DISTANCE-BASED INDEXING FOR HIGH-DIMENSIONAL METRIC SPACES[*]

Tolga Bozkaya
Department of Computer Engineering & Science
Case Western Reserve University
email: bozkaya@alpha.ces.cwru.edu

Meral Ozsoyoglu
Department of Computer Engineering & Science
Case Western Reserve University
ozsoy@alpha.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 data domain is high dimensional, or the distance function used to compute distances between data objects is non-Euclidean. In this paper, we introduce a distance based index structure called multi-vantage point (mvp) tree for similarity queries on high-dimensional metric spaces. The mvp-tree uses more than one vantage point to partition the space into spherical cuts at each level. It also utilizes the pre-computed (at construction time) distances between the data points and the vantage points. We have done 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 20% to 80% for varying query ranges and different distance distributions.

1. Introduction

In many database applications, it is desirable to be able 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, 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, we would like to find similar patterns among a given collection of sequences. Image databases can be queried to find and retrieve images in the database that are similar to the query image with respect to 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.

In all the applications above, the problem is to find similar data items 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 data domain is high dimensional, and the distance function employed is *metric*. It is important for an application to have a metric distance function to make it possible to do filtering of distant data items for a similarity query by using the *triangle inequality* property (section 2). Because of the high dimensionality, the distance calculations between data items are assumed to be very expensive. Therefore, an efficient access mechanism should certainly have to 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 that are used 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 as 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 the further identification and semantic interpretation of images to users.

In this paper, 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

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

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 at positions to create sublists of equal cardinality. The order of the tree corresponds to the number of partitions to be 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 points 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. The efficient filtering at the leaf level is utilized more by making the leaf nodes to have higher node capacities. By this way, the major filtering step during search is delayed to the leaf level.

We have done experiments with 20-dimensional Euclidean vectors and gray-level images to compare vp-trees and mvp-trees to demonstrate mvp-trees' efficiency. The distance distribution of data points plays an important role in the efficiency of the index structures, so we experimented on two sets of Euclidean vectors with different distance distributions. In both cases, mvp-trees made 20% to 80% less number of distance computations compared to vp-trees for small query ranges. For higher query ranges, the percentagewise difference decreased gradually, yet the mvp-trees performed 10% to a respectable 30% less distance computations for the largest query ranges we used in our experiments.

Our experiments on gray-level images using L_1 and L_2 metrics (see section 5.1) also revealed 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 upto 20-30% less distance computations.

The rest of the paper is organized as follows. Section 2 gives the definitions for high dimensional metric spaces and similarity queries. Section 3 presents the problem of indexing in high dimensional spaces and also presents previous approaches to this problem. The related work for distance-based index structures to answer similarity based queries 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. We summarize our results and point out future research directions in section 6.

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 should be assuming when designing an index structure based on distances between objects in a metric space. Note that, we cannot make use of any geometric information about the metric space, unlike the way we can for a Euclidean space. 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:

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 $\{ X_i \mid X_i \in X$ 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 near neighbor query we have given above.

Here, we are mainly concerned on distance based indexing for high-dimensional 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 we assume that distance computations in high-dimensional metric spaces are very expensive. In the next section, we discuss the indexing problem for high-dimensional metric spaces, and review previous approaches to the problem.

3. Indexing in High-Dimensional 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.

The problem of indexing high-dimensional spaces can be approached in different ways. One approach is to use distance preserving transformations to Euclidean spaces, which we discuss in section 3.1. Another approach is using distance-based index structures. In section 3.2, we discuss distance-based index structures, and briefly review the previous work. In section 3.3, we discuss the vp-tree structure in detail since it is the most relevant approach to work.

3.1 Distance Preserving Transformations

There are ways to use conventional spatial structures for high-dimensional domains. One way is to apply a mapping of objects from a high-dimensional space to a low-dimensional (Euclidean) space by using a distance preserving transformation, and then using conventional index structures (such as R-trees) as a major filtering mechanism. A distance preserving transformation is a mapping from a high-dimensional domain to a lower-dimensional domain where the distances between objects before the transformation (in the actual space) are greater than or equal to the distances after the transformation (in the transformed space). That is, the distance preserving functions underestimate the actual distances between objects in the transformed space. Distance preserving transformations have been successfully used to index high-dimensional data in many applications such as time sequences [AFA93, FRM94], and images [FEF+94].

The distance preserving functions such as DFT, Karhunen-Loeve are applicable to any Euclidean domain. Yet, it is also possible to come up with application specific distance preserving transformations for the same purpose. In the QBIC (Query By Image Content) system [FEF+94], color content of images can be used to answer similarity queries. The difference of 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 (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, the 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 are proven to be less than or equal to the distance between their color histograms, that is, the transformation is distance preserving. Similarity queries on color content of images are answered by first using the index on average color vectors as the major filtering step, and then refining the result by actual computations of histogram distances.

Note that, although the idea of using distance preserving transformation works fine for many applications, it makes the assumption that such a transformation exists and applicable to the application domain. Transformations such as DFT or Karhunen-Loeve are not effective in indexing high-dimensional vectors where the values at each dimension are uncorrelated for any given vector. Therefore, unfortunately, it is not always possible or cost effective to employ this method. Yet, there are distance based indexing techniques that are applicable to all domains where metric distance functions are employed. These techniques can be directly employed for high-dimensional spatial domains as the conventional distance functions (such as Euclidean, or any L_p distance) used for 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 in the case of 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 below.

3.2 Distance-Based Index Structures

There are a number of research results on efficiently answering similarity search queries in different contexts. In [BK73], Burkhard & Keller 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.

In the second approach in [BK73], they partition the space into a number of sets of keys. For each set, they arbitrarily pick a *center* key, and calculate the *radius* which is the maximum distance between the *center* and any other key in the set. The keys in a set are partitioned into other sets 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 approach of [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 should be 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, such as the one in [SW90], pre-computed distances between the data elements are used 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 distance computations. In [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 other pre-computed distances. The technique of storing and using pre-computed distances may be effective for data domains with small cardinality, however, the space requirements and the search complexity becomes overwhelming for larger domains.

In [Uhl91], Uhlmann introduced 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 to 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], the vp-tree structure was enhanced by an algorithm to pick vantage-points for better decompositions. In [Chi94] the vp-tree structure is modified to answer nearest neighbor queries. We talk about the vp-trees in detail in section 3.3.

The gh-tree (*generalized hyperplane tree)* structure was also introduced in [Uhl91]. It is constructed as follows. At the top node, two points are picked and the remaining points are divided into two groups depending on which of these two points they are closer to. The two branches for the two groups are built recursively in the same way. 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.

More recently, Brin introduced the GNAT (*Geometric Near-Neighbor Access Tree*) structure [Bri95]. 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* datasets (one for each *split point*), depending on which *split point* they are closest to. For each *split point,* the minimum and maximum distances from the points in the datasets of other *split points* are recorded. The tree is recursively built for each dataset 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 number of distance computations in the experiments for different data sets.

3.3 Vantage point tree structure

Let us briefly discuss the vp-trees 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 right and left 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 the 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 $\{ d(S_i, S_v) \mid \forall S_i \in S\}$
   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.

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 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. Here, 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 compared to binary vp-trees at the construction stage.

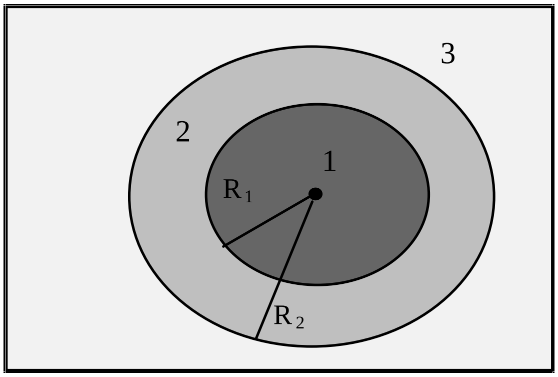

Figure 1. The root level partitioning of a vp-tree with branching factor 3. The three different regions are labelled 1, 2, 3, and they are all shaded differently.

However, there is one problem with high-order vp-trees when the order is large. The vp-tree partitions the data space into spherical cuts (see Figure 1). 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 in doing similarity searches. 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 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, we can consider the N-dimensional volumes of regions 1 and 2 to be 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. As 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 point at one level and keeps extra information for the data points in the leaf nodes for effective filtering of non qualifying points in a similarity search operation.

4.1 Motivation

Before we introduce the mvp-tree, we first discuss a few useful observations that can be used as heuristics for a better search structure. The idea is to partition the data space around a vantage point at each level for a hierarchical search.

Observation 1*:* It is possible to partition a spherical shell-like region using a vantage point chosen from outside the region. This is shown in Figure 2, where a vantage point outside of the 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). The vantage point does not have to be from inside the region, unlike the strategy followed in vp-trees.

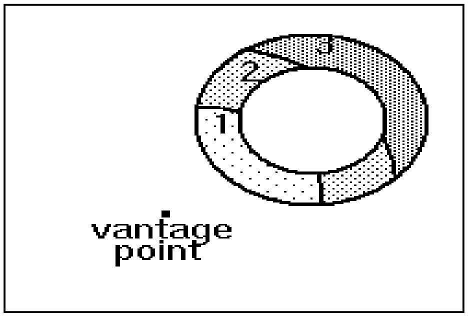

Figure 2. Partitioning a spherical shell-like region using a vantage point from outside.

This means that we can use the same vantage point 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 we can use the same vantage point for all the children of a node, we can as well keep that vantage point in the parent. This way, we would be keeping more than one vantage point in the parent node. We can avoid creating the children nodes by incorporating them in the parent. This could be done by increasing the fanout of the parent node. The mvp-tree takes this approach, and uses more than one vantage points in the nodes for higher utilization.

Observation 2*:* In the construction of the vp-tree structure, for each data point in the leaves, we compute the distances between that point and all the vantage points on the path from the root node to the leaf node that keeps that data point. So for each data point, ($\log_m$ n) distance computations (for 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.

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 less number of vantage points in the non-leaf levels.

In this section, we will 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 at the leaves to be kept (*p*).

In binary mvp-trees, the first vantage point (we will refer to it by S_{v1}) divides the space into two parts, and the second vantage point (we will refer to it by 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 further partitions with respect to the second vantage point.

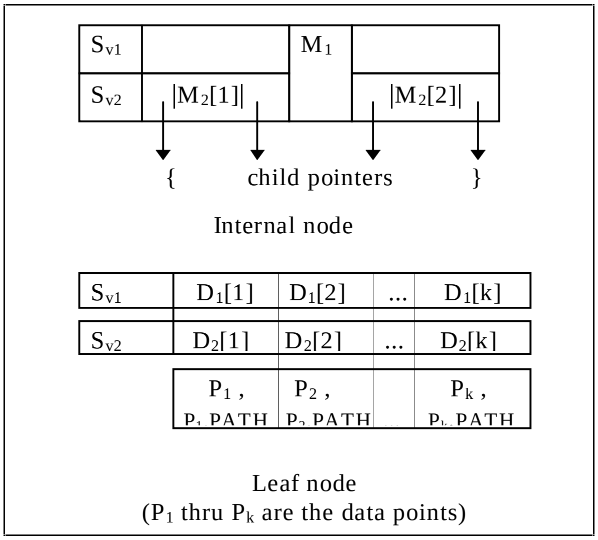

Figure 3. Node structure for a binary mvp-tree.

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

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 3 below shows the structure of internal and leaf nodes of a binary mvp-tree.

Having given the explanation for the parameters and the structure, we present the construction algorithm next. Note that, 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 we have explained above. The variable *level* is used to keep track of the number of vantage points used along the path from the current node to the root. It is initialized to 1.)

```
1) If $|S| = 0$, then create an empty tree and quit.

2) If $|S| \leq k+2$, then
  2.1) Select an arbitrary object from S. ($S_{v1}$ is the first vantage point)
  2.2) Let S := S - { $S_{v1}$ } (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) Let S := S - { $S_{v2}$ } (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)Let S := S - { $S_{v1}$ } (Delete $S_{v1}$ from S)
   3.3) Calculate all $d(S_i, S_{v1})$ where $S_i \in S$
          if ($level \leq$ p) $S_i$.PATH[$l$] = $d(S_i, S_{v1})$.
   3.4) Order the objects in S with respect to their distances from $S_{v1}$.
   $M_1$= median of { $d(S_i, S_{v1})$ | $\forall S_i \in S$} Break this list into 2 lists of equal cardinality at the median. Let $SS_1$ and $SS_2$ these two sets in order, i.e., $SS_2$ 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 $SS_m$)
   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 { $d(S_j, S_{v2})$ | $\forall S_j \in SS_1$}
          $M_2[2]$= median of { $d(S_j, S_{v2})$ | $\forall S_j \in SS_2$}
   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. Also, higher fanouts at the internal nodes are also possible, and may be more favorable in most cases.

Observe that, we chose the second vantage point to be one of the farthest points from the first vantage point. If the two vantage points were close to each other, they would not be able to effectively partition the dataset. 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 chose the best vantage point) for vp-trees can also be applied to the mvp-trees.

The construction step requires O(n $\log_m$ n) distance computations for the mvp-tree. There is an extra storage requirement for the mvp-trees as we keep *p* distances for each data point in the leaf nodes, however it does not change the order of storage complexity.

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 we keep two vantage points 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 the leaf nodes.

It is a good idea to keep *k* large so that most of the data items are kept in the leaves. If k is kept 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 non-qualifying (out of the search region) points from further consideration by making use of the *p* pre-computed distances for each leaf point. 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 means of avoiding unnecessary distance computations.

4.3 Search algorithm for mvp-trees

We present the search algorithm below. Note that the search algorithm proceeds depth-first for mvp-trees. We need to 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 eliminating data points in the leaves from further consideration (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 search algorithm as follows:

```
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
          ( $PATH[i] - r \leq S_i.PATH[i] \leq PATH[i] + r$ )
    holds,
          then compute $d(Q, S_i)$. If $d(Q, S_i) \leq r$, then $S_i$
    is in the answer set.

3) Else if the current node is an internal node
        3.1) if ($l \leq p$) $PATH[l] = d(Q, S_{v1})$,
             if ($l<p$) $PATH[l+1] = d(Q, S_{v2})$.
        3.2) if $d(Q, S_{v1}) + r \leq M_1$, then
               if $d(Q, S_{v2}) + r \leq M_2[1]$ then recursively
               search the first branch with $l=l+2$
               if $d(Q, S_{v2}) - r \geq M_2[1]$ then recursively
               search the second branch with $l=l+2$
        3.3) if $d(Q, S_{v1}) - r \geq M_1$, then
               if $d(Q, S_{v2}) + r \leq M_2[2]$ then recursively
               search the third branch with $l=l+2$
               if $d(Q, S_{v2}) - r \geq M_2[2]$ then recursively
               search the fourth branch with $l=l+2$
```

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) (N is the cardinality of the dataset) distance computations. 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 some experiments to study the performance of mvp-trees.

5. Implementation

We have implemented the main memory model of the mvp-trees with different parameters 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 very costly for high-dimensional metric spaces, we use the number of distance computations as the cost measure. We counted the number of distance computations required for similarity search queries by both mvp and vp-trees for the same set of queries 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) are used for empirical study.

A. High-Dimensional Euclidean Vectors:

We used two sets of 50.000 20-dimensional vectors as data sets. Euclidean distance metric is used as the distance metric in both cases. For the first set, all vectors are chosen randomly from a 20-dimensional hypercube with each side of size 1. Each of these vectors is simply generated by randomly choosing 20 real numbers from the interval [0,1]. The pairwise distance distribution of these randomly chosen vectors are shown as a histogram in Figure 4. The distance values are sampled at intervals of length 0.01.

Note that this data set is highly synthetic. As the vectors are uniformly distributed, they are mostly far away from each other. 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) ineffective in queries having values of *r* (similarity measure) larger than 0.5, although higher *r* values are quite reasonable for legitimate similarity queries.

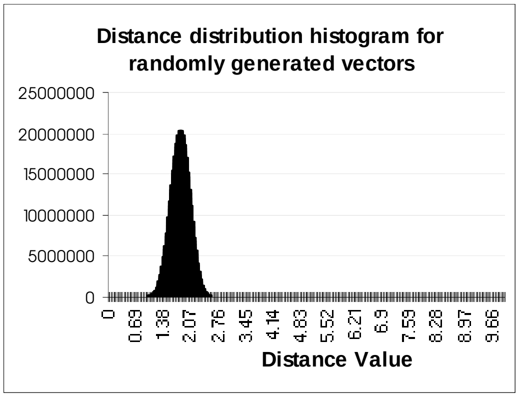

Figure 4. Distance distribution for randomly generated Euclidean vectors.

(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)

The second set of Euclidean vectors 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 [-ε, ε], where ε is a small constant (such as between 0.1 to 0.2).

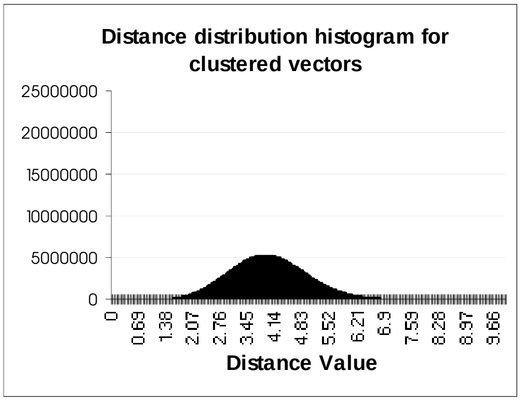

Figure 5. Distance distribution for Euclidean vectors generated in clusters.

(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)

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 hypercube of side 1. 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 very close in the Euclidean space. In Figure 5, we see the distance distribution histogram for a set of clustered data 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.

B. Gray-Level MRI Images:

We also experimented on 1151 MRI images with 256*256 pixels and 256 values of graylevel. 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:

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

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

When calculating distances, we simply treat these images as 256*256=65536-dimensional Euclidean vectors, and accumulate the pixel by pixel intensity differences 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 kept on secondary storage using around 61K per image (images are in binary PGM format using one byte per pixel).

We see the distance distributions of the MRI images for L_1 and L_2 metrics in the two histograms shown below in Figures 6 & 7. There are (1150*1151)/2 = 658795 different pairs of images and hence, that many computations. The L_1 distance values are normalized by 10000 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.

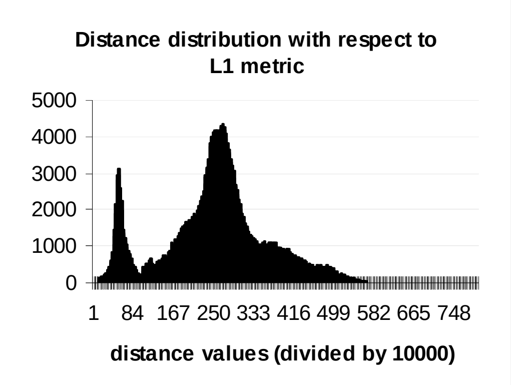

Figure 6. Distance histogram for images when L1 metric is used.

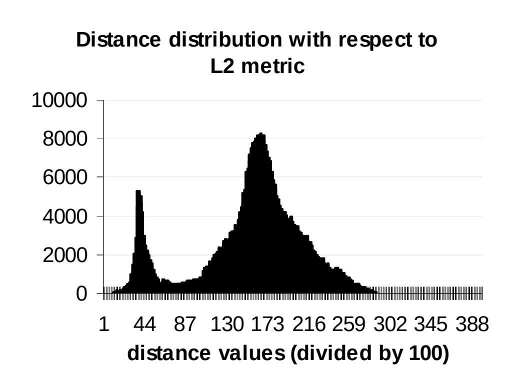

Figure 7. Distance histogram for images when L2 metric is used.

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 these sense that we can see what distance ranges can be considered similar. If L_1 metric is used, a tolerance factor (*r*) around 500000 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.

For gray level images, color histograms can be used to compute similarity. Unlike color images, there is no cross talk (between the colors) in graylevel (or any mono-color) images, and therefore, an L_p metric can be used to compute distances between color histograms. The histograms will simply be treated as if they are 256-dimensional vectors, and then, an L_p metric can be used.

5.2 Experimental Results

A. High-Dimensional Euclidean Vectors:

In Figures 8 and 9, we present the search performances of four tree structures for two different data sets of Euclidean vectors. 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. We have experimented with vp-trees of higher order, however higher order vp-trees gave similar or worse performances, therefore, we do not present the results for them. We have also tried several mvp-trees with different parameters, however, we have observed that order 3 (*m*) gives the most reasonable 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. All the results 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 with randomly selected query objects from the 20-dimensional hypercube with each side of size 1. In Figures 8 and 9, 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.

As shown in Figure 8, 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). The gap closes 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 (0.15 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 non-qualifying points for the search operations.

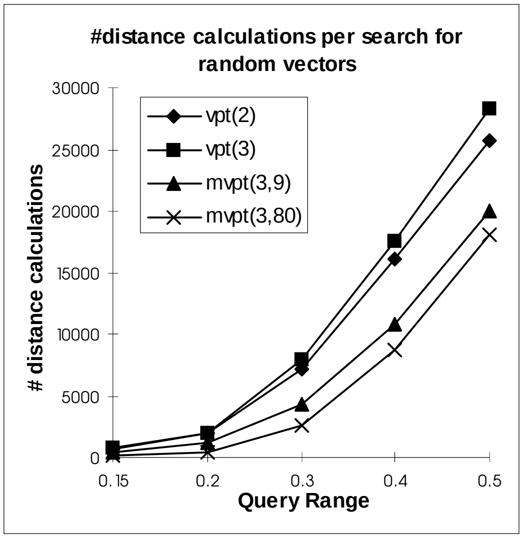

Figure 8. Search performances of vp and mvp trees for randomly generated Euclidean vectors.

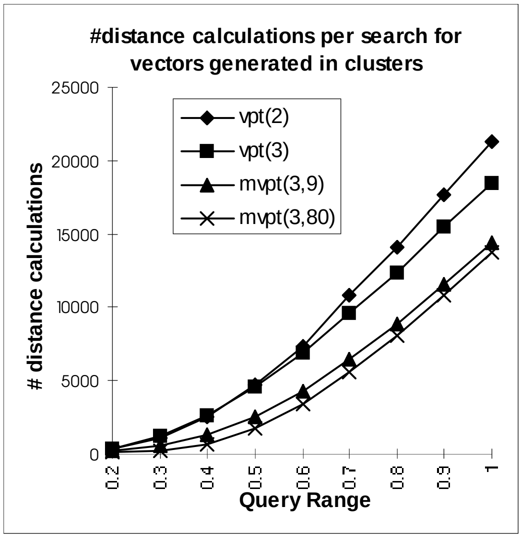

Figure 9. Search performances of vp and mvp trees for Euclidean vectors generated in clusters.

Figure 9 shows the performance results for the data set where the vectors are generated in clusters. For this data set, 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 the 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. We have also run experiments on the same type of data with different cluster sizes, however the percentages did not differ much.

We can summarize our observations as follows:

- Higher order vp-trees perform better for wider distance distributions, however the difference is not much. For datasets 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 delay 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 (30% for the largest ranges we have tried).

B. Gray-Level MRI Images:

We display the experimental results for the similarity search performances of vp and mvp trees on MRI images in Figures 10 and 11. 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 for each of them, 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, where each query object is an MRI image selected randomly from the data set.

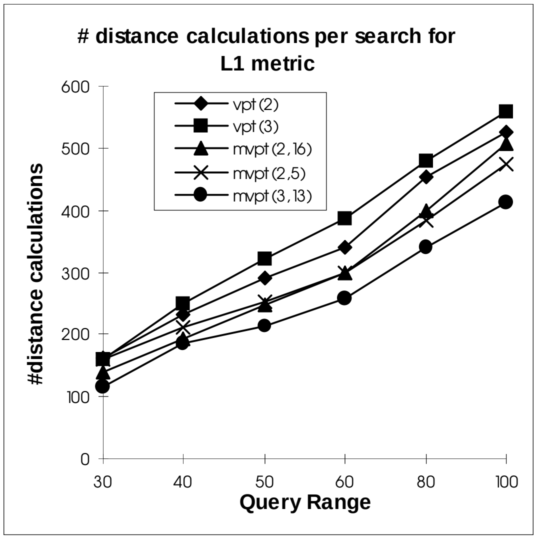

Figure 10. Similarity search performances of vp and mvp trees on MRI images when L_1 metric is used for distance computations.

Figure 10 shows the search performance of these 5 structures when L_1 metric is used. 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).

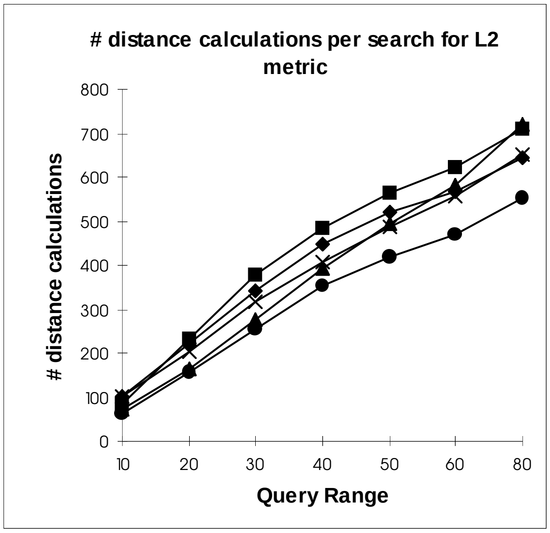

Figure 11. Similarity search performances of vp and mvp trees on MRI images when L_2 metric is used for distance computations.

Figure 11 shows the search performances when L_2 metric is used. Similar to the case when L_1 metric was used, vpt(2) outperforms vpt(3) wih 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. 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 dataset of gray-level images support our previous observations about the efficiency of mvp-trees with high leaf-node capacity. Even though our image dataset 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 alarger set of images, we would expect higher performance gains.

6. Conclusions

In this paper, we 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 assumption on the geometry of the application space, and provides a filtering method for similarity search queries only based on relative distances between the data objects. Similar to an existing structure, the vp-tree, mvp-tree takes the approach of partitioning the data space around *vantage-points*, but behaves much clever in choosing these points and makes use of the pre-computed distances (at the construction stage) when answering similarity search queries.

Mvp-trees, like other distance based index structures, is a static index structure. It is constructed in a top down fashion on a static set of data points, and guarantees the fact that it is a balanced structure. Handling update operations (insertion and deletion) without major restructuring, and without violating the balanced structure of the tree is 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. We plan to look further into this problem of extending mvp-trees with insertion and deletion operations that would not imbalance the structure.

It would be also interesting to determine the best vantage point for a given set of data objects. Methods to determine better vantage points with a little extra cost would pay off in search queries by causing less number of distance computations to be done. We also plan to look further into this problem.

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.

[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.

[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 Strcuture for Spatial Searching", *Proceedings of the 1984 ACM SIGMOD Conference, Boston,* pages 47-57, June 1984.

[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 the13th 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

Let us 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.