters used to tune the M-tree for better performance, that is, minimum node utilization and page size. We chose the minimum node utilization as 0.2 and page size 8K. The minimum node utilization does not affect the number of distance computations made for a search query too much [Ciaccia and Patella 1998], although it makes a difference in the building costs (which we do not consider here). We actually tried two values, 0.2 and 0.3. M-trees; since minimum utilization of 0.2 performed slightly better, we only include the results for that value. The node size affects query performance for both I/O costs and distance computations [Ciaccia et al. 1998b]. (Note that we do not consider I/O performance here.) A page size of 4K was shown to be the best choice for minimizing distance computations (referred to as the CPU cost) [Ciaccia et al.1998b] for experiments with 5–dimensional vectors and using the L_∞ metric, although the difference from the other choices in page size was not drastically different (around 10% for 1K to 16K). We tried three page sizes, 4K, 8K, and 16K. M-trees with a page size of 8K gave the best performance in terms of minimizing the number of distance computations, so we show the results for a page size of 8K.

The experiments were conducted with the same data sets of Euclidean vectors discussed in Section 5. For comparison, we included three structures next to M-trees. vpt(2) is the vantage point tree of order 2, also used for the experiments discussed in Section 5. The mvp-trees are, as usual, denoted by their m, v, k, p parameters. Remember that mvpt(2,1,13,7) is actually a vp-tree that makes use of $p(7)$ of the precomputed distances and that mvpt(2,10,50,7) has one internal node with 10 vantage points and a

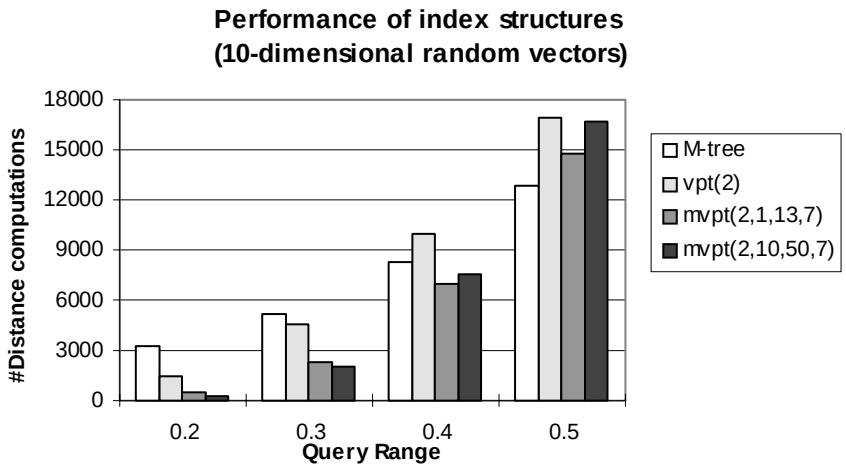

Fig. 25. Comparison of M-tree performance with two mvp-trees and a vp-tree of order 2 for 10-dimensional randomly generated Euclidean vectors.

large leaf capacity. Note that the results for the vp-tree and the mvp-trees are also given in Sections 5.2 and 8.1.

In Figures 25–27, we see the query performances of the four index structures in terms of the average number of distance computations made in a similarity search query. For 10-dimensional randomly generated Euclidean vectors (Figure 25), M-tree performs very close to vpt(2) for moderate query ranges. For small query ranges, its performance is poor. It gets better as the query range increases, and for the range 0.5 it performs the best, surpassing vpt(2) and mvpt(2,10,50,7) performance by making 24% fewer distance computations and that of mvpt(2,1,13,7) performance by making around 10% fewer. We believe that M-trees catch up (and even surpass) mvp-trees because the distance distribution is narrow and mvp and vp-tree performances degenerate faster by increasing the query range as compared to M-trees. In mvp-trees (and vp-trees) a vantage point is employed at each node to partition the data points that are below that node, which are not necessarily physically close to each other or to the vantage point. For uniformly distributed data, this leads to the filtering mechanism being less effective when the query ranges become relatively large, but highly efficient for small to moderate query ranges. When we talk about a query range being relatively large, we mean relative to the distance distribution of the data space. M-trees adjust to increasing query ranges more gracefully because they partition the data points based on physical closeness (trying to create clusters); but this leads to too much overhead for small query ranges, where they become less efficient compared to mvp-trees.

The M-tree does not perform as well for vectors generated in clusters. When query set Q_1 is used (Figure 26), for the range 1.0, the mvp-trees make fewer than around 50% distance computations than the M-tree. (To give an idea, for query range 0.7, it is %72 for mvpt(2,1,13,7) and 80% for

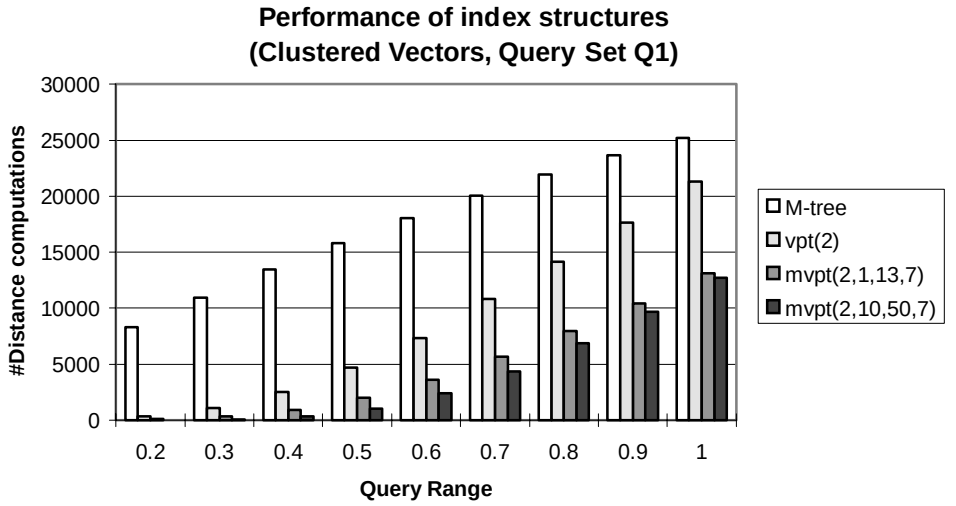

Fig. 26. Comparison of M-tree performance with two mvp-trees and a vp-tree of order 2 for 20-dimensional Euclidean vectors generated in clusters (query set Q_1).

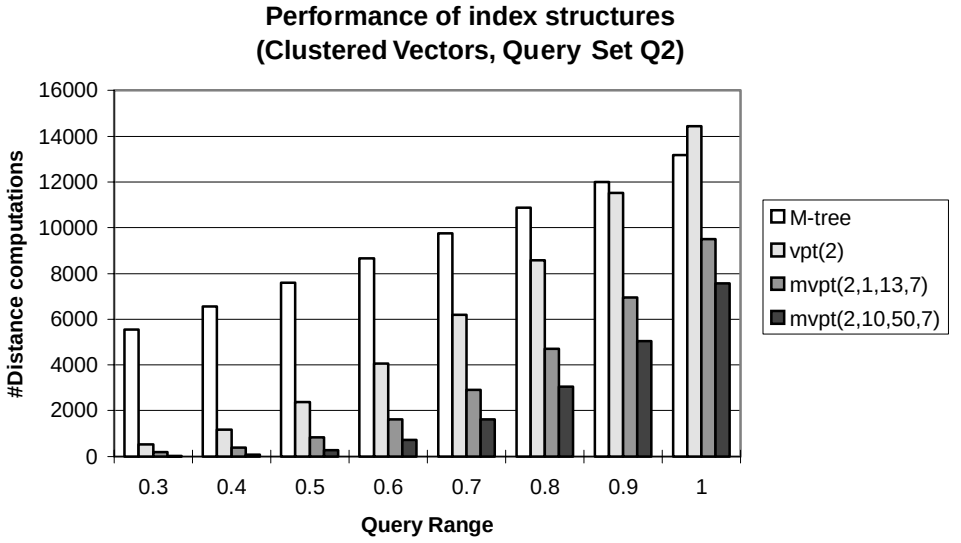

Fig. 27. Comparison of M-tree performance with two mvp-trees and a vp-tree of order 2 for 20-dimensional Euclidean vectors generated in clusters (query set Q_2).

mvpt(2,10,50,7)) . For small query ranges, the M-tree makes at least an order of magnitude more distance computations than the other structures do. When the query set Q_2 is used, the relative performances display a similar pattern as that in Figure 27. For query set Q_2, the M-tree performs close to vpt(2) for large query ranges, and actually performs fewer distance computations for the range 1.0.

The partitioning strategy of M-trees is different from mvp-trees because they (M-trees) partition the data space into a number of sphere-like regions

Table V. Performance Results for Queries with Data Set where Data Points Form Several Physical Clusters

Query Range	Total number of near neighbors found (100 queries)	Total number of distance computations			
		M-tree	vpt(2)	mvpt(2,1,13,7)	mvpt(2,10,50,7)
0.3	248	4869	5019	4774	4588
0.4	11104	5265	5194	5104	5087
0.5	98069	5545	5290	5202	5617

in every node recording the center point (routing object) and the covering radius for each partition. Naturally, this kind of a partitioning strategy is supposed to work well for applications where the data objects form several clusters of points that are physically close. We have also experimented using a data set of 50,000 20-dimensional vectors with 10 clusters. The clusters' centers are uniformly distributed in the unit hypercube and the variance is $\sigma^2 = 0.1$. Again, Euclidean distance (the L_2 metric) is used as the distance metric. We also generated 100 query points that conform to the same distribution. Table V provides the performance results for three different query ranges (0.3, 0.4, 0.5) in terms of the number of distance computations made. As expected, the performance of the M-trees in terms of the number of distance computations made is much better for the data set with physical clusters. What is more important is that the performances of mvp-trees as well as the vp-tree are comparable, the difference being less than 5% in all cases.

9. CONCLUSIONS

In this paper we have introduced the mvp-tree, which is a distance-based index structure that can be used in any metric data domain. Like the other distance-based index structures, the mvp-tree does not make any assumptions about the geometry of the data space and provides a filtering method for similarity search queries based only on relative distances between data objects. Similarly to the vp-tree, the mvp-tree takes the approach of partitioning the data space around the *vantage points*, but behaves much better in choosing these points and makes use of the precomputed (at construction time) distances when answering similarity search queries. We generalize the idea of using multiple vantage points in an internal node and experiment on mvpt-trees with different numbers of vantage points in an internal node. The experimental results show that using a small set of vantage points and a single directory node provides the best results.

Like most of the distance-based index structures, the mvp-tree is a static index structure. It is constructed top-down on a given set of data points, and is guaranteed to be balanced. Handling update operations (insertion and deletion) with reasonable costs for the mvp-tree is currently an open problem. In general, the difficulty for distance-based index structures stems from the fact that it is not possible or it is not cost-efficient to impose a global total order or a grouping mechanism on the objects of the

application data domain. In the case for mvp-trees, to keep the tree balanced, periodic restructuring operations may be needed. Although restructuring is unavoidable for balanced mvp-trees, it can be done with a minimum number of distance computations if all the precomputed distances are kept and an mvp-tree with a single directory node is used.

We considered the number of distance computations as the cost measure for comparing performance of mvp-trees with other access structures. While this is justifiable for applications where distance computations are very expensive, in the general case, I/O costs may not be negligible. Performance comparisons incorporating I/O costs as well as distance computations remain for future research.

In Section 6 we discussed some heuristics for choosing better vantage points, and demonstrated empirically that these heuristics improve the performance of mvp-trees. However, selection of better vantage points at construction time is still an open problem, especially for metric spaces where data distribution is not known a priori and no geometry of the data space can be assumed. It is especially important for the mvp-trees that employ a large number of vantage points in the internal nodes, since the vantage points in a node are used to partition the data space in a hierarchical manner and each vantage point (except the first one in every node) is used to partition a multiple number of regions.

APPENDIX

Below we show the correctness of the search algorithm for vp-trees.

Let Q be the query object, r be the query range, S_v the vantage point of a node that we visit during the search, and M the median distance value for the same node. We have to show that

if $d(Q, S_v) + r < M$, then we do not have to search the right branch. (I)

if $d(Q, S_v) - r > M$, then we do not have to search the left branch. (II)

For (I), let X denote any data object indexed in the right branch, i.e.,

$d(X, S_v) \geq M$	(1)	
$M > d(Q, S_v) + r$	(2)	(hypothesis)
$d(Q, S_v) + d(Q, X) \geq d(X, S_v)$	(3)	(triangle inequality)
$d(Q, X) > r$	(4)	(summation of (1), (2), and (3))

Because of (4), X cannot be in the query result, which means that we do not have to check any object in the right branch.

For (I), Let Y denote any data object indexed in the left branch, i.e.,

$M \geq d(Y, S_v)$	(5)	
$d(Q, S_v) - r > M$	(6)	(hypothesis)
$d(Y, S_v) + d(Q, Y) \geq d(Q, S_v)$	(7)	(triangle inequality)
$d(Q, Y) > r$	(8)	(summation of (5), (6), and (7))

Because of (8), Y cannot be in the query result, which means that we do not have to check any object in the left branch.

REFERENCES

AGRAWAL, R., FALOUTSOS, C., AND SWAMI, A. 1993. Efficient similarity search in sequence databases. In *Proceedings of the Conference on FODO*

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

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 1990 ACM SIGMOD International Conference on Management of Data* (SIGMOD '90, Atlantic City, NJ, May 23–25, 1990), H. Garcia-Molina, Ed. ACM Press, New York, NY, 322–331.

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 '97, Tucson, AZ, May 12–14, 1997), A. Mendelzon and Z. M. Özsoyoglu, Eds. ACM Press, New York, NY, 78–86.

BERCHTOLD, S., KEIM, D. A., AND KRIEGEL, H.-P. 1996. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd International Conference on Very Large Data Bases* (VLDB '96, Mumbai, India, Sept.) 28–39.

BOZKAYA, T. AND OZSOYOGLU, M. 1997. Distance-based indexing for high-dimensional metric spaces. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (SIGMOD '97, Tucson, AZ, May 13–15), J. M. Peckman, S. Ram, and M. Franklin, Eds. ACM Press, New York, NY, 357–368.

BRIN, S. 1995. Near neighbor search in large metric space. In *Proceedings of the 21st International Conference on Very Large Data Bases* (VLDB '95, Zurich, Sept.) 574–584.

CHIUEH, T. 1994. Content-based image indexing. In *Proceedings of the 20th International Conference on Very Large Data Bases* (VLDB'94, Santiago, Chile, Sept.) VLDB Endowment, Berkeley, CA, 582–593.

CIACCIA, P. AND PATELLA, M. 1998a. Bulk loading the M-tree. In *Proceedings of the Australasian Conference on Databases* (ADC, Perth, Australia)

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 '97, Athens, Greece, Aug.) 426–435.

CIACCIA, P., PATELLA, M., AND ZEZULA, P. 1998b. Processing complex similarity queries with distance-based access methods. In *Proceedings of the 6th International Conference on EDBT* (EDBT, Mar.)

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 '98, Seattle, WA, June 1–3, 1998), A. Mendelson and J. Paredaens, Eds. ACM Press, New York, NY, 59–68.

FALOUTSOS, C., BARBER, R., FLICKNER, M., HAFNER, J., NIBLACK, W., PETKOVIC, D., AND EQUITZ, W. 1994a. Efficient and effective querying by image content. *J. Intell. Inf. Syst. 3*, 3/4 (July 1994), 231–262.

FALOUTSOS, C., RANGANATHAN, M., AND MANOLOPOULOS, Y. 1994b. Fast subsequence matching in time-series databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (SIGMOD '94, Minneapolis, MN, May 24–27), R. T. Snodgrass and M. Winslett, Eds. ACM Press, New York, NY, 419–429.

GUTTMAN, A. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference on Management of Data* ACM Press, New York, NY, 47–57.

LIN, K-I., JAGADISH, H., AND FALOUTSOS, C. 1994. The TV-tree—An index structure for high dimensional data. *VLDB J. 3* (Oct.), 517–542.

OTTERMAN, M. 1992. Approximate matching with high dimensionality R-trees. Department of Computer Science, University of Maryland, College Park, MD.

ROUSSOPOULOS, N., KELLEY, S., AND VINCENT, F. 1995. Nearest neighbor queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (SIGMOD '95, San Jose, CA, May 23–25), M. Carey and D. Schneider, Eds. ACM Press, New York, NY, 71–79.

SAMET, H. 1990. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Series in Computer Science. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.

SELLIS, T., ROUSSOPOULOS, N., AND FALOUTSOS, C. 1987. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases* (Brighton, UK, Sept.) 71–79.

SHASHA, D. AND WANG, T.-L. 1990. New techniques for best-match retrieval. *ACM Trans. Inf. Syst. 8*, 2 (Apr. 1990), 140–158.

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

BAEZA-YATES, R., CUNTO, W., MANBER, U., AND WU, S. 1994. Proximity matching using fixed-queries trees. In *Proceedings of the Fifth Symposium on Combinatorial Pattern Matching* (LNCS 807, June) Springer-Verlag, New York, 198–212.

YIANNILOS, P. 1993. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* ACM Press, New York, NY, 311–321.

Received: February 1998; revised: July 1999; accepted: August 1999