INDEXING LARGE METRIC SPACES FOR SIMILARITY SEARCH QUERIES[1][2]

Tolga Bozkaya
Oracle Corporation
USA
tbozkaya@us.oracle.com

Meral Ozsoyoglu
Department of Computer Engineering & Science
Case Western Reserve University
ozsoy@ces.cwru.edu

Abstract

In many database applications, one of the common queries is to find 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 used is metric. In this paper, we consider using distance-based index structures for similarity queries on large metric spaces. We elaborate on the approach of using reference points (vantage points) to partition the data space into spherical shell-like regions in a hierarchical manner. We introduce the multi-vantage point tree structure (mvp-tree) that uses more than one vantage points to partition the space into spherical cuts at each level. In answering similarity based queries, the mvp-tree also utilizes the pre-computed (at construction time) distances between the data points and the vantage points.

We summarize the experiments to compare mvp-trees with vp-trees which have a similar partitioning strategy, but use only one vantage point at each level, and do not make use of the pre-computed distances. Empirical studies show that 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 done to see how varying the number of vantage points used in a node affects search performance, and how much performance gain is obtained by making use of pre-computed distances. The results show that, after all, it may be best to use a large number of vantage points in an internal node to end up with a single directory node, and keep as many of the pre-computed distances as possible to provide more efficient filtering during search operations. Finally, we provide some experimental results comparing mvp-trees with M-trees, which is a dynamic distance based index structure for metric domains.

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, etc.. In genetics, the concern is to find DNA or protein sequences that are similar in a genetic database. In time-series analysis, one 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 a specified criteria.

Similarity between images can be measured in a number of ways. Features such as shape, color, texture can be extracted from images in the database to be used as content information where the distance calculations will be based on. 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.

[1] A preliminary version of this paper ([BO97]) appeared in ACM-SIGMOD 1997

[2] This research is partially supported by the National Science Foundation grant IRI 92-24660, and the National Science Foundation FAW award IRI-90-24152

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 do filtering of 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 to improve the speed in answering them. This is usually done by employing techniques and index structures to filter out distant (non-similar) data items quickly, avoiding expensive distance computations for each of them.

The 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 would pick the most semantically related images 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 simply by extracting features in the image. The best an image database can do is to 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 partly justified since our target applications are the ones where the distance computations are very expensive. In such applications, the distance computations measure also reflects the I/O costs (or other costs, such as network costs) to a degree as a distance computation requires the retrieval of a database object from secondary memory (it does not reflect the I/O operations required by the index structure though). 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 the performance comparisons of mvp-trees with other access structures for the general case where I/O costs can not be neglected remains to be done as future research.

We introduce the mvp-tree (multi-vantage point tree) as a general solution to the problem of answering similarity based queries efficiently for high-dimensional metric spaces. The mvp-tree is similar to the vp-tree (vantage point tree) [Uhl91] in the sense that both structures use relative distances from a vantage point to partition the domain space. In vp-trees, at every node of the 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. Then, these points are 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 keep 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 made are between the query point and the vantage points. Because of using more than one vantage point in a node, the mvp-tree has less 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 to 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 and mvp-trees to demonstrate mvp-trees' efficiency. 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 20-dimensional Euclidean vectors showed that mvp-trees require 40% to 80% less distance computations compared to vp-trees for small query ranges. For higher query ranges, the percentagewise difference decrease gradually, yet mvp-trees still perform better, making up to 30% less distance computations for the largest query ranges used.

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 on (and therefore had rather shallow trees), and the mvp-trees performed up to 20-30% less 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 of choosing 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 have 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 presents 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 having 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. Note that, for a metric space, no geometric information can be utilized 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 one asks for all data objects that are within some specified distance from a given query object. These queries require retrieval of *near neighbors* of the query object. The formal definition for this type of queries is as follows:

Near Neighbor Query: From a given set of data objects $X = \{X_1, X_2, ..., X_n\}$ 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 will be $\{ X_i \mid X_i \in X \text{ and } d(X_i,Y) \leq r \}$. 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 closest object 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 formulation of all these queries are similar to the definition of 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 approach 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 ([Sam89]) such as R-trees (and its variations) [Gut84, SRF87, BKSS90] can be used effectively to answer similarity queries. In such cases, a near neighbor search query would ask for all the objects in (or that intersects) 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 [RKV95], 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 [Ott92] 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 using 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 [AFS93, FRM94], and images [FEF+94].

Although it is possible to make use of general transformations such as DFT, Karhunen-Loeve [Fuk72] for any Euclidean domain, it is also possible to come up with application-specific distance transformations. In the QBIC (Query By Image Content) system [FEF+94], color content of images was used to compute similarity between images. The difference between the color contents of two images are computed from their color histograms. Computation of a distance between the color histograms of two images is quite expensive as the color histograms are high-dimensional (the number of different colors is generally 64 or 256) vectors, and also *crosstalk* (as some colors are similar) between colors have to be considered. To increase speed in color distance computation, QBIC keeps an index on average color of images. The average color of an image is a 3-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 color contents 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 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 each dimension are uncorrelated for 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 directly used for high-dimensional spatial domains as the conventional distance functions (such as Euclidean, or any L_p distance) defined on these domains are metric. Sequence matching, time-series analysis, image databases are some example applications having such domains. Distance based techniques are also applicable for domains where the data is non-spatial (that is, data objects can not be mapped to points in a multi-dimensional 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 on efficiently answering similarity search queries in different contexts. Burkhard & Keller [BK73] suggested the use of three different techniques for the problem of 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 multi-way 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. The keys that are of the same distance from that key get into the same group. Note that this is possible since 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 [BK73] partitions the data space is into a number of sets of keys. For each set, a *center* key is arbitrarily picked, 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 multi-way 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 [BK73] 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 the representative keys to be the ones that appear in as many *cliques* as possible.

In another approach, Shasha and Wang [SW90] suggested using pre-computed distances between data elements to efficiently answer similarity search queries. The aim is to minimize the number of distance computations as much as possible, as 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 [SW90], a table of size $O(n^2)$ keeps the distances between data objects if they are pre-computed. The other pairwise distances are estimated (by specifying an interval) by making use of the pre-computed distances. The technique of storing and using pre-computed distances may be effective for data domains with small cardinality, however, space requirements and search complexity becomes overwhelming for larger domains.

Uhlmann introduced [Uhl91] two hierarchical index structures for similarity search. The first one 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 as the *ball decomposition* in the paper, is similar to the first method presented in [BK73]. 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 multi-way tree for larger fanouts. In [Yia93], Yiannilos provided some analytical results on vp-trees, and suggested ways to pick better vantage points. In [Chi94], Chiueh 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 (*generalized hyperplane tree)* structure was also introduced in [Uhl91]. 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 [BCMW94]. The main difference from the vp-tree is that 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 given in [BK73]. 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 the FQ-trees is also given in [BCMW94]. The idea of using a single reference point for all nodes in the same level is an interesting idea. We use a similar technique in the design of mvp-trees.

The GNAT (*Geometric Near-Neighbor Access Tree*) structure [Bri95] 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 are 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 distance from the points in the data sets of other *split points* are recorded. The tree is recursively built for each data set at the next level. The number of *split points, k,* is parameterized and is 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. introduced the M-tree structure [CPZ97], which differs from the other distance-based index structures by being able to handle dynamic operations. The M-tree is constructed bottom-up (in contrast to the other structures such as vp-tree, GNAT, gh-tree, that are constructed top-down), and it can handle dynamic operations with reasonable costs without requiring periodical restructuring. An M-tree stores a given set of objects $\{o_1, .., o_2\}$ into fixed-size leaf nodes, which correspond to sphere-like regions of the metric space. Each leaf node entry contains a the id of a data object, its feature values (used in a distance computation), and its distance from a routing object that 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, off course), 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 [CPZ97, CP98, CPZ98a, CPZ98b]. An analytical cost model based on distance distribution of the objects is derived in [CPZ98b] for M-trees. Evaluation of complex similarity queries (with multiple similarity predicates) using M-trees are discussed in [CPZ98a]. [CP98] provides 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 forming 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. Left branch of the node indexes the points whose distances from S_v are less than or equal to M, and right branch of the node indexes the points whose distances from S_v are greater than or equal to M. In leaf nodes, instead of pointers to the left and right branches, references to the data points are kept.

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 therefore can 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 are 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)
```

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 the Appendix).

Generalizing binary vp-trees into multi-way vp-trees.

The binary vp-tree can be easily generalized into a multi-way 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 will 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.

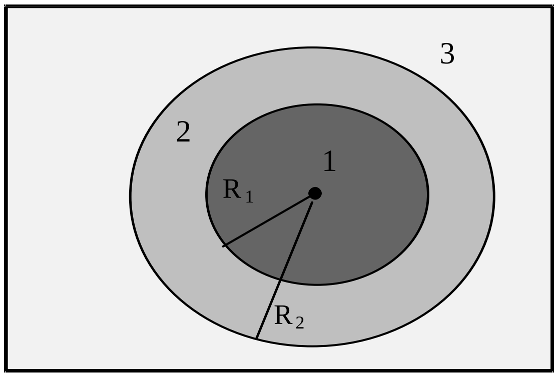


Figure 3.1. The root level partitioning of a vp-tree with branching factor 3. The three different regions are labeled 1, 2, 3, and they are all shaded differently.

However, there is one problem with high-order vp-trees. A vp-tree partitions the data space into spherical cuts (see Figure 3.1), and those spherical cuts become too thin for high-dimensional domains, leading the search regions to intersect with many of them, and therefore leading 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 3 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 3.1. The three different regions are colored differently and labeled as 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. Because of 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, $R_2 = 1.007\ R_1$.

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. Multi-vantage-point trees

In this section, we present the mvp-tree (*multi vantage 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 points 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 4.1, 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 as well be kept in the parent. This way, we would be keeping more than one vantage points 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 points in the nodes for higher utilization.

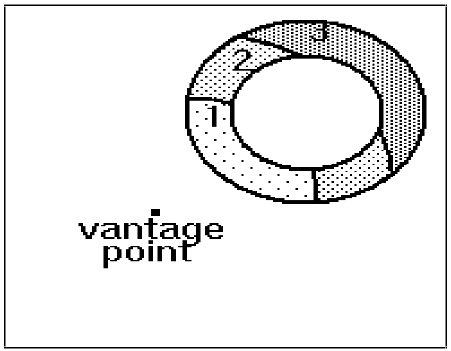
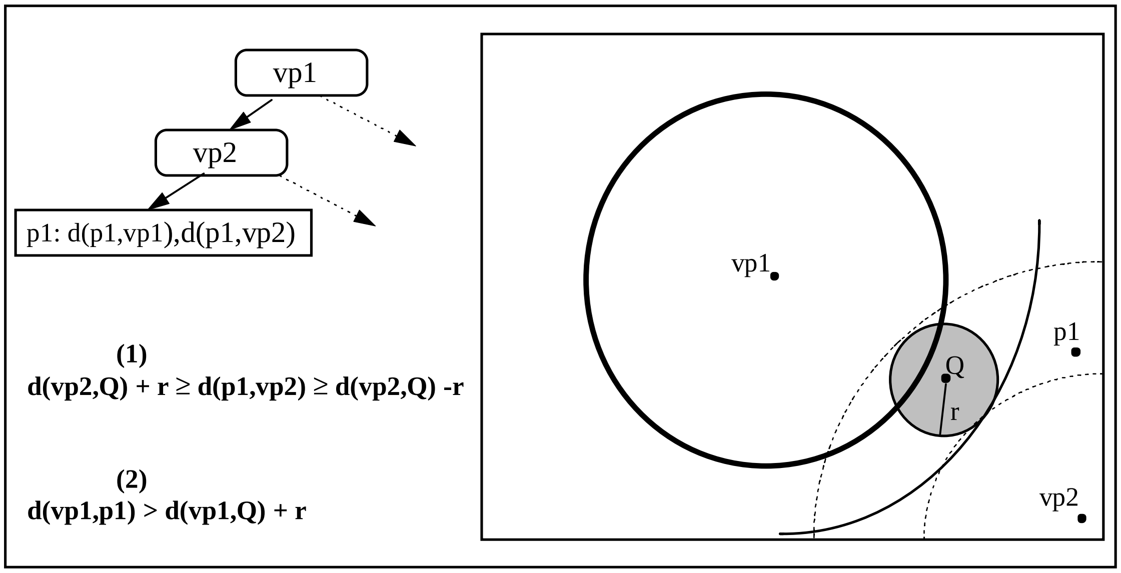


Figure 4.1. Partitioning a spherical shell-like region using a vantage point from outside.

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 which keeps that data point. So for each data point, ($\log_m n$) distance computations (for a 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 4.2 illustrates how the pre-computed 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 4.2. 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).



(1)

$$d(vp2,Q) + r \geq d(p1,vp2) \geq d(vp2,Q) - r$$

(2)

$$d(vp1,p1) > d(vp1,Q) + r$$

Figure 4.2. A vp-tree decomposition where the use of pre-computed (at construction time) distances make it possible to filter out a distant point (p1).

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 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 non-leaf 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 3 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 as S_{v1}) divides the space into two parts, and the second vantage point (referred 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 these 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 the partitions with respect to the second vantage point.

In a leaf node, exact distances between data points in that leaf and vantage points of that leaf are kept. $D_1[i]$ and $D_2[i]$ (i=1, 2, .. 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.

For each data point x in the leaves, the array x.PATH[p] keeps the pre-computed distances between the data point x and the first p 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 4.3 shows the structure of internal and leaf nodes of a binary mvp-tree.

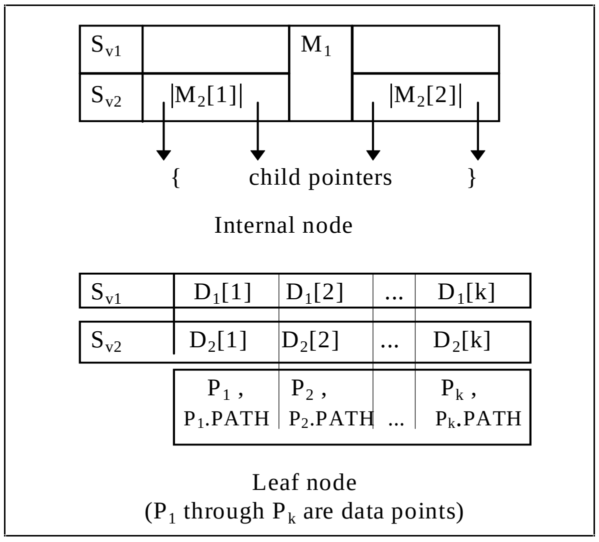


Figure 4.3. Node structure for a binary mvp-tree.

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

Construction of 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 above in Figure 4.3. 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| ≤ $k$+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 ≤ p) $S_i$.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 ($level < p$) $S_j.PATH[level+1] = d(S_j, S_{v2})$
    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 2 vantage points can be kept in one node. We will 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 choose 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. That 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 the 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. That is actually twice the number of nodes in the mvp-tree as 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* pre-computed 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 .. p
            ( PATH[i] - $r \le S_i$.PATH[i] $\le$ PATH[i] + $r$ ) holds,
            then compute d(Q, $S_i$). If d(Q, $S_i$) $\le r$, then $S_i$ is in the answer set.

3) Else if the current node is an internal node
   3.1) If ($level \le p$) then PATH[$level$] = d(Q, $S_{v1}$).
        If ( $level<p$) then PATH[$level$+1] = d(Q, $S_{v2}$).
   3.2) If d(Q, $S_{v1}$) + $r \le M_1$ then
        if d(Q, $S_{v2}$) + $r \le M_2$[1] then recursively search the first branch with $level=level+2$
        if d(Q, $S_{v2}$) - $r \ge M_2$[1] then recursively search the second branch with $level=level+2$
   3.3) If d(Q, $S_{v1}$) - $r \ge M_1$ then
        if d(Q, $S_{v2}$) + $r \le M_2$[2] then recursively search the third branch with $level=level+2$
        if d(Q, $S_{v2}$) - $r \ge M_2$[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, query range, and the 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). The search algorithm, in this case, 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.

A. High-Dimensional Euclidean Vectors:

We used two sets of Euclidean vectors with two different distributions. Euclidean distance metric was used as the distance metric for all the experiments. Note that 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 [LJF94], X-trees [BKP96] 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 [CPZ98b]. So, although mvp-trees and vp-trees are not specifically designed for Euclidean Vectors we experimented with, using them in the experiments provides us a convenient test bed for relating search performances with different distance distributions.

Our first set of experiments are conducted on uniformly distributed Euclidean vectors. We used 50,000 uniformly distributed vectors in 10 and 20-dimensional Euclidean spaces. For these sets, all vectors were chosen randomly from the 10 or 20-dimensional unit hypercube. The pairwise distance distribution of these uniformly distributed vectors for 10 and 20 dimensions are shown in Figure 5.1. The distance values are sampled at intervals of length 0.01.

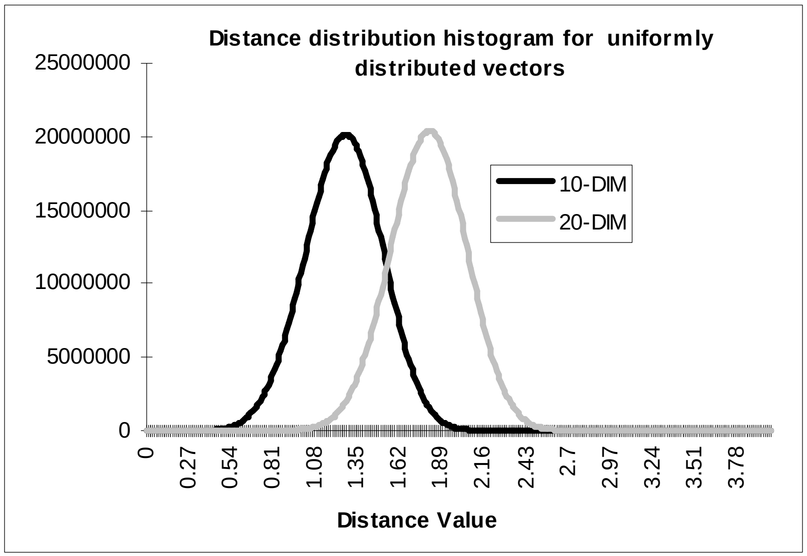


Figure 5.1. Distance distribution for uniformly distributed vectors in 10 and 20-dimensional Euclidean Space.
(Y axis shows the number of data object pairs that have the corresponding distance value)
(The distance values are sampled at intervals of length 0.01)

Consider the distance distribution for the 20-dimensional case. The vectors are mostly far away from each other and their distance distribution is similar to a sharp Gaussian curve where the distances between any two points fall mostly within the interval [1, 2.5] concentrating around the midpoint 1.75. As a result, the vantage points (in both vp-trees and mvp-trees) always partition the space into thin spherical shells and there is always a large, void spherical region in the center that does not accommodate any data points. This distribution makes both structures (or any other hierarchical method) not very effective in queries having values of *r* (similarity ranges) larger than 0.5, although higher *r* values may be quite reasonable for legitimate similarity queries. As a matter of fact, when the query points are also generated in the same way, it is practically impossible to find a near neighbor of a query point for similarity ranges less than 0.5, as there are very few randomly generated pairs of points that are closer than 0.5 to each other. This means that the selectivity of queries with similarity ranges of 0.5 and below are too low, mostly close to zero, if not equal to it. However, as the distance distribution is very narrow, it becomes very hard to trim the search for larger similarity ranges, which leads to only slight improvements over the sequential search. This is why we also experimented with 10 dimensional random vectors. In 10-dimensions, the distance distribution has the same pattern, but the data points are not as far from each other where the pairwise distances fall mostly within the interval [0.5, 2.0] concentrating around the midpoint 1.25. The selectivity of queries with small query ranges (0.5 and smaller) are 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 are presented in section 5.2.

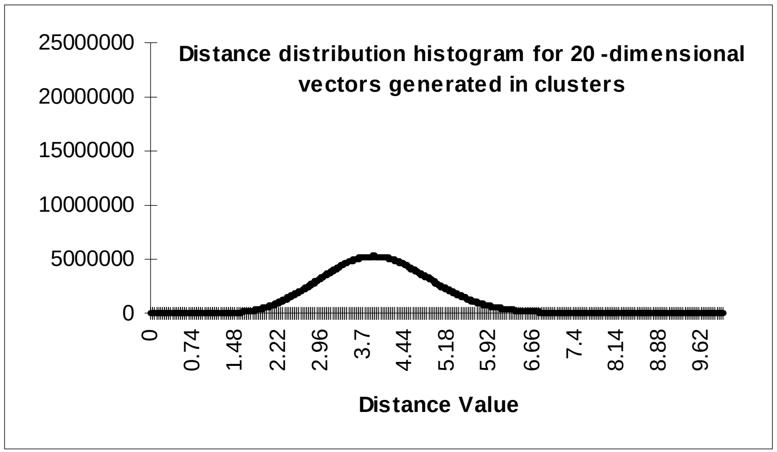


Figure 5.2. Distance distribution for 20-dimensional Euclidean vectors generated in clusters.

(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)

Another set of experiments are conducted on 20-dimensional Euclidean vectors that are generated in clusters of equal size. The clusters are generated as follows. First, a random vector is generated from the 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 (such as 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 other), and many are outside of the unit hypercube. We call these groups of points as 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 5.2, 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 can quickly realize 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 (that are generated in the same way) as we were able 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.

B. 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 do 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 by $D_p(X,Y)$) is calculated as follows:

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

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.

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 those images are stored on disk using around 61K per image (images are in binary PGM format using one byte per pixel).

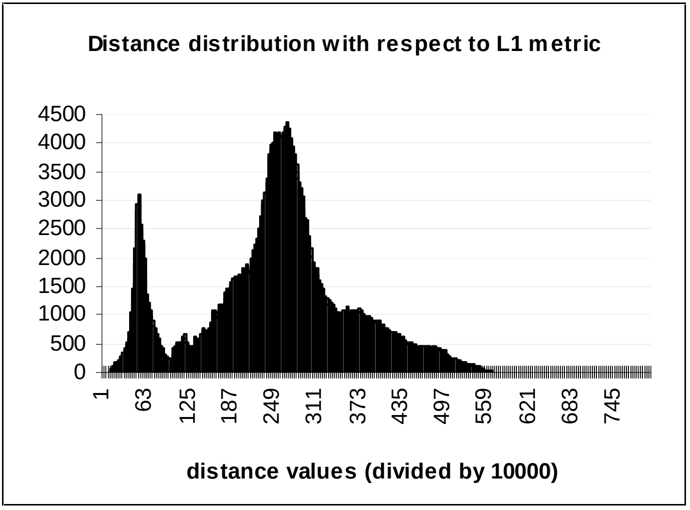


Figure 5.3. Distance histogram for images when L1 metric is used.

The distance distributions of the MRI images for L_1 and L_2 metrics are shown in the two histograms in Figures 5.3 and 5.4. 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 normalized by 100 similarly. After the normalization, the distance values are sampled at intervals of length 1 in each case.

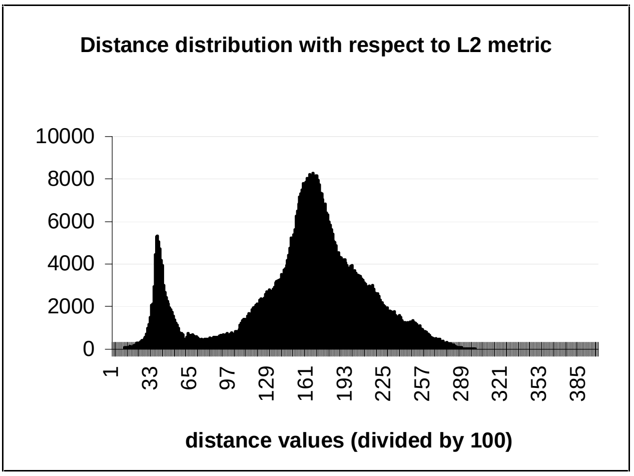


Figure 5.4. Distance histogram for images when L2 metric is used.

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 L_1 metric is used, a tolerance factor (r) around 500,000 is quite meaningful, where 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 would be assigned a weight that would be used to multiply intensity differences of two images at that pixel position when 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

A. 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, are the four structures. 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 have 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 that we display the results for have different k (leaf capacity) values to see how it effects the search efficiency. We do not take into account how the leaf and internal nodes of an mvpt-tree are paged as we do not consider the I/O behavior in this study. In the following figures, the mvp-tree with (m,k,p) values (3,9,5) is referred as mvpt(3,9) and the other mvp-tree is referred as mvpt(3,80) since both trees have the same p values. The vp-trees of order 2 and 3 are referred 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 Figures 5.5 and 5.6 are obtained by taking the average of 4 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.

Figure 5.5 shows the performance results for 20-dimensional uniformly distributed Euclidean data set. As shown in the figure, both mvp-trees perform much better than the vp-trees, and vpt(2) is slightly better than (around 10%) vpt(3). mvpt(3,9) makes around 40% less number of distance computations compared to the vpt(2) for small query ranges. The performance gap shrinks slowly when the query range increases, where mvpt(3,9) makes 20% less distance computations for the query range of 0.5. mvpt(3,80) performs much better, and needs around 80% to 65% percent less number of distance calculations compared to vpt(2) for small ranges (0.2 to 0.3). For query ranges of 0.4 and 0.5, mvpt(3,80) makes 45% and 30% (respectively) less distance computations compared to vpt(2). For higher query ranges, the gain in efficiency decreases, which is due to the fact that the data points in the domain are themselves quite distant from each other, making it harder to filter out distant points for the search operations.

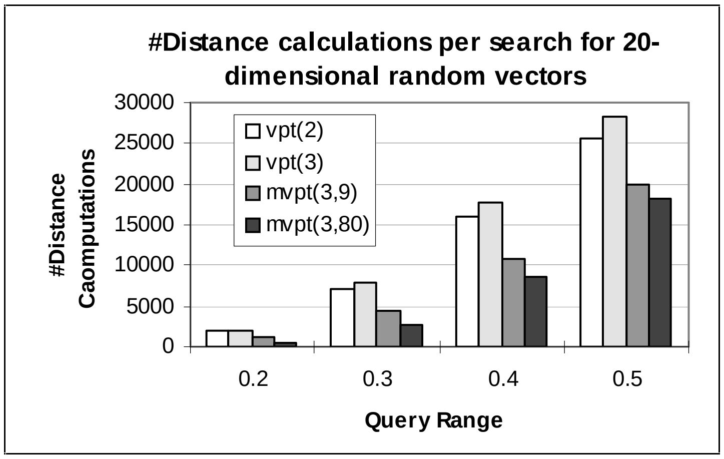


Figure 5.5. Search performances of vp and mvp trees for 20-dimensional randomly generated Euclidean vectors.

As we have said in Section 5.1, the selectivity of the queries for 20-dimensional vectors are mostly zero. This is not the case for the experiments on 10-dimensional Euclidean vectors. The selectivity versus query range information is given in Table 5.1, where total number of near neighbors found in 100 queries is shown for query ranges 0.2 thru 0.5. The results are by averaging four different runs with different seeds. The query points are also distributed uniformly.

Query Range (r)

	0.2	**0.3**	**0.4**	**0.5**
# Near neighbors found	1	38	503	3431

Table 5.1. Total number of near neighbors found in experiments with 10-dimensional random vectors.

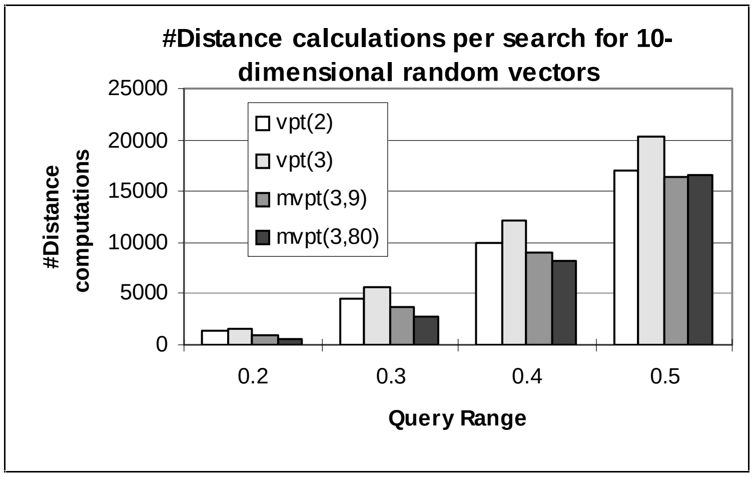


Figure 5.6. Search performances of vp and mvp trees for 10-dimensional randomly generated Euclidean vectors.

Figure 5.6 shows the performance results for 10-dimensional uniformly distributed Euclidean vectors. Again, vpt(2) was superior to vpt(3) for all query ranges. Compared to vpt(2), mvpt(3,9) performed 25%, 20%, 10%, and 4% less 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% less distance computations for the same query ranges. We see that the performances are very close for high query ranges. This is because points are much closer to each other (see Figure 5.1) in 10-dimensional space compared to the 20-dimensional space, which makes it harder to filter out non-near neighbor points during the search for all structures. This can also be observed from Table 5.1, where the selectivity jumps very quickly for high query ranges.

For Euclidean vectors generated in clusters, we use two different query sets. In the first query set that we will refer to as Q_1, all query objects are generated randomly from the 20-dimensional unit hypercube, as it was done for the previous test case. In the second query set, that we will refer to as Q_2, query objects were generated by slightly altering randomly chosen data objects, so that we are guaranteed to find some near neighbors during query evaluation. Note that this set of query points actually conform to the data distribution as the data points are generated in the same way. The experimental results for the two query sets are as shown in Figures 5.7 and 5.8. Each query set contains 100 objects, and results are obtained by averaging two different runs (with different seeds). Table 5.2 shows total number of near neighbors found during the evaluation of these queries for different query ranges.

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 (**Q2**)	2	95	101	132	246	344	464	647	875

Table 5.2. Total number of near neighbors found for query sets Q_1 and Q_2 with respect to different query ranges (For Euclidean vectors generated in clusters).

Figure 5.7 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 this data and query sets, vpt(3) performs slightly better than vpt(2) (around 10%). The mvp-trees perform again much better than vp-trees. Mvpt(3,80) makes around 70% - 80% less number of distance computations than vpt(3) for small query ranges (up to 0.4), where mvpt(3,9) makes around 45% - 50% less number of 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% less distance computations compared to vpt(3) and mvpt(3,9) requires 20% less.

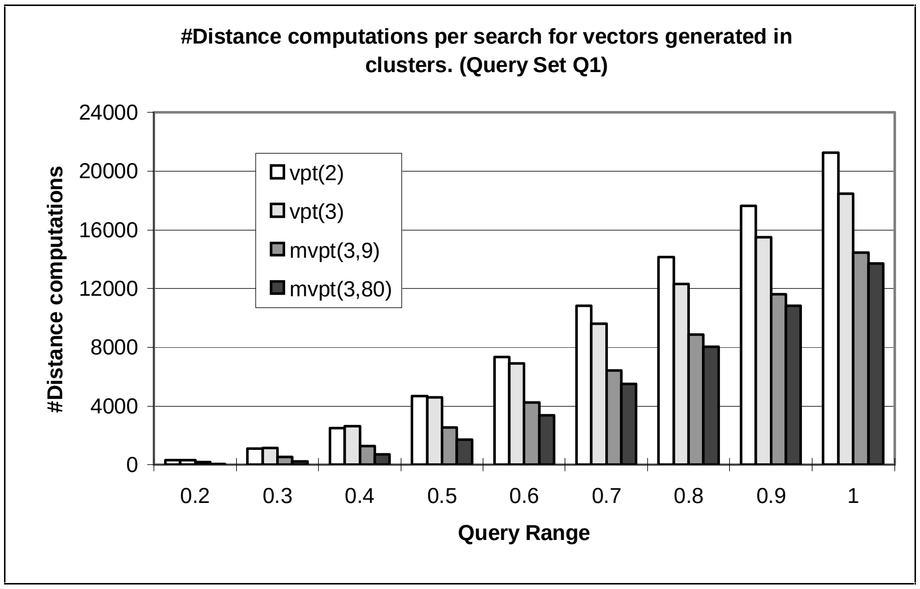


Figure 5.7. Search performances of vp and mvp trees for Euclidean vectors generated in clusters. (Query set Q_1).

Figure 5.8 shows the performance results for the data set where the vectors are generated in clusters (query set Q_2). For this data set, vpt(3) and vpt(2) performed very close to each other. Other than that, relative performances of the index structures are very similar to the previous case, although the absolute number of distance computations are less than it was for randomly generated query points (query set Q_1). Again, mvpt(3.80) starts making around 70% less 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% less distance computations compared to vpt(3).

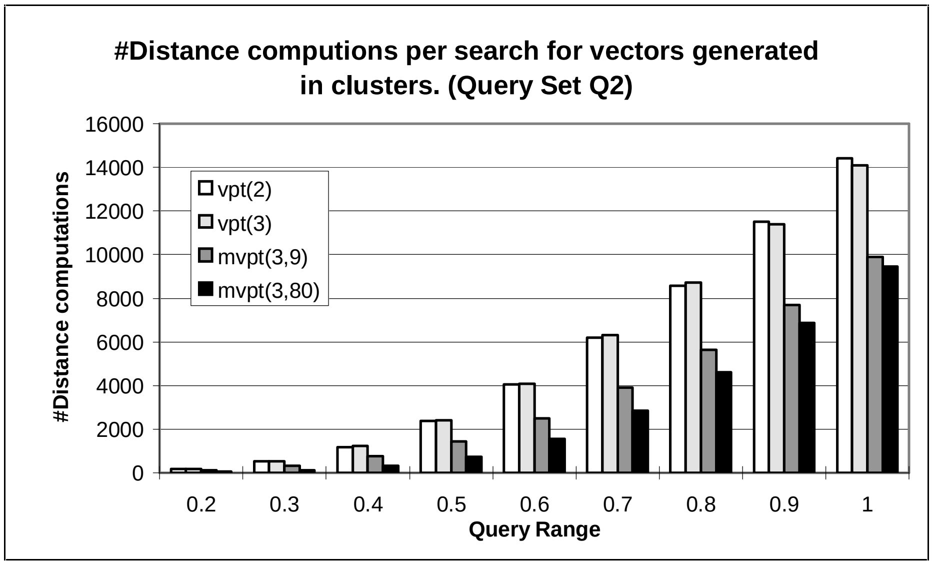


Figure 5.8. Search performances of vp and mvp trees for Euclidean vectors generated in clusters. (Query set Q_2).

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 perform a considerable improvement over vp-trees, especially for small query ranges (up to 80%). The efficiency gain (in terms of number of distance computations made) is smaller for larger query ranges, but still significant (around 30% for 20-dimensional vectors).

B. Gray-Level MRI Images:

The experimental results for the similarity search performances of vp and mvp trees on MRI images are given in Figures 5.9 and 5.10. 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 as vpt(2) and vpt(3). All the mvp-trees have the same *p* parameter 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 one 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 mvp-tree structures because of 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 5.3 and 5.4 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.

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

Table 5.3. Average number of near neighbors found for images when L_1 metric is used

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

Table 5.4 Average number of near neighbors found for images when L_2 metric is used

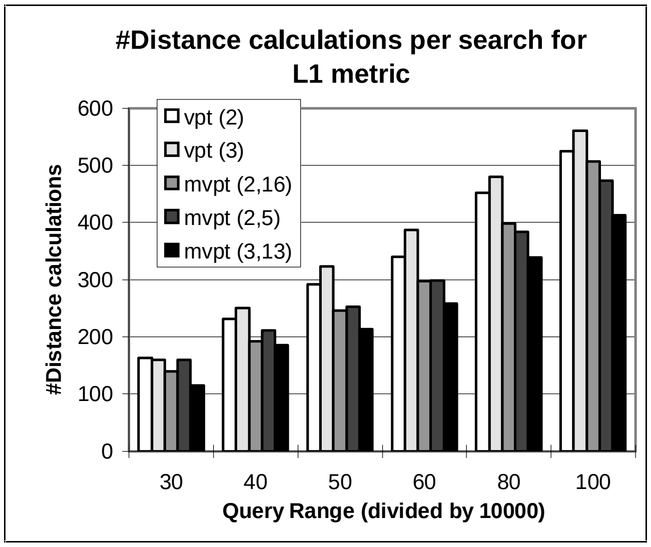


Figure 5.9. Similarity search performances of vp and mvp trees on MRI images when L_1 metric is used for distance computations.

The search performances of the 5 structures we tested (two vp-trees and three mvp-trees) for L_1 metric are shown in Figure 5.9. The query range values shown are normalized by 10,000 as it was done for Figure 5.3. Between 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% less number of distance computations compared to vpt(2).

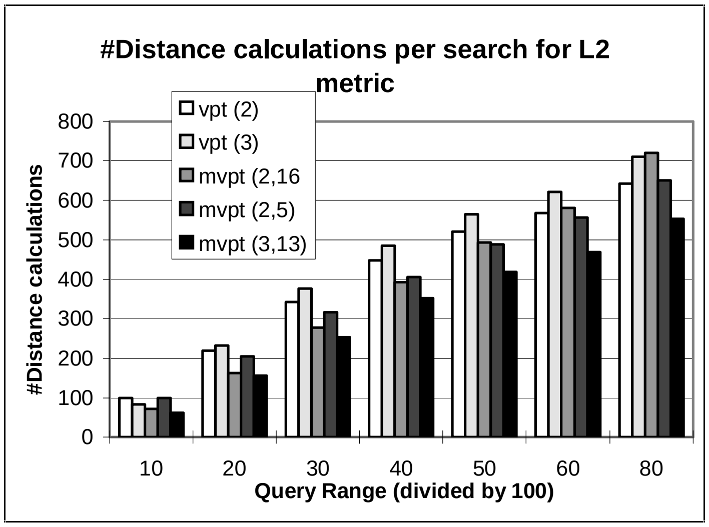


Figure 5.10. Similarity search performances of vp and mvp trees on MRI images when L_2 metric is used for distance computations.

The search performances for the L_2 metric are shown in Figure 5.10. The query range values shown are normalized by 100 as it was done for Figure 5.4. Similar to the case when 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, because the random function that is 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% less 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 [Yia93], Yiannilos 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 have any idea about where the corners of the data space are. The only information one 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 distributions for the corner points are more *flat* than the distance distributions for the center points. Figure 6.1 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 one is a corner point. As can be easily seen from the figure, the distribution for the center point is sharper, which means that using the center point as a vantage point would be less useful in trimming the search during similarity queries.

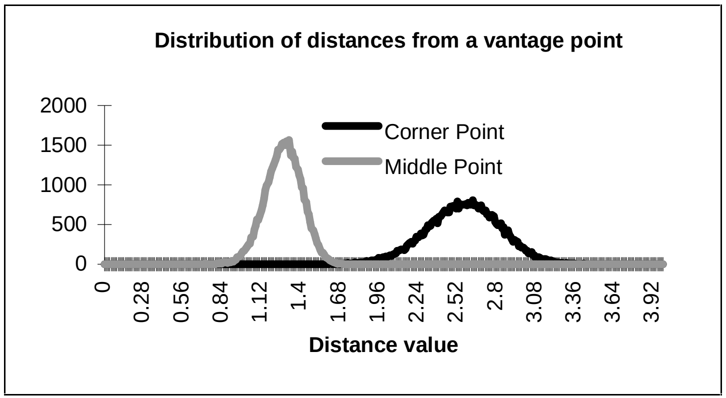


Figure 6.1. Distance distributions for two vantage points, one is the center of the 20-dimensional Euclidean hypercube (middle point), the other is one of the corners of the cube (corner point). The data points were distributed uniformly.

Most of the data points are far away from the corner points, which is a trivial fact that can also be observed from Figure 6.1. 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, however, it helps 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 which were also used in comparing vp-trees and mvp-trees. However, this time, the comparison is between mvp-trees that choose the first vantage point in any internal node randomly, and the mvp-trees that choose the first vantage point using the heuristic shown below. For both data sets (uniformly distributed vectors and vectors generated in clusters), we show the results only for randomly generated query objects (referred as Q_1 in section 5).

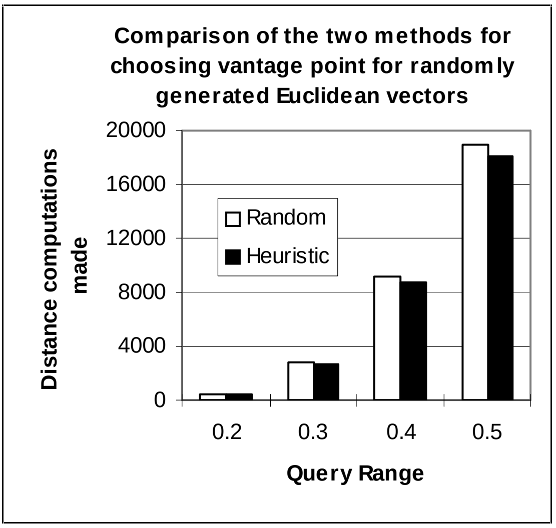


Figure 6.2. Query performance of mvpt(3,80) for the two different methods of choosing vantage points for uniformly distributed 20-dimensional Euclidean vectors.

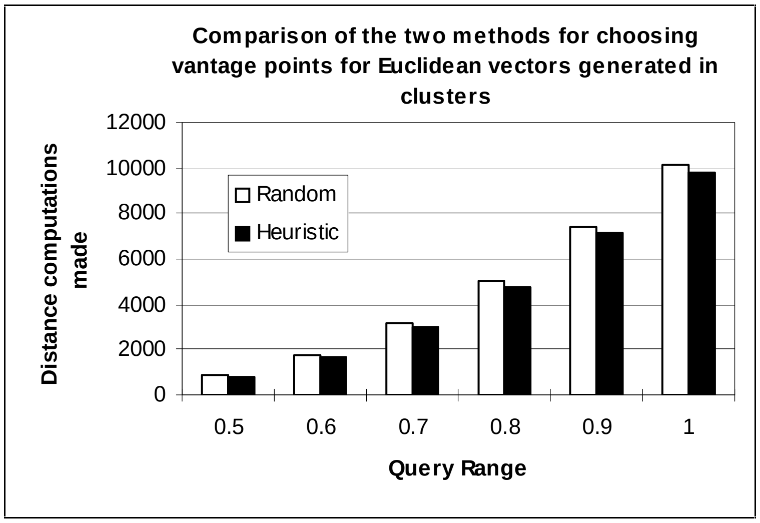


Figure 6.3. Query performance of mvpt(3,80) for the two different methods of choosing vantage points for 20-dimensional Euclidean vectors generated in clusters.

Figures 6.2 and 6.3 show the result of this comparison when an mvp-tree with parameters m=3, k=80, p=5 (mvpt(3,80)) is used. In both cases, the performance gain varied between %5 to %10, in terms of the average number of distance computations made in a query. Note that this performance gain comes at the expense of 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 that are seen along a path are not very close to each other. In mvp-trees, this would also make better utilization of the pre-computed 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 multi-vantage 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 easily modified 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 used 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* pre-computed distances for each data point. When the search proceeds down to a leaf node, only these *p* pre-computed 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 Construction of Generalized Mvp-trees

The vantage points in an internal node are selected in a similar way as 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 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 $\{R_{i,j} \mid i,j = 1,.. m\}$. Here, $R_{i,1}$ thru $R_{i,m}$ 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, as 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 $\{R_{i1,\ ...,iv} \mid i1,..iv = 1, ..., m\}$. The construction algorithm is given below.

Construction of an mvp-tree with parameters m, v, k, p

Given a finite set $S=\{S_1, ... ,S_n\}$ 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 used 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 $SR_1$, .. ,$SR_m$.
  3.5) let j=2 (j is just a loop variable)
```

```
3.6) While $j \leq v$ do
    3.6.1) Choose $S_{vj}$ to be the farthest point from $S_{v(j-1)}$ in $SR_{m,..,m}$ ((j-1) m's)
        3.6.2) Delete $S_{vj}$ from $SR_{m,..,m}$
    3.6.3) Calculate all $d(S_j, S_{vj})$ where $S_j \in SR_{i1,i2,...,i(j-1)}$ where i1,i2, ... ,i(j-1) =1, .., m
    3.6.4) if ($level \leq p$) then
        $S_i.PATH[level] = d(S_j, S_{vj})$
    3.6.5) Use these distances to partition each of the $SR_{i1,i2,...,i(j-1)}$ (i1,i2, ... ,i(j-1) =1, .., m) regions
        further into m more regions, creating $SR_{i1,i2,...,ij}$ (i1,i2, ..,ij =1, ...., m) . Record cutoff values.
    3.6.6) level=level+ 1;
           j=j+1;
3.7) Recursively create the mvpt-tree on each of the $m^v$ partitions, namely $SR_{i1,i2,...,iv}$ (i1,i2, .., iv =1, ..., 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 pre-computed 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. As 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 conform 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 pre-computed distance values kept in the leaf nodes. If all pre-computed 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 were 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 it was done in section 5. In section 8.1, we investigate the effect of using different number of vantage points in the internal nodes, and utilization of pre-computed 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[3], for comparison.

[3] The authors would like to thank Paolo Ciaccia and Marco Patella for providing the code of M-tree.

8.1 Tuning Mvp-tree Parameters

In the first set of experiments, we try to see the effects of changing the number of vantage points used in an internal node of an mvp-tree. Figures 8.1-4, show the results for 6 mvp-trees that have 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 8.1-4, these structures are referred by their (*m*,*v*,*k*,*p*) parameters. The parameter *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 will be 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 pre-computed distances are used in this structure during search time, and the leaf size is larger. We should also mention that 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.

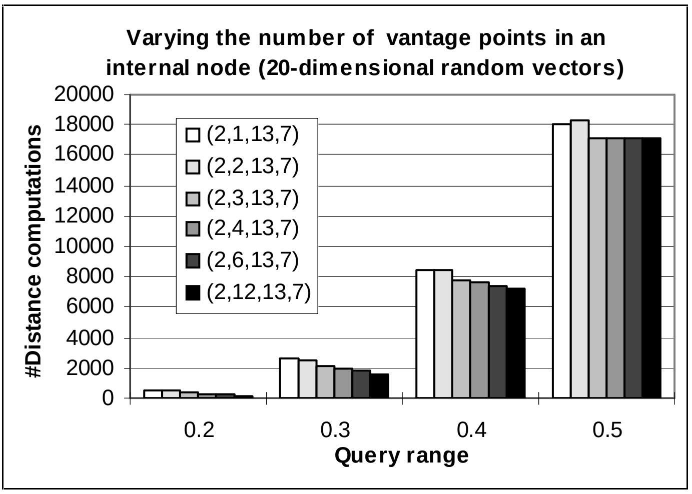


Figure 8.1. Performance of mvp-trees with different *v* values for 20-dimensional uniformly distributed Euclidean vectors.

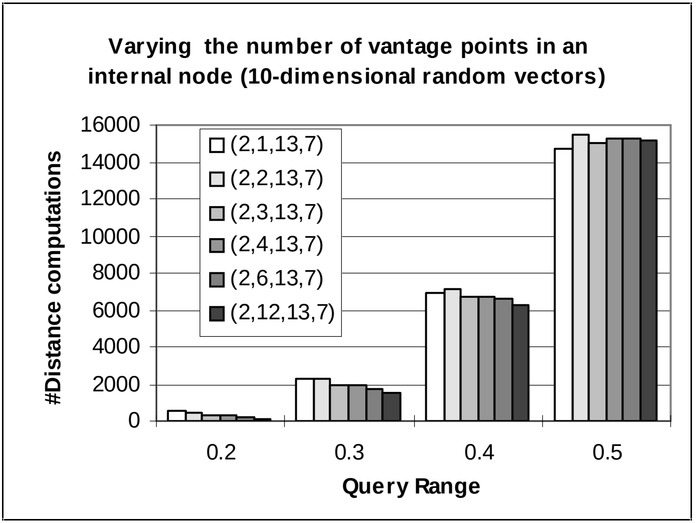


Figure 8.2. Performance of mvp-trees with different *v* values for 10-dimensional uniformly distributed Euclidean vectors.

The performance results shown in Figures 8.1-4 are quite interesting. For clustered vectors and query set Q1 (randomly generated query objects) (Figure 8.3), we see that mvpt(2,1,13,7) catches up with the other ones for moderate query ranges, and then actually performs slightly better for large query ranges, except for mvpt(2,12,13,7), which remains to be the best for all query ranges. This tells us that for large query ranges, when a lot of 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. For 20-dimensional random vectors (Figures 8.1), we observe a similar behavior; however, since we do not use very large query ranges, although mvp-trees with smaller *v* values close the performance gap (percentagewise) as the query ranges get larger, they do not perform better than mvp-trees with larger *v* values (except for query range 0.5 where mvpt(2,1,13,7) performs slightly better than mvpt(2,2,13,7)). The situation is similar in the case 10-dimensional random vectors are used, however, mvp-trees with smaller *v* values close the gap faster and perform around the same for the largest query range. For query set Q_2 on clustered vectors (Figure 8.4), mvp-trees with higher *v* values performed better for all query ranges. The percentagewise performance difference was quite high especially for small query ranges.

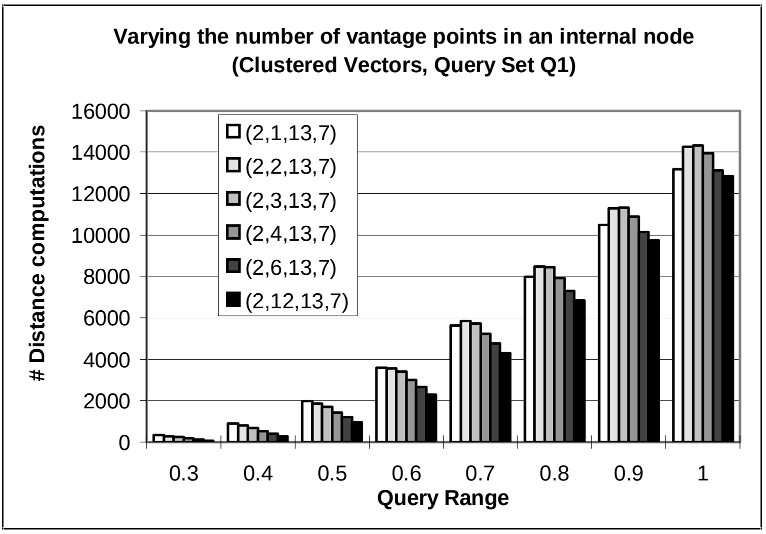


Figure 8.3. Performance of mvp-trees with different *v* values for Euclidean vectors generated in clusters. (Query Set Q_1).

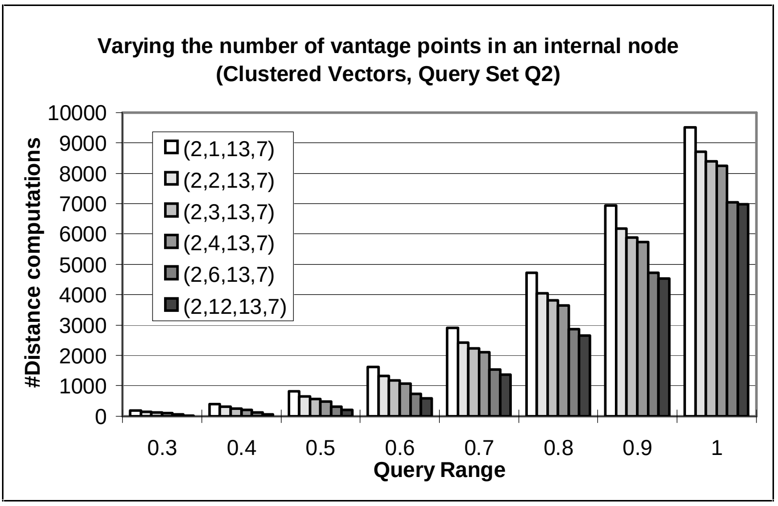


Figure 8.4. Performance of mvp-trees with different *v* values for Euclidean vectors generated in clusters. (Query Set Q_2).

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 shown in Figures 8.5-8. The parameter *p* is varied from 7 to 12 (the maximum).

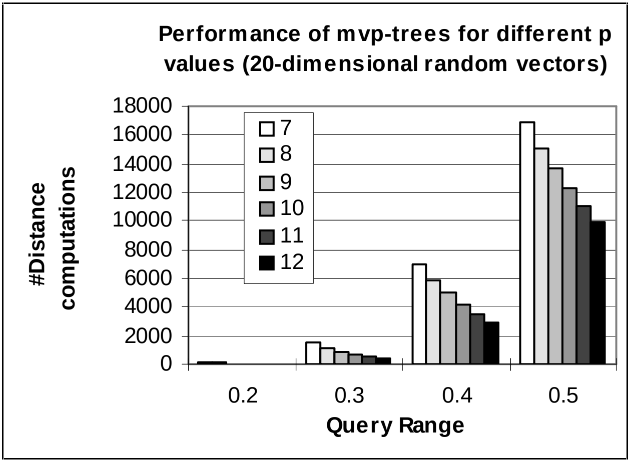


Figure 8.5. Performance of mvpt(2,12,13,*p*) for different *p* values using 20-dimensional uniformly distributed Euclidean vectors.

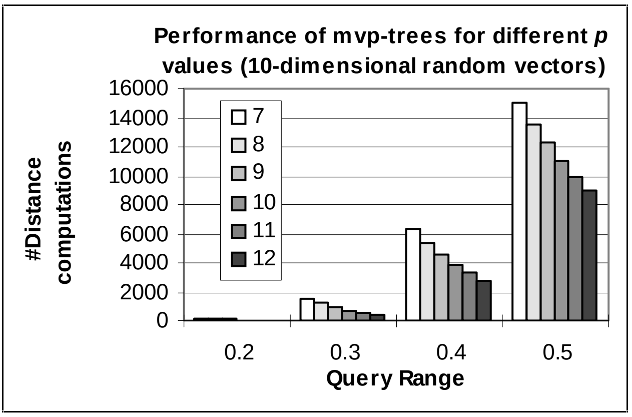


Figure 8.6. Performance of mvpt(2,12,13,*p*) for different *p* values using 10-dimensional uniformly distributed Euclidean vectors.

One can clearly observe that using higher *p* values improves the search performance significantly. To give an idea, in Figure 8.5, mvpt(2,12,13,12) performs 84% less 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 40% less distance computations. The performance gains were percentagewise similar for 10-dimensional random vectors as well (Figure 8.6). Similar behavior is also observed for the experiments with Euclidean vectors generated in clusters. For query set Q_1 (Figure 8.7), the performance difference ranges from 70% to 27% for query ranges 0.3 to 1.0. For query set Q_2 (Figure 8.8), 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.

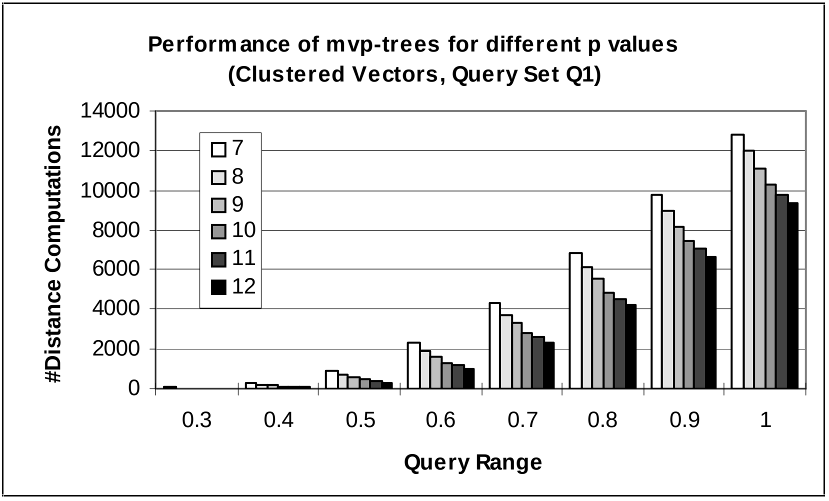


Figure 8.7. Performance of mvpt(2,12,13,*p*) for different *p* values using Euclidean vectors generated in clusters. (Query Set Q_1).

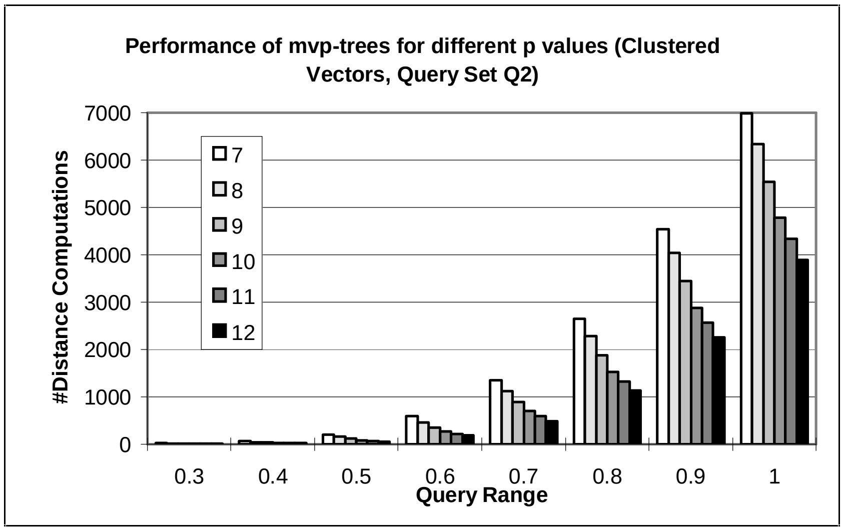


Figure 8.8. Performance of mvpt(2,12,13,*p*) for different *p* values using Euclidean vectors generated in clusters. (Query Set Q_2).

Our last set of experiments are done to investigate how changing the leaf size together with the number of vantage points in the internal nodes affects the performance. We experimented on six mvp-trees where each has only a single directory (internal) node, but each has a different number of vantage points in their directory nodes, 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, and 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, and it is 7. The performance results are shown in Figures 8.9-12 below.

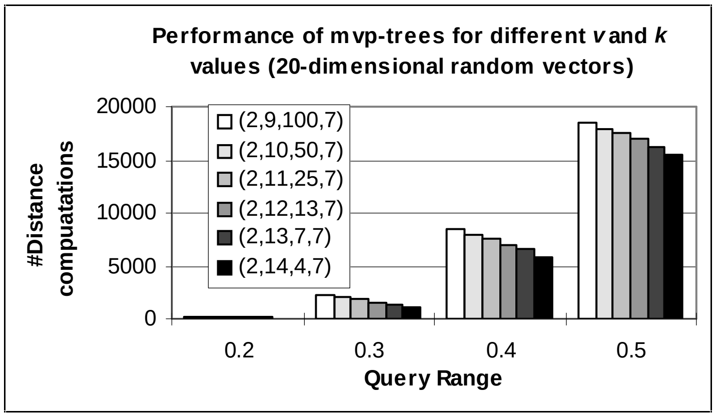


Figure 8.9. Performance of mvp-trees with different *v* and *k* parameters using 20-dimensional uniformly distributed Euclidean vectors.

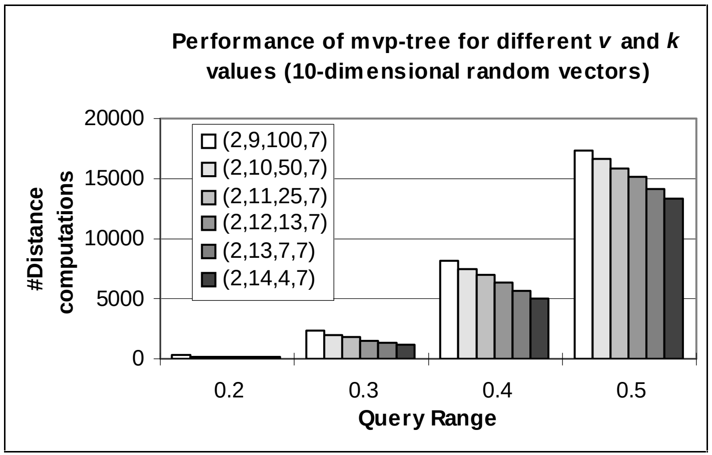


Figure 8.10. Performance of mvp-trees with different *v* and *k* parameters using 10-dimensional uniformly distributed Euclidean vectors.

From Figures 8.9-12, 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 pre-computed 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 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 pre-computed distances are kept for the data points in the leaves. Furthermore, any performance difference between these structures is simply because of the extra trimming that is done in the internal level.

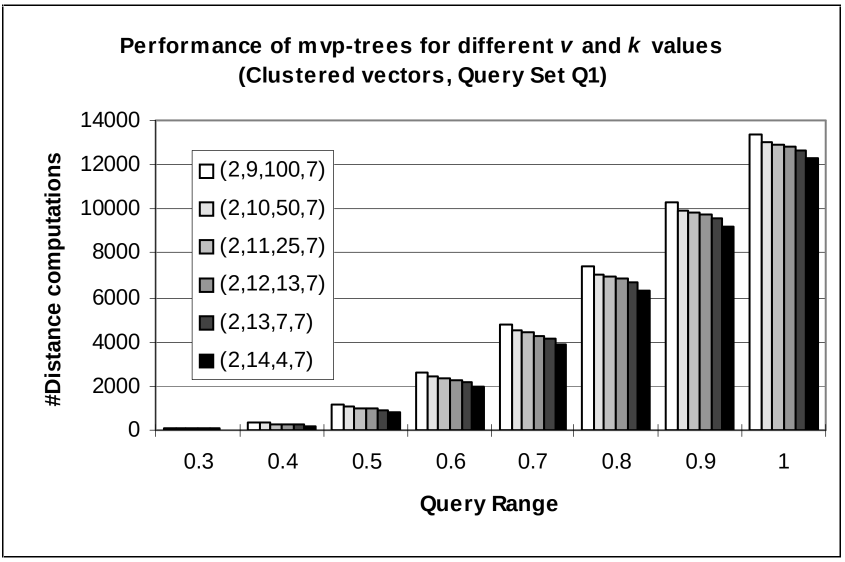


Figure 8.11. Performance of mvp-trees with different *v* and *k* parameters using Euclidean vectors generated in clusters. (Query Set Q_1).

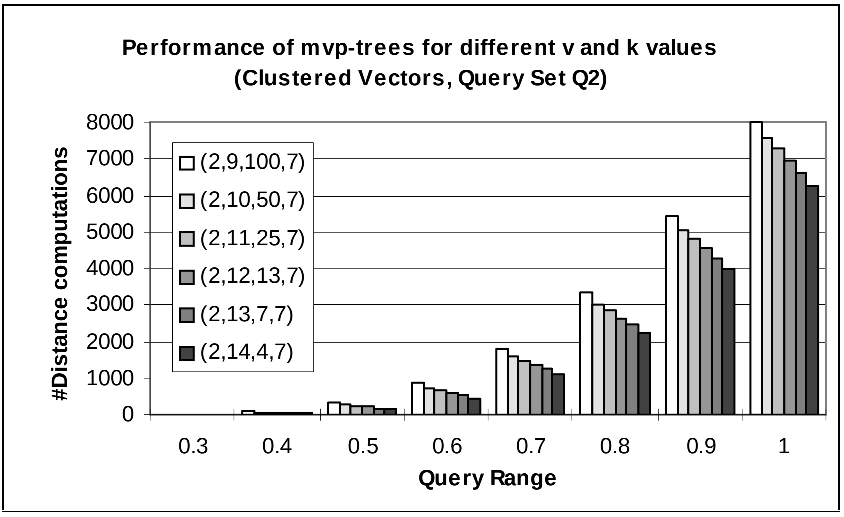


Figure 8.12. Performance of mvp-trees with different *v* and *k* parameters using Euclidean vectors generated in clusters. (Query Set Q_2).

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 17% less distance computations (performance difference decreasing for increasing query ranges) for 20 dimensional random vectors, in 65% to 25% for 10-dimensional random vectors, 44% to 9% less number of 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:

- The best performance is obtained by using only one internal node and a single set of vantage points as it can be seen from Figures 8.1-4, 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 pre-computed distances at search time clearly improves search performance in terms of the number of distance computations made. If all the pre-computed 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 have the further implication 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 are 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 to compare the M-tree structure [CPZ97] with mvp-trees in terms of the average number of distance computations made in similarity queries. We used the *BulkLoading* algorithm [CP98] in creating the M-tree. We remind that M-tree, unlike the other distance based index structures including mvp-trees, is a dynamic index structure, and it is created bottom up. It is designed as a paged index structure to minimize I/O operations required as well as distance computations. There are two parameters that are used to tune the M-tree for better performance, which are the minimum node utilization and the page size. We have chosen the minimum node utilization to be 0.2, and page size to be 8K. The minimum node utilization does not affect the number of distance computations made for a search query too much [CP98], 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 with minimum utilization of 0.2 performed slightly better, so we only include the results for that value. The node size affects the query performance regarding both I/O costs and distance computations made [CPZ98b] (Note that, we do not consider the I/O performance here). A page size of 4K was shown to be the best choice for minimizing distance computations (which is referred as the CPU cost) [CPZ98b] for experiments with 5 dimensional vectors and using L_∞ metric, although the difference from the other choice of page sizes 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 are conducted with the same data sets of Euclidean vectors discussed in section 5. For comparison, we included 3 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 used are denoted by their *m, v, k, p* parameters as usual. Remember that mvpt(2,1,13,7) is actually a vp-tree that makes use of *p* (7) of the pre-computed distances, and that mvpt(2,10,50,7) has one internal node with 10 vantage points, and a large leaf capacity. Note that the results for the vp-tree and the mvp-trees were also given in sections 5.2 and 8.1.

In Figures 8.13-16, 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 20-dimensional randomly generated Euclidean vectors (Figure 8.13), 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 as good as the mvp-tree structures, surpassing vpt(2) performance by making 30% less distance computations. The reason we believe the M-trees catch up with mvp-trees is that as the distance distribution is very narrow, mvp and vp-tree performances degenerate faster by increasing query range 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 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 as they partition the data points based on physical closeness (trying to create clusters), however, this leads to too much overhead for small query ranges, where they become less efficient compared to mvp-trees. We see a similar trend in the 10-

dimensional case where the performance of M-trees are even better (Figure 8.14). For the 10-dimensional case, although M-trees perform worse for query ranges 0.2 and 0.3, they catch up when the query range increases and actually give the best performance for the query range 0.5 (which is a high query range in 10 dimensional space unlike the case in 20-dimensional space). Since the distances in 10-dimensional space are smaller (compared to 20-dimensional case), M-tree structure becomes more efficient as it is also able to decrease the radii of the spherical regions that it partitions the data space into. This results in comparably better performances for high query ranges.

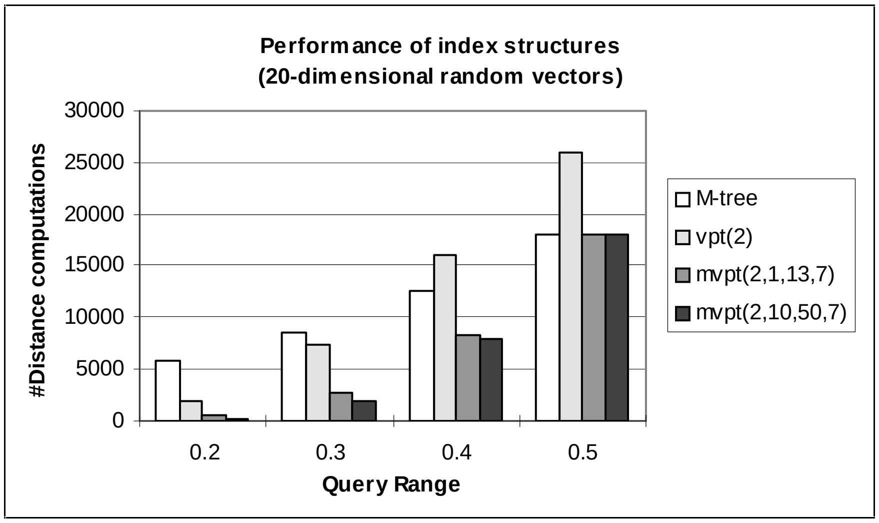


Figure 8.13. Comparison of M-tree performance with two mvp-trees and a vp-tree of order 2 for 20-dimensional randomly generated Euclidean vectors.

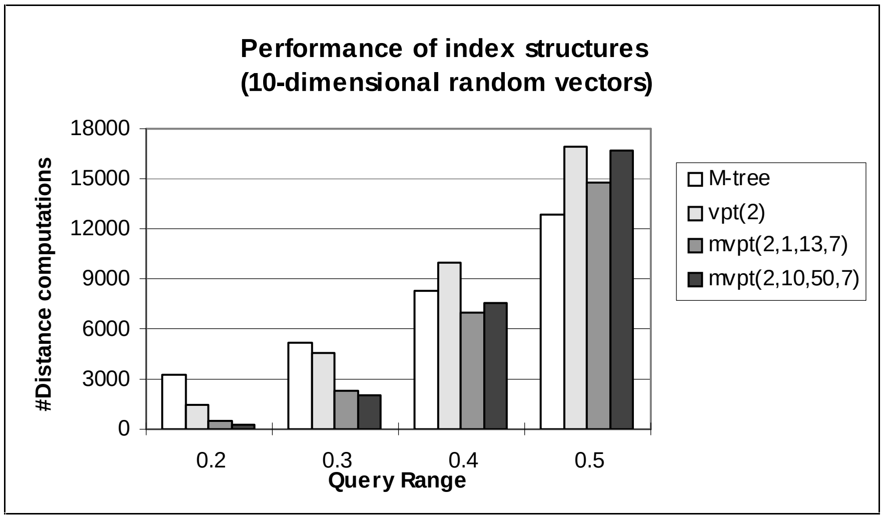


Figure 8.14. Comparison of M-tree performance with two mvp-trees and a vp-tree of order 2 for 10-dimensional randomly generated Euclidean vectors.

M-tree does not perform as good for the vectors generated in clusters. When query set Q_1 is used (Figure 8.15), for the range 1.0, the mvp-trees make less than around 50% distance computations than the M-tree. For query range 0.7, it is %72 for mvpt(2,1,13,7) and 80% for mvpt(2,10,50,7) to give an idea. For small query ranges, M-tree makes at least an order of magnitude more distance computations than the other structures, especially mvp-trees. When the query set Q_2 is used, the relative performances display a similar pattern as seen in Figure 8.16. For query set Q_2, M-tree performs close to vpt(2) for large query ranges, and actually performs less distance computations for range 1.0.

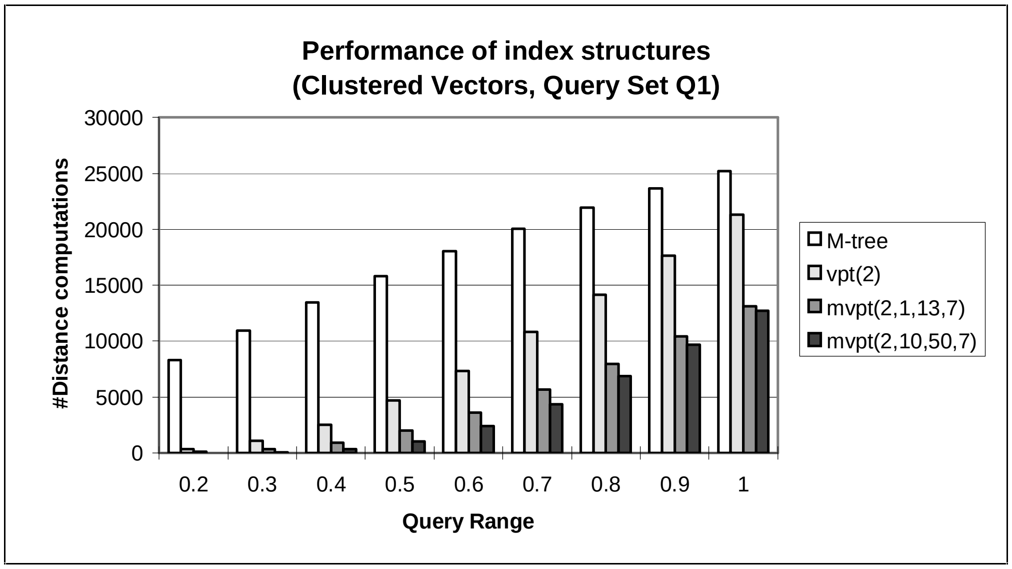


Figure 8.15. Comparison of Mtree performance with two mvp-trees and a vp-tree of order 2 for 20-dimensional Euclidean vectors generated in clusters. (Query Set Q_1).

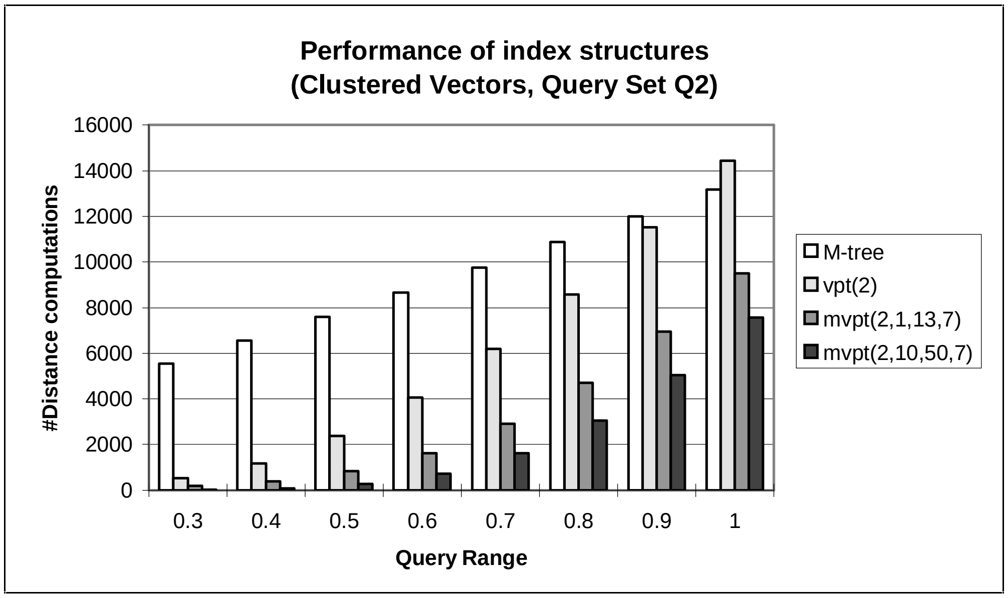


Figure 8.16. Comparison of Mtree performance with two mvp-trees and a vp-tree of order 2 for 20-dimensional Euclidean vectors generated in clusters. (Query Set Q_2).

The partitioning strategy of M-trees is different than mvp-trees as they partition the data space into a number of sphere-like regions 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 done some experiments using such 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 (L_2 metric) was used as the distance metric. We also generated 100 query points that conform to the same distribution. Table 8.1 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.

Query Range	**Total number of near neighbors found (100 queries)**	**Total number of distance computations M-tree**	**Total number of distance computations vpt(2)**	**Total number of distance computations mvpt(2,1,13,7)**	**Total number of distance computations 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

Table 8.1. Performance results for the queries with the data set where data points form several physical clusters

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 on the geometry of the application data space, and provides a filtering method for similarity search queries only based on relative distances between the data objects. Similar to vp-tree, mvp-tree takes the approach of partitioning the data space around *vantage points*, but behaves much wiser in choosing these points and makes use of the pre-computed (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 number of vantage points used 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, mvp-tree is a static index structure. It is constructed in a top down fashion on a given set of data points, and it 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, the restructuring is unavoidable for balanced mvp-trees, it can be done with minimum number of distance computations if all the pre-computed distances are kept and an mvpt-tree with a single directory node is used.

We considered the number of distance computations as the cost measure for the performance comparisons of mvp-trees with other access structures. While this is justifiable for applications where the 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 remains to be done as future research.

In section 6, we discussed some heuristics on choosing better vantage points and show 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 the data distribution is not known a priori and no geometry of the data space can be assumed. It becomes especially important for the mvp-trees that employ a large number of vantage points in the internal nodes, because 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.

References

[AFA93] R. Agrawal, C. Faloutsos, A. Swami. “Efficient Similarity Search In Sequence Databases”. *In FODO Conference,* 1993.

[BK73] W.A. Burkhard, R.M. Keller, “Some Approaches to Best-Match File Searching”, *Communications of the ACM,* 16(4), pages 230-236, April 1973.

[BKSS90] N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger, “The R*-tree: An Efficient and Robust Access Method for Points and Rectangles”, *Proceedings of the 1990 ACM SIGMOD Conference, Atlantic City,* pages 322-331, May 1990.

[BBKK97] S. Berchtold, C. Bohm, D.A. Keim, H-P. Kriegel, “A Cost Model for Nearest Neighbor Search in High-Dimensional Data Space”, *Proceedings of the 16th Symposium on Principles of Database Systems,* pages 68-77, 1997.

[BKP96] S. Berchtold, D.A. Keim, H-P. Kriegel, “The X-tree An Index Structure for High Dimensional Data”, *Proceedings of the 22nd VLDB Conference*, pages 28-39, 1996.

[BO97] T. Bozkaya, M. Ozsoyoglu, “Distance-Based Indexing for High-Dimensional Metric Spaces”, *Proceedings of the 1997 ACM SIGMOD Conference, Tucson,* pages 357-368, 1997,

[Bri95] S. Brin, “Near Neighbor Search in Large Metric Spaces”, *Proceedings of the 21st VLDB Conference,* pages 574-584, 1995.

[Chi94] T. Chiueh, “Content-Based Image Indexing”, *Proceedings of the 20th VLDB Conference, pages* 582-593, 1994.

[CP98] P. Ciaccia, M. Patella, “Bulk Loading the M-tree”, *Proceedings of the 9th Australasian Database Conference (ADC), Perth, Australia,* February 1998.

[CPZ97] P. Ciaccia, M. Patella, P. Zezula, “M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces”, *Proceedings of the 23rd VLDB Conference*, 1997.

[CPZ98a] P. Ciaccia, M. Patella, P. Zezula, “Processing Complex Similarity Queries with Distance-Based Access Methods”, *Proceedings of the 6th EDBT International Conference,* March 1998.

[CPZ98b] P. Ciaccia, M. Patella, P. Zezula, “A Cost Nodel for Similarity Queries in Metric Spaces”, *to appear in Proceedings of the 17th Symposium on Principles of Database Systems,* May 1998.

[FEF+94] C. Faloutsos, W. Equitz, M. Flickner et al., “Efficient and Effective Querying by Image Content”, *journal of Intelligent Information Systems (3),* pages .231-262, 1994.

[FRM94] C. Faloutsos, M. Ranganathan, Y. Manolopoulos. “Fast Subsequence Matching in Time-Series Databases”. *Proceedings of the 1994 ACM SIGMOD Conference*, *Minneapolis,* pages 419-429, May 1994.

[Gut84] A. Guttman, “R-Trees: A Dynamic Index Structure for Spatial Searching”, *Proceedings of the 1984 ACM SIGMOD Conference, Boston,* pages 47-57, June 1984.

[LJF94] K-I. Lin, H.V. Jagadish, C. Faloutsos, “The TV-tree-An Index Structure for High Dimensional Data”, *VLDB Journal, volume* 3, pages 517-542, October 1994.

[Ott92] M. Otterman, “Approximate Matching with High Dimensionality R-trees”. *M.Sc Scholarly paper, Dept. of Computer Science, Univ. of Maryland, Collage Park, MD, 1992. Supervised by C. Faloutsos.*

[RKV95] N. Roussopoulos, S. Kelley, F. Vincent, “Nearest Neighbor Queries”, *Proceedings of the 1995 ACM SIGMOD Conference, San Jose,* pages 71-79, May 1995.

[Sam89] H. Samet, “The Design and Analysis of Spatial Data Structures”, *Addison Wesley*, 1989.

[SRF87] T. Sellis, N. Roussopoulos, C. Faloutsos, “The R+-tree: A Dynamic Index for Multi-dimensional Objects”, *Proceedings of the 13th VLDB Conference,* pages 507-518, September 1987.

[SW90] D. Shasha, T. Wang, “New Techniques for Best-Match Retrieval”, *ACM Transactions on Information Systems,* 8(2), *pages* 140-158, 1990.

[Uhl91] J. K. Uhlmann, “Satisfying General Proximity/Similarity Queries with Metric Trees”, *Information Processing Letters, vol* 40, *pages* 175-179, 1991.

[Yia93] P.N. Yiannilos, “Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces”, *ACM-SIAM Symposium on Discrete Algorithms,* pages 311-321, 1993.

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 be the vantage point of a node that we visit during the search, and M be 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.