FASTMAP AS A FILTER FOR SPATIAL ACCESS METHODS

by

DONALD J. SANTOS

Submitted in partial fulfillment of the requirements

For the degree of Master of Science

Thesis Adviser: Dr. Z. Meral Ozsoyoglu

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

May, 2005

Contents

Figures

Definitions and Symbols

Metric Space: A space containing a set of objects where the distances between the objects retain the properties of symmetry, non-negativity and adherence to the triangle inequality.

Distance Function: A metric distance function calculates the dissimilarity between two objects in a metric space. Examples of this are any of the LP-Norms.

Similarity Search: Given a dataset of objects and one query object, search the dataset for objects which are similar within a given radius to the query object. These are also called range queries, proximity queries, or near-neighbor queries.

Exhaustive Search: A similarity search in which the distance between the query object and every single object in the dataset is computed, also called a brute force search.

Image: An object which has been mapped into a point in vector space of arbitrary dimensionality **(K)**.

Original Space: The metric space in which the pure dataset resides with no dimensionality reduction involved.

Target Space: The metric space in which the dataset images reside. Since the images are K-dimensional points in vector space, this is also called k-d space.

D()	-	The metric distance function being used for a similarity search.
N	-	The total number of objects in the dataset.
R	-	The search radius; the maximum dissimilarity of a query match.
d	-	The dimensionality of the original space.
K	-	The dimensionality of the target space.

True Distance Computation: D(Oi, Oj) where Oi and Oj are objects in original space.

KDC (k-d space distance computation): D(Xi, Xj) where Xi and Xj are object images.

FastMap as a Filter for Spatial Access Methods

Abstract

by

DONALD J. SANTOS

It is very useful to be able to search a large database for objects based on their similarity to a given query object. There is a wide array of indexing structures available which have been studied well in an attempt at increasing query efficiency, but most techniques run into significant problems when the objects being indexed are very complex or high in dimensionality. One of the most promising ideas is to map the objects as points in an arbitrary k-d vector space, and then build the indexing structure using these points. These query results can be used as a "filter" to attain the true similarity query set in a space of much lower dimensionality, significantly lowering the computational cost. This thesis provides analysis and experimental data to explore the usefulness of FastMap, one such dimensionality reduction technique, as it is applied to point and spatial access methods as a filter.

1. Introduction

Given a large database of objects, it is very useful in many applications to be able to search for objects that are similar to a given query object. Some of these applications include, but are not limited to, image recognition in multimedia databases, audio and video similarity searches, DNA or protein sequence similarity searches in genetic databases, and text document searching. Taking into account the diversity of these applications and the wide breadth of their usefulness, there have been many studies done to increase the efficiency and practicality of methods for indexing such databases.

The structures employed to index large databases are often called point access methods (PAMs) in the case of point objects, and spatial access methods (SAMs) when objects represented by regions in a metric space with area or volume can be supported. These types of indexing methods are typically categorized as distance-based indexing methods or multidimensional indexing methods. The primary distinction between these two categories of access methods is that distance-based indexing structures are built based on a distance function, which measures the dissimilarity between two objects, whereas multidimensional indexing methods have no defined distance function when they are built, and hence can be used for queries using any arbitrary distance function. One drawback of multidimensional indexing structures is that they assume the objects being indexed can be represented by feature vectors containing scalar values, which requires either a workable dataset or a domain expert to extract these features. The number of features for an object usually translates to a notion of dimensionality, and it follows that objects with no clear notion of dimensionality are very difficult to work with in a context which requires feature extraction. Distance-based indexing methods overcome this problem by relying solely on the distance function for partitioning and querying the metric space in which the objects reside. Of course if one desired to perform a similarity query on a distance-based indexing structure using a distance function that differed from the one with which it was built, then the entire structure would need to be reconstructed with the new distance function.

While there are excellent multidimensional and distance-based indexing structures available to utilize in different contexts, they all suffer from one central drawback. When the metric space in which the objects reside becomes very high in dimensionality, the cost of construction and the degeneration of query efficiency grow exponentially. This has been referred to as the "Curse of Dimensionality" [Bel57]. In the case of distance-based indexing, this translates to the effect of very costly distance computations being performed on an order worse than that of an exhaustive, brute force search [HAK00]. For multidimensional indexing methods, many dataset types are not even usable. For example, in a database of character sequences, the string objects lack a notion of dimensionality and spatial coordinates.

One promising way of dealing with all of these problems is an area of study called dimensionality reduction, or mapping techniques. The objective of these techniques is to take a dataset of objects and, given a distance function **D()**, map them as points in an arbitrary k-d vector space such that their original distances are preserved as well as possible. This process has a few implications pertaining to the problems facing point and spatial access methods. First, it effectively derives feature vectors from objects of any dimensionality, or no notion of dimensionality at all, given no other information except the distances between objects. By doing this, multidimensional access structures (e.g. the

R-Tree) [Gutt84] can be employed on datasets that were originally not suitable. In addition to this, the ability to map to an arbitrary dimensionality provides the potential for decreasing the complexity of the objects being indexed, and hence drastically reduces the computational cost of performing proximity comparisons between objects. For distance-based indexing methods, this has great potential for supplemental improvement in that objects of potentially huge dimensionality can be mapped as points in a more manageable metric space, and the previously costly distance comparisons that constituted the majority of the computational burden are now greatly assuaged. Finally, it has also been suggested that mapping complex or high dimensional datasets to two or three dimensions can reveal information about the intrinsic geometry of the data, and clustering of the data [Fal95]. This can be useful in data mining or any application where data visualization is difficult.

With our mapped dataset of points in k-d space, we can now utilize both multidimensional as well as distance-based index structures. Since both PAMs and SAMs can handle datasets consisting of points in vector space, we may now make use of virtually any indexing method we want in within the realm of metric space partitioning. The key to mapping techniques being useful, however, is that the mapping process is contractive; that is, the distance between any two points in the new k-d space will be less than or equal to the distance between the two original objects. This property guarantees that a range query on any of the points in k-d space will return a result set that is a superset of the same range query on the original object in the original metric space. Retrieving such a query set using the mapped points is called a “filtering” process, because it effectively filters out a portion of the dataset that are known to be true dismissals. Since this set is indeed a superset resulting from the approximation of the inter-object distances, it will contain the entire true query set and also some false matches. Thus, a “refinement” process is necessary to validate which points in the set correspond to an original object that is in the true range query result set. This can be done by simply using the distance function on the query object and all objects referenced by the query set retrieved in k-d space.

This filtering and refinement process is the topic of this thesis. We take one of the most promising dimensionality reduction techniques called FastMap [Fal95], and apply it as a filter to one distance-based PAM, the Vantage Point Tree [Yian93], and one multidimensional indexing SAM, the R*-Tree [BKSS90]. FastMap has been claimed as an effective method for mapping objects into points in an arbitrary dimension, and a useful tool for visualizing datasets in large metric spaces. In this thesis, we examine these claims in attempt to validate them, and examine ways to marginally improve the performance of spatial access methods by using FastMap as a filter. We apply the filtering and refinement process to various real and synthetic datasets, and expound upon the results. Our intention and ultimate goal is to gather data to illuminate the questions of how useful mapping techniques can be for improving the efficiency of range queries in large metric spaces. We find with our experiments that FastMap can indeed improve range query efficiency in many cases, depending on the indexing method being used, the dataset being used, and the types of queries being performed. In almost all of our experiments, FastMap cut query search time by 50-75 percent. We conclude that given a dataset of fairly complex or highly dimensional objects (10+ dimensions) and mapping to a relatively low dimensionality, FastMap is very useful indeed. Since the worst case of

FastMap is an exhaustive search of **O(N)** and most indexing methods degenerate beyond **O(N)** when dimensionality grows high, FastMap grows more useful as the problem of dimensionality increases.

In the remainder of this thesis, we will survey some related techniques, explore VP-Trees and R*-Trees in greater detail, and then present our experiments and results for applying FastMap as a filter to these access methods.

2. Survey and Related Work

In this section we will briefly overview some of the existing methods for indexing large metric spaces. These are categorized in terms of several defining traits. The two primary categories are that of multidimensional indexing and distance-based indexing techniques. Another distinction that can be made is between point access methods and spatial access methods, though this is not as critical since the terminology “spatial access method” can refer to both structures as it can support point objects easily; this characterization is more important for identifying indexing methods that cannot handle spatial objects (as in the case of point access methods). A third and also very important trait of an index structure is whether it is static or dynamic. Dynamic index structures can support insertions and deletions without reconstruction or significant re-structuring, while static structures must be re-built when any new insertions or deletions occur. While there are certain environments where a static structure is perfectly acceptable, the property of being dynamic is extremely useful in a general context.

Based on a categorization by these differing traits, we will present a brief summary of some of the more useful indexing techniques. The focus of our survey will be to acquaint the reader with some modern techniques in multidimensional indexing, distance-based indexing and dimensionality reduction that are being used with a fair amount of popularity. We will not mention every single method available, nor survey the majority of the earlier or highly specialized work, as there are excellent papers available that do exactly that. The surveys by Gaede and Gunther [GG98], Chavez et al. [CNBM01] and Hjaltason and Samet [HS03] provide a wide breadth of information and very fine detail on both old and new techniques in these areas.

2.1 Multidimensional Indexing

Here we will briefly mention some of the more common techniques for multidimensional indexing. While some of these structures are point access methods and others can support spatial data, all are alike in that they are built without knowing which distance function will be used during similarity queries, and hence can support arbitrary distance metrics at query time.

2.1.1 R-Trees

R-Trees [Gut84] and their variants, particularly the R+-Tree [SRF87] and the R*-Tree [BKSS90], are perhaps the most popular spatial access method for multidimensional indexing. In essence the R-Tree is a dynamic SAM based on sub-space partitioning via minimum bounding rectangles (MBRs). The search tree is built using these MBRs, and queries and pruning are done by calculating intersection or containment queries between a query region and the MBRs at the current level of the tree. Data resides in the leaf nodes and is often of a spatial nature, that is, polygons representing areas or clusters of points. The R-Tree is versatile and performs well in a wide variety of applications for indexing point and spatial data. In section 5, we cover the R-Tree in greater detail.

2.1.2 K-d Trees

The k-d tree [Bent75] is a point access method and a binary search tree. At each level of the tree, the current space is partitioned into two sections by (**d-1**)-dimensional hyperplanes, which are iso-oriented. These partitions intersect one of the data points, and partition the space such that some of the remaining data points will lie on both sides (at least one on each side, because points are needed to represent the leaf nodes) [GG98]. A binary search is then performed based on the partitions in order to retrieve a point at a leaf node. A newer version of the k-d tree was developed and named the adaptive k-d tree [Bent79], as it was recognized that k-d trees had the propensity to become quite unbalanced as a result of the partitioning routine not always ensuring a somewhat equal distribution of points in each subdivision. The adaptive k-d tree attempts to balance the tree by evening out these point distributions during the division of space.

2.1.3 BSP-Trees

The BSP-Tree [FKN80; FGA83] is extended from the work of k-d trees, and follows similar space subdivision with (**d-1**)-dimensional hyperplanes; however, they may have arbitrary alignment based on the distribution of objects in that particular subspace. The theory in this is that an arbitrary hyperplane based upon the current distribution of objects in the subspace (independent of the history of partitioning in the higher levels of the tree) will make for a better split, and improve the search efficiency of the tree. When the tree is completely built, every partitioning hyperplane will correspond to one internal node, and every partitioned subspace will correspond to a leaf node. Similarity searches consist of a simple binary search in which the query object is compared to the partitioning hyperplane to see which side it is on, and follow the corresponding path down the tree. It is noted by Gaede and Gunther that although BSP-Trees are very adaptable to different types of datasets and distributions, they are often unbalanced and hence very deep, causing degenerative search quality [GG98].

2.1.4 Quadtrees

Quadtrees [Sam84] are an augmentation of the k-d tree, as they partition space using iso-oriented hyperplanes, with the difference that space is now partitioned into 2^d subdivisions at each node, instead of a binary tree; the partitions need not be of equal size. Quadtrees are not always balanced, since the termination condition for partitioning a subspace is usually met when a certain minimum number of objects is contained in the space; thus, more densely populated regions have deeper corresponding subtrees [GG98]. To search a quadtree, you simply examine the **d** subtrees to check which need to be further searched, based upon whether or not the query object intersects with the represented subspace.

2.2 Distance-Based Indexing

In this section we briefly mention some common methods for distance-based indexing. Here again, some of these structures are point access methods and others can support spatial data. All distance-based index structures are built with a specific distance function (usually one of the LP-Norms) and this distance function must be used during similarity queries; if another distance function is required for a query, the structure must be rebuilt using that distance function to accommodate the search.

2.2.1 VP-Trees

Vantage Point Trees (*VP-trees*) [Yian93] and their variants are one of the most popular point access methods for distance-based indexing. In essence, VP-trees are binary search trees in which at each level of the tree, space is partitioned based on the distances between the objects and an arbitrary pivot (vantage point) object. In each node, the vantage point and median distance between the vantage point and all other points is recorded. To perform queries, we can prune away branches of the tree by computing the distance between the query object and the vantage point, offset this distance by the query radius, and compare that value to the median distance at the current tree level. The structure is static, which is perhaps the largest disadvantage of it; however, it has been studied well and improved upon a great deal, and remains an effective tool for indexing. In section 4 we cover the VP-tree in greater detail.

2.2.2 MvP Trees

The Multi-Vantage Point Tree (*MvP Tree*) [BO97] was proposed as a way of mitigating a few shortcomings of the common vantage point tree, most prominently the low fan-out factor and the amount of distance computations necessary for range queries. The MvP tree’s strategy for overcoming these problems is to use two vantage points at every node, effectively simulating two levels of a normal vantage point tree at every level; also, the fan-out of MvP trees is a parameter of the construction algorithm. The pruning abilities are improved by storing **p** distances in the leaf node objects at construction, such that these pre-computed distances can be used to avoid distance computation at query time. It is noted by Bozkaya and Ozsoyoglu that by increasing the fan-out parameter **k** to be very large, most of the objects can exist in the leaf nodes, where the potential filtering takes place [BO97]. In this case, the pruning power of the MvP tree may be exploited to the fullest by avoiding as many internal node distance computations as possible.

2.2.3 GH-Trees

Generalized Hyperplane Trees (*GH-Trees*) [Uhl91] are also pivot based binary search trees, but two pivots are chosen at each node instead of one. At each node, the space is partitioned into two divisions each corresponding to one of the pivot objects. In this way, for every object in the given space, whichever pivot object it is closer to, it becomes a member of the partition for which that pivot object is associated with. In this way, the two pivot objects form a (**d-1**)-dimensional hyperplane that divides the space in two [HS03]. Because this dividing hyperplane is not axis aligned, it was labeled as "generalized hyperplane partitioning" in some published works following the algorithms initial invention [Brin95; BO99] (it was not named so originally). Similarity queries can be performed as a binary search, at each node of the tree comparing the distance of the query object to the pivot objects.

Many variants of the GH-Tree have been proposed to improve the pruning power of the algorithm. The Bisector Tree [KM83] was suggested as a way of improving the pruning by defining a maximum distance for all pivot objects such that no object exceeding this distance could exist in its subtree. This effectively creates "covering balls" [HS03] for each pivot, so that objects with covering balls farther away from the query object than the current search radius can be pruned. An even further extension of this is the Monotonous Bisector Tree (MB-Tree) [NVZ92], in which every node inherits its parent node's pivot object and covering ball as one its own pivots, in order to ensure stability as the tree descends.

2.2.4 GNAT

Another advanced indexing technique that can be considered a generalization of the GH-Tree is the Geometric Near-Neighbor Access Tree (GNAT) [Brin95]. Here, instead of one or two pivots, **m** pivots are chosen, creating an *m-ary* tree, where m is arbitrary. The major difference in this indexing method is that information is stored at every node pertaining to the distance ranges between the pivot objects and the rest of the objects in their sub-space. This range information is used to increase pruning power at the expense of extra cost during construction and storage space for the structure itself.

2.2.5 M-Trees

The M-Tree [CPZ97] was designed to be a distance-based indexing method which would overcome the obstacle of being static by emulating the index structure of the dynamic R-Tree. Whereas R-Trees use minimum bounding rectangles to contain subregions of the metric space, the M-Tree uses “balls” surrounding each pivot object (also called “routing objects”) to contain the space [CPZ97]. Like the GNAT, M-Trees are m-ary, and contain some distance information in the nodes to increase pruning power. However, unlike GNAT, M-Trees store all objects in the leaf nodes, whereas all internal nodes are references to child nodes [HS03]. Since the structure is dynamic, insertions and deletions are made without reconstruction, such that new entries are inserted into leaf nodes, and overflows cause splits that can propagate back up to the root of the tree. One variation that has been proposed is the M+-Tree [ZWYY03] which implements the notion of a “key dimension”, in which nodes are further partitioned into “twin-nodes” in order to increase the fan-out of the tree, and hence increase query efficiency.

2.3 Dimensionality Reduction

Now we will present some techniques for dimensionality reduction and hyperspace approximation that are aimed at assuaging the “Curse of Dimensionality”. The terminology “dimensionality reduction” has been applied to various algorithms that do very different things; we will assume it to refer to any method for simplifying or approximating a large metric space of high dimensionality.

2.3.1 Z-Ordering

Z-Ordering [OM84] is an approximation technique for spatial data that results in the ability to index the approximated regions with point access methods, often a simple B+-Tree [Com79]. The algorithm is similar to k-d trees, in that the space is first partitioned by (**d-1**)-dimensional hyperplanes. Again, these hyperplanes are iso-oriented and alternate among the **d** possible directions, but now the space is always divided into two parts of equal size. This partitioning process may continue until a certain desired level of accuracy is attained, and the end result will be that the spatial data is approximated by a grid-like structure of d-dimensional rectangles (hypercubes). Each rectangle in space is called a z-region, and is assigned a unique bit string to identify it based on its position in space [GG98]. At each sub-division, you check the position of the z-region in comparison to the partitioning hyperplanes, in fixed order, and concatenate a “0” or “1” onto the end of the bit string depending on its position. For example, in 2-d space the first partitioning routine subdivides the space into 4 quadrants. If the z-region in question is in the upper-left quadrant, we start with “01” because it is to the left of the vertical line (yielding a 0) and above the horizontal line (yielding a 1). If the upper-left quadrant is further partitioned, and within that rectangle the z-region lies in the bottom-right quadrant, this concatenates a “10” onto the “01” resulting in “0110”. This process continues to the lowest level of the partitioning. The z-region bit strings are then used as an index by a PAM such as a B+-Tree. One benefit of this method is that of fast indexing on potentially large spatial data. Another good aspect of this process is that a collection of rectangles may be a better approximation of spatial data than one single bounding rectangle, as is in the case of the R-Tree and similar structures. In addition to this, there’s a double-edged nature to the specificity of accuracy to z-ordering. The quality of the approximation is bound to the density of the partitioning, but too many subdivisions will cause the index to grow in size and reduce performance. It is suggested that there is an optimum amount of partitioning that can be achieved to balance these conflicting values [OM84]. This is covered in some detail by Gaede in his paper on optimal redundancy [Gae95].

2.3.2 FastMap

FastMap is a “retrieval and visualization tool” for large databases, and has the promising ability to map an arbitrary dataset as points in k-d space given no input except a metric distance function [Fal95]. As FastMap is central to the topic of this thesis, we cover it in detail in section 3.

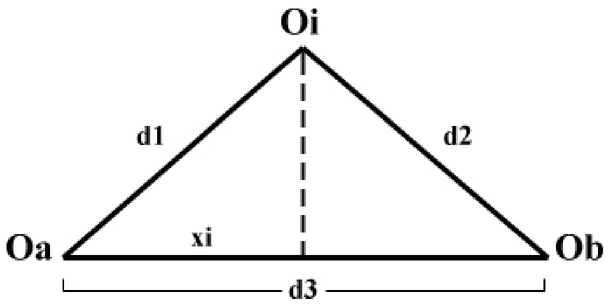
3. Fastmap

FastMap was proposed in 1995 by Christos Faloutsos as a dimensionality reduction technique to provide a “retrieval and visualization tool” for large datasets [Fal95]. The primary advantages to this proposal were those of speed and convenience. FastMap is convenient because it will map objects into points in k-d space (where K is arbitrary) even if the objects in question have no clear notion of dimensionality, such as in the case of a sequence of characters. The only input the algorithm requires other than the objects themselves is a metric distance function (such as the Euclidean Distance) in order to measure the dissimilarity between objects. FastMap is also (as the name implies) fast, with a construction cost of **O(N*K)** and a range query cost of **O(K)**.

3.1 How FastMap Works

The essence of the FastMap algorithm is to examine the metric space in a current dimensionality and choose two “pivot” objects, preferably two objects that are very far away from each other in the given space [Fal95]. Once these pivot objects are chosen, the entire dataset is then projected onto a plane that is perpendicular to the line which runs through the two pivot objects. This is repeated until the desired dimensionality is achieved. In the simple case, one could imagine a 3-dimensional sphere which contained many points in 3-space being flattened into a circle, such that the positions of all the new 2-d points would depend on how you rotated the sphere before flattening it; this rotation would be analogous to how you choose the pivot objects.

The key to the usefulness of this process is that when an object is being mapped, its new k-dimensional coordinates are computed by using only the distances between the object and the pivot objects. Figure 3.1 illustrates this process.



$$xi = \frac{d1^2 + d3^2 - d2^2}{2(d3)}$$

Figure 3.1: How FastMap calculates k-d coordinates.

In Figure 3.1, object **Oi** is being mapped into k-d space using pivot objects **Oa** and **Ob**. FastMap iterates **k** times, such that in the first iteration we calculate the first dimension (for example, x-coordinate) of the new image **Xi** that will represent **Oi** in k-d space. The value of this dimensional coordinate is calculated by the formula in Figure 3.1 which exploits the cosine law to obtain the distance **xi**. These values of xi are in fact the new dimensional coordinates in k-d space. The only other caveat to take into account is that FastMap is contractive, such that the inter-object distances are less than or equal to the original distances. In this way, whereas we could apply the above formula as stated to achieve the x coordinate (x1), the y coordinate would subtract the previous value such that x2 would be equal to (x2-x1). Continue until all **k** coordinate values are computed.

3.2 FastMap for Visualizing High-Dimensional Data

One point that has been convincingly asserted by Faloutsos is that FastMap is useful for visualizing high dimensional data, making it easier to identify potential clusters in the data, as well as other correlations between data attributes that can be used in data mining [Fal95]. As a precursor to testing FastMap applied to SAMs, we reproduced one of the experiments done by Faloutsos to confirm and illustrate its behavior. Figure 3.2 is a dataset provided by Duda and Hart [DH01], consisting of 30 points in 3-Dimensions, forming a spiral. Then, in Figure 3.3 we see the results of FastMap in 2-Dimensions, which clearly have retained a large amount of the information about the dissimilarity between the objects.

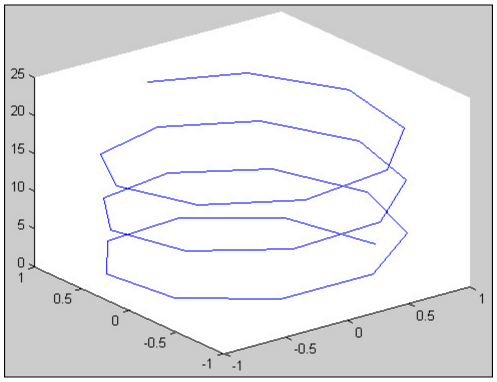


Figure 3.2: Spiral Dataset originally in 3D.

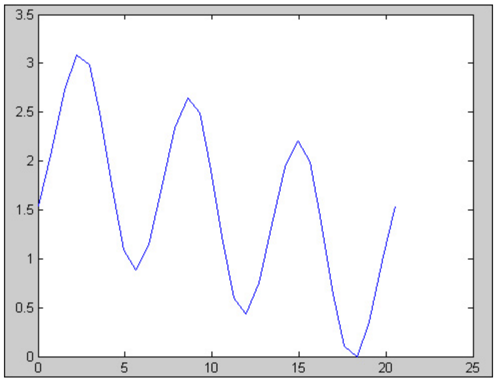


Figure 3.3: Spiral Dataset mapped to 2D.

3.3 FastMap with Non-Dimensional Datasets

One of the most significant advantages of FastMap is that datasets and query objects with no notion of dimensionality can be used. An example of this could be a string of characters, such as a DNA or protein sequence, or a word out of a large text. While this versatility is extremely useful in potential application, all of our experiments in this thesis will concentrate on vector space examples with a definite notion of dimensionality. The reason for this is that we are primarily interested in illustrating the performance of FastMap with regard to the amount of dimensionality reduction taking place. In the case of character sequences, it's very difficult to define exactly what that amount is. However, for all practical purposes, FastMap's performance in the case of high dimensional vector space should directly translate to its performance with non-dimensional data as well. There are certainly techniques, primarily in distance-based indexing, that provide efficient indexing of string data (see Sahinalp et al.) [STMO03]. These methods should have similar usages of filtering techniques for their datasets.

3.4 Near-Neighbor Queries using FastMap

FastMap has been shown to be truly useful for providing intuitive visualizations of large metric spaces in lower dimensions, providing some insight as to the geometry and clustering of the dataset. However, it is not entirely clear as to whether or not it is useful for range queries in an arbitrary dataset. To clarify this somewhat, we provide the following examples.

To begin with, we want to examine the loss of precision that occurs as a result of our FastMap dimensionality reduction. We know that if we have a dataset in 10 dimensions, and we map it down to 3 dimensions, we are going to lose some of the information about the dissimilarity between the objects. It is useful to examine just to what extent this takes place, and then utilize this information to postulate as to how useful FastMap will be when applied to a SAM. To get a good idea as to how much precision is lost, we perform experiments on some simple synthetic datasets. First consider the 3D dataset in figure 3.4 and its FastMap results in figure 3.5.

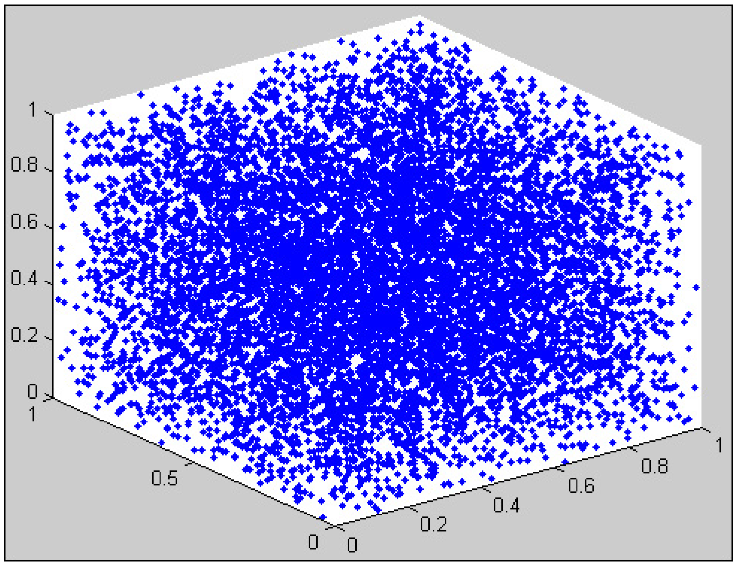


Figure 3.4: Uniform Distribution of 10,000 points in 3D vector space.

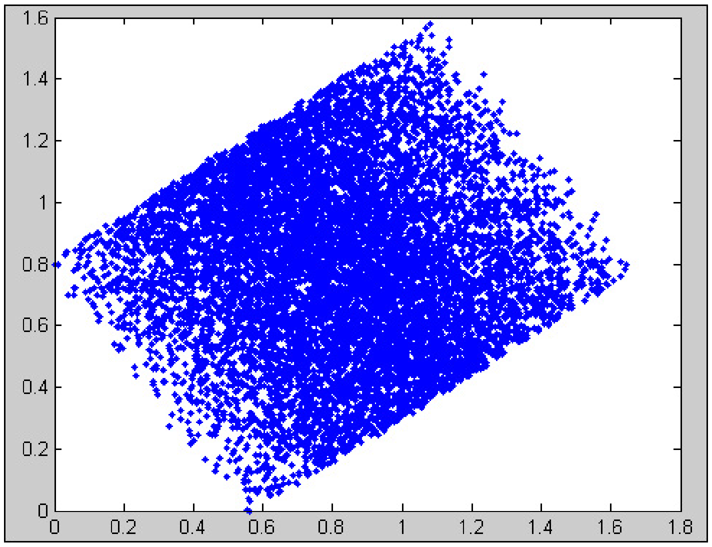


Figure 3.5: Resulting 2D dataset after FastMap is performed.

A standard near-neighbor range query states that all objects that are within a user-defined distance from the query object should be returned in the search. Thus given a radius **R** and query object **Op**, we wish to retrieve all objects such that **D(Op,Oi) <= R** (for i = 1,…,N).

Notice that although the dimensions of the dataset in Figure 3.5 have been reduced, the search radius is still the same. It is easy to see that a search in the 2D set with the same radius will return a greater number of objects. Although it might not be a large difference in this case, if the original space was a hypercube of 10+ dimensions, even a very small radius (that is, a radius that would be the right size to retrieve a small percentage of the database) could engulf the entire dataset when mapped down to 2 dimensions. We are reminded that the volume a sphere with a radius of 2 in 20-D space is a million times larger than a sphere with a radius of 1 [Brin95]. With this in mind, it is clear that depending on how many dimensions we have reduced, we are going to retrieve a lot more erroneous objects after FastMap due to the large radius. We won't lose any data—all of the objects that would have been retrieved by a brute-force distance comparison in full dimensionality will be in our search results—however, we will have a large amount of extra data that we definitely do not want. For each such object, we must perform a distance computation against the query object to assert its membership in the query results; therefore it is optimal to minimize erroneous object retrieval. So the question remains, can we reduce the amount of erroneous objects retrieved, and if not, how useful is FastMap for range queries given this loss of precision?

3.5 Axis Shifting Resulting from FastMap

Since FastMap is contractive such that inter-object distances remain the same or decrease, it is the case that the original radius is approximately the same as the distance between the mapped query object and the furthest point from it in the target space. FastMap guarantees only that the distance information will be preserved, not the intrinsic geometry of the data, since the randomness of choosing the pivot objects at each dimensional reduction results in a random shifting on the dimensional axes. While this may not be that important when considering a spherical query, it becomes a serious issue when using a hypercube as the range query intersection region. A simple way of imagining this is the case of a 3D cube being flattened into 2 dimensions. The resulting square in 2 dimensions may have shifted on its axes, turning into a diamond shape. Because of this arbitrary rotation, one must use the half the diagonal width of the original query-region hypercube as the radius of the new query in target space in order to retrieve all objects. This is illustrated in Figure 3.6. The original cube query in 3D space is shown on the left, with a diagonal radius of **R'**, which is notably larger than the enclosed sphere's radius **r**. On the right side, it can be seen that a large circle with the radius **R'** is needed in order to retrieve the full set of a cube-region query, because of the diamond pattern rotation which can occur. It is necessary to acknowledge this, because some SAMs, like the R*-Tree, typically use hypercube region intersection queries instead of sphere regions. In the case of a sphere query though, as is the case with a Vantage Point Tree, simply using **"r"** in both cases will achieve the desired results.

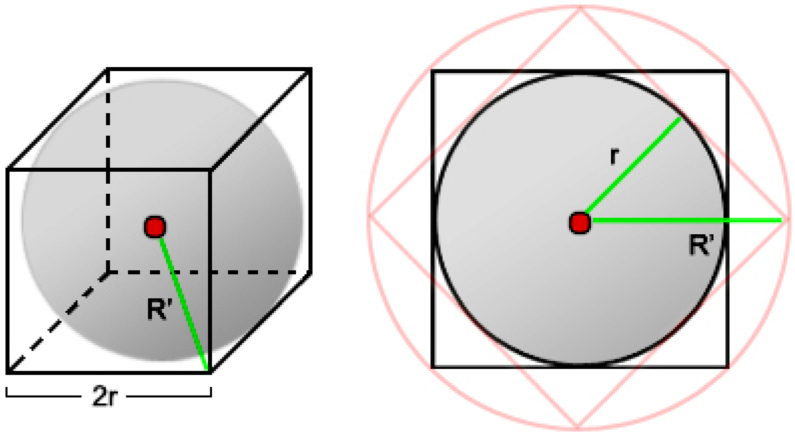


Figure 3.6: Axis shifting resulting from FastMap.

It should also be noted that in the case of such a cube query, what we are retrieving is a superset of the true query results we are after. In an R*-Tree cube-region query in 3 dimensions for instance, we will retrieve all objects that lie within the cube, but we are only interested in those objects which lie within the enclosed sphere. That means that even without FastMap, all the objects retrieved must be checked for their true distance from the query object, to eliminate the "corner" objects from the cube query. Fortunately, this works out well for FastMap, because by performing a cube query in the target space (in this case, a square in 2D) we are assured to retrieve a superset of the true query results by using a cube with a half-width of **"r"**. This can be visualized easily in Figure 3.6, because clearly any way in which you rotate the diamond, it still completely encloses the smaller circle which contains the true query results.

3.6 Using FastMap as a SAM Filter

The idea of filtering our dataset for use in a SAM is fairly simple. We know that if we map a d-dimensional dataset to **K** dimensions, and then use the resulting target space as the new dataset for our indexing method, we will have marginally faster indexing at the cost of some loss of precision. This "loss of precision" refers to the extra erroneous data we retrieve in the FastMap set. After we query the target space and retrieve the superset **Q'**, we simply have to determine which of the objects in **Q'** are in the true query result set **Q**. This would be done by taking every object referenced by **Q'** and computing the distance between it and the query object; any object referenced by **Q'** that has a distance from the query object greater than **R** will be excluded from the result set, and we will end up with the true query results **Q**.

The strong advantage of this strategy lies in the assumption that the original number of dimensions is large, and K is very small. With this assumption, we notice that all of our distance computations during the filtering process are being performed in k-d space, thus assuaging the costly nature of such distance computations. During the filtering process, we are in k-d space, and after the filtering process, we have eliminated (hopefully) the majority of the objects in the dataset as potential query matches. In this way, there should be potential for FastMap to decrease the amount of true distance computations necessary by acting as a filter when the amount of dimensionality reduction being done is significant. Using this as a strategy, we can try and apply this filtering method to SAMs, such that we will still have to perform distance computations on the original objects, but hopefully the amount will be reduced greatly by the filtering process. In this thesis, we apply this method to two indexing techniques in particular to test the results; the Vantage Point Tree, and the R*-Tree.

4. The Vantage Point Tree

A Vantage Point Tree (VP-Tree) [Yian93] is a static point access method that is used for distance-based indexing. The VP-Tree begins with a collection of objects representing a metric space, and recursively constructs itself as a binary tree, at each level of the tree partitioning the metric space into two spherical cuts. This is done by choosing a pivot, or "vantage point" object **V** among the available objects in the current space, randomly or by some chosen strategy, and then partitioning the space based on the distance of each object from the vantage point; all objects that are further away from the vantage point than the median distance are placed in the "right" half of the partition, and all other objects in the "left" half. At each node, the median distance **m** is stored, which is needed for searching the binary tree. The search routine is fairly simple. Given a query object **Op** and desired radius **r** (which represents the maximum desired dissimilarity between the given query object and all other objects in the metric space), you start at the root of the tree and follow two rules:

If (D(V,Op) – r <= m) then search left.
If (D(V,Op) + r > m) then search right.

Figure 4.1 illustrates the basic components of a near-neighbor search on a vantage point tree, showing the space partitioned into two subsections, where "**d**" is the distance between the vantage point **V** and the query object **Q**. Recursive partitions are created by utilizing the median distance "**m**". A range query with a radius of "**r**" will return all objects that lie within the circle surrounding the query object. If the circle intersects both right and left regions of the space, both will need to be searched, otherwise only the section that the circle lies in needs to be searched, and the other side may be pruned.

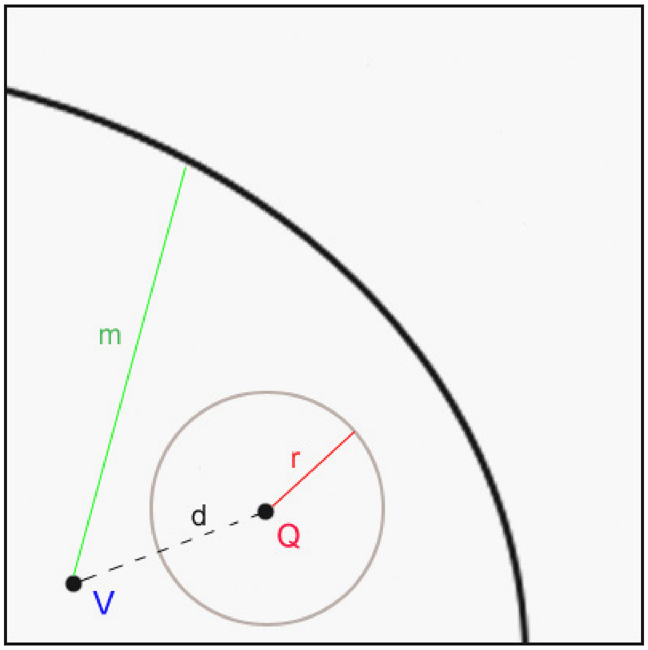


Figure 4.1: Near-Neighbor Search for Vantage Point Trees.

The performance of searching on a vantage point tree is dependent upon the quality of the vantage points chosen at each level of the tree. There are many theories and documented experiments on proposed methods of choosing vantage points; most widely known, it has been suggested that choosing two vantage points that are farthest apart from each other in the metric space may provide optimal query efficiency [BO97; BO99]. For the purposes of this thesis, choosing vantage points randomly will suffice, since we are not interested in altering the SAM itself, we are only interested in observing the effects of FastMap upon the given SAM. Utilizing FastMap as a filter should provide a marginal change in the amount of distance computations needed to index our metric space no matter how good or bad our indexing method is. Any improvement upon the inherent performance of a SAM can be made in addition to the utilization of the filter. In the case of a VP-Tree, it would seem that once it was established that FastMap did or did not improve the efficiency of range queries, any and all improvements made to the VP-Tree structure itself, such as choosing better vantage points, would be beneficial in a strictly peripheral and supplemental context.

4.1 Measuring Vantage Point Tree Performance

In order to actually measure the goodness or badness of the VP-Tree's performance, there are a few factors to take into account. The most important aspect of performance is the efficiency of querying, which directly translates to minimizing the amount of distance computations required. Secondary to this are issues like construction cost of the structure and memory usage. In addition to these, we must acknowledge that the VP-Tree is a static PAM, such that new objects cannot be added on the fly into the structure. In this way, though the VP-Tree may prove to be a very efficient PAM, it would not appear to be the best choice in a highly dynamic environment, and this needs to be taken into consideration when evaluating its performance.

4.1.1 Near-Neighbor Query Performance Metric

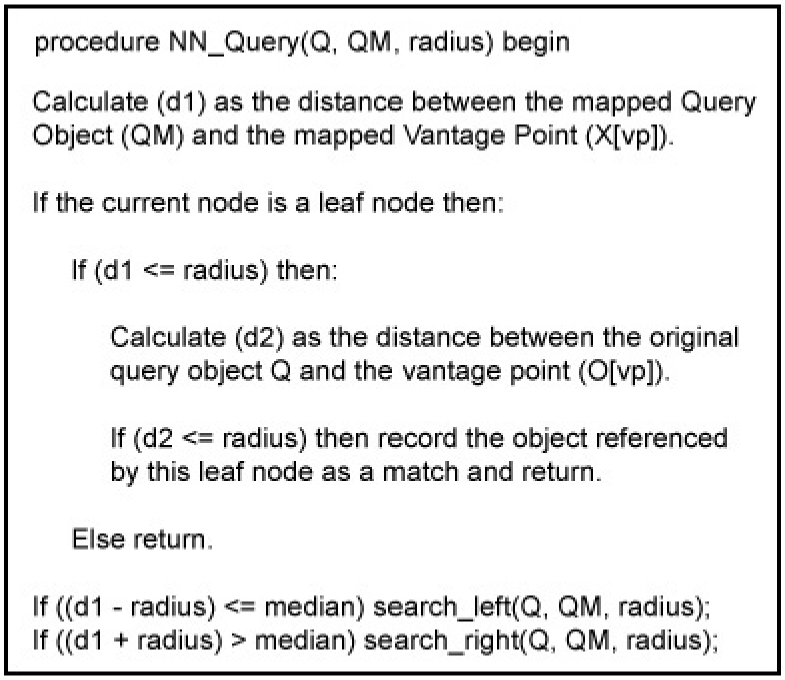
To measure the performance of a VP-Tree in terms of range query efficiency is easy. Since the search algorithm involves one comparison at each tree node between the query object and the vantage point, the resulting penalty is one distance computation per node visited. The leaf nodes typically contain a reference to one object in the dataset, so in general, the badness of the query can simply be measured by the number of internal nodes visited during the search routine.

4.1.2 Construction Costs

A normal VP-Tree must build itself as a binary tree, which of course is of height **lg(N)**. At each level of the tree, the distance is computed between the vantage point and every other object. Since this is an **O(N)** operation performed at every level of the tree, the construction cost of a VP-Tree is **O(N*lg(N))**. We recall that the construction costs for FastMap are on the order of **O(N*K)**, where K is an arbitrary dimensionality of the target space. This can be safely assumed to be slightly better than the VP-Tree's construction costs, as said, because **K** is usually very small (2 or 3), and the original dimensionality **d** is assumed to be potentially huge.

4.2 Applying FastMap as a Filter to a Vantage Point Tree

In the case of FastMap, the VP-Tree does not change, with the exception that we are now using the object images as the dataset, and we must also map our query objects into k-d space. Also, the query routine changes slightly, such that we have to make a distinction between true distances and distances in target space. This slight alteration is described in Figure 4.2. Note that the construction costs have been reduced since all distance computations performed are in target (k-d) space, which is assumed to be a significant reduction.



```
procedure NN_Query(Q, QM, radius) begin

Calculate (d1) as the distance between the mapped Query
Object (QM) and the mapped Vantage Point (X[vp]).

If the current node is a leaf node then:

    If (d1 <= radius) then:

        Calculate (d2) as the distance between the original
        query object Q and the vantage point (O[vp]).

        If (d2 <= radius) then record the object referenced
        by this leaf node as a match and return.

    Else return.

If ((d1 - radius) <= median) search_left(Q, QM, radius);
If ((d1 + radius) > median) search_right(Q, QM, radius);
```

Figure 4.2: Near-Neighbor Query Algorithm for a Vantage Point Tree using FastMap.

The important aspect of the search routine to take notice of is that we never perform a true distance computation unless we first discover that one of the images is within the specified distance of the mapped query object. In this way, we have filtered out any images that are not in the superset of the true query results.

4.3 Measuring VP Tree Performance with FastMap Integrated

When testing the performance of a VP-Tree or any spatial access method that has utilized FastMap (or any other dimensionality technique) it is important to make the distinction between true distance computations as opposed to distances between the mapped images. Any metric which measures query efficiency or construction costs must focus on minimizing true distance computations, and regard image distances as present but potentially insignificant. This is the central caveat of analyzing FastMap results.

4.3.1 Near-Neighbor Query Performance Metric

With respect to range queries, our metric for testing the efficiency of the queries must be modified slightly because of FastMap utilization. We know that FastMap is an approximation, such that given the same radius in the target space we will retrieve a set of object references that is a superset of the true range query in the original space. This superset of the true query results, which we will call **Q'**, is retrieved by performing a range query on the VP-Tree which was constructed with the object images being used as the input dataset. During the range query, whereas before we simply counted the number of internal nodes visited, we now perform distance computations in k-d space, which may or may not be considered consequential in terms of computational burden, depending on how small K is. The first thing we acknowledge then, is that the size of the set **Q'** now indicates a number of mandatory distance computations we must perform in order to determine which objects referenced by the superset are in the true query results.

The goal is to achieve the true query set, which we will call **Q,** and to do this we simply take the original unmapped objects referenced by **Q'** and compare them to the unaltered query object. In this way, we must now additionally perform a number of true distance computations in the original space that corresponds to the size of **Q'**. The implications of this are that whereas a normal VP-Tree is simple to interpret, as it can be measured by how many distance computations are required, using FastMap requires a metric which is not quite as definitive in terms of badness, because it depends on how costly distance computations are in k-d space. In essence, it just depends on what **K** is. To illustrate this, take two theoretical queries:

VP-Tree Query: Objects Found = 100. Distance Computations = 1000.
Using FastMap: Objects Found = 300. Distance Computations = 2000.

In the Vantage Point Tree we see there are 100 true query results, requiring 1000 distance computations to retrieve in total. In the FastMap set, we've retrieved the same 100 objects plus 200 extra objects which are not in the true query set, for a total of 300. The cost of this was 2000 distance computations in k-d space. From there, we need to take the 300 objects retrieved, and perform true distance computations between those and the true query object to determine which are in the true result set. So in the case of FastMap here, we have 300 true distance computations plus 2000 distance computations in k-d space, as opposed to 1000 true distance computations. Obviously in a situation where **d**-dimensional vector space is being mapped to **K** dimensions, if **d** is much larger than **K**, FastMap will have made a large improvement, since we have much fewer true distance computations. But in a case such that **d** is very large, and **K** is also very large, FastMap would have not been worth the effort, because the k-d distance computations won't be all that much better than the true distance computations. However, in all our empirical data we will assume that **d** is very large and **K** is very small, because this at least proves any potential for improvement that FastMap has. It should be noted again that there may be no notion of original dimensionality at all, as in the case of character sequences. While there is no universal way of measuring how much you are reducing dimensionality in this case, since the whole idea of dimensionality becomes fuzzy, it still holds true that a small value of **K** should produce a large potential for improvement, since a large degree of mapping will take place.

4.3.2 Construction Costs

In the case of a VP-Tree utilizing FastMap at construction, we have eliminated all of the true distance computations in the construction of the tree, and replaced them with distance computations between the images instead of the original objects. Since the assumption is that our original objects are very large in dimensionality with expensive distance computations, and that our target space is of a reasonably low dimensionality, this appears to make the construction costs of the vantage point tree significantly reduced in terms of computation. It must be at least noted though that we do require **O(N*lg(N))** distance computations in k-d space to construct the VP-Tree. More significantly, we have to account for the construction costs of the FastMap routine. In this case we have **O(N*K)** true distance computations, and also an amount of memory required to store the new, low-dimensional dataset. Since the memory issue is not largely important, it is fairly clear that we have replaced the **O(N*lg(N))** true distance computations of the VP-Tree in exchange for the **O(N*K)** construction costs of FastMap, which is generally better. Of course, we are interested in integrating FastMap with other structures besides the VP-Tree, and as such, it follows plainly that as long as construction costs remain linear (or at the worst, **O(N*lg(N))**), the use of FastMap is progressive in terms of reducing computation costs. Since FastMap is indeed linear in construction cost, and we assume that the original construction costs of the SAM in question without using FastMap were linear or worse in the order of true distance computations required at construction, it clearly follows that using FastMap should never increase the order of the construction algorithm, and usually improve it, as is the case here with VP-Trees.

5. The R*-Tree

R*-Trees [BKSS90] are an extension of the R-Tree proposed by Antonin Guttman in 1984 [Gutt84]. In general, all variations of the R-Tree work by recursively subdividing the metric space into regions enclosed by the minimum bounding rectangle (MBR) of a group of objects or points. R*-Trees perform near-neighbor range queries based these MBR hypercubes, ignoring (pruning) all sub-trees that are associated with the non-intersecting MBRs. Because of the nature of the MBRs, R*-Trees can support complex d-dimensional spatial objects (e.g. polygons, polyhedra) in addition to common points in d-space, since all that is required is that the objects must be containable within a hypercube MBR. Queries are performed by doing simple recursive intersection or containment queries at each branch of the tree, by using the corresponding MBR. Since FastMap is only useful for point queries, we will assume our R*-Tree uses a containment query for all data points that lie within the hypercube given as the query region.

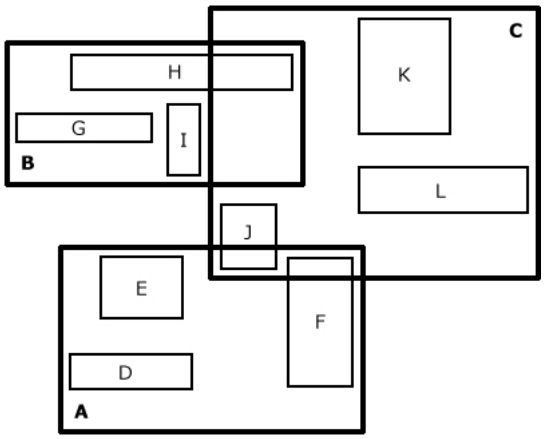


Figure 5.1: MBRs that form an R*-Tree.

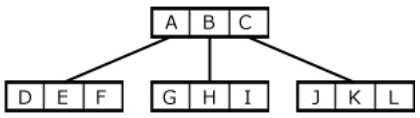


Figure 5.2: The tree structure of Figure 5.1.

An important thing to note is that these types of R*-Tree range queries have a fairly significant problem when the number of dimensions of our search space grows large. The R*-Tree typically uses a hypercube as the containment region for the range query, but what we are really interested in is a hypersphere region query given a radius; obviously a sphere is the geometrical object which envelopes all near-neighbor objects within a defined distance, not a cube. If you imagine a perfect sphere in 3D space with a radius R, and then imagine a perfect bounding box around that sphere, then this describes the query problem we are posed with. What we will retrieve is all the points within the cube, but that's more than we want; we are only interested in the objects that lie within the sphere. Thus, after the hypercube query retrieval, we need to take every object retrieved by the query and re-assert as to whether or not it is in the true query results. Notwithstanding the possibility of a more sophisticated technique of dealing with these, this will require a true distance computation for every object retrieved to check its validity.

With this in mind, the biggest problem of the R*-Tree as the number of dimensions increases lies within the question of how much extra data do we retrieve in the "corners" of the cube in comparison to what we actually need from the enclosed sphere region? That is, how much larger in volume is the bounding cube than the sphere?

The volume of an **(d)** dimensional hypersphere with a radius of **(R)** can be calculated by the following formula, and the corresponding formula for the area of its bounding box hypercube is trivial, since the length of each side is double the radius **(2R)**.

Volume of a Hypersphere:	**Volume of its Bounding Box Hypercube:**
$\dfrac{R^d \cdot \pi^{d/2}}{\Gamma\left(\frac{d}{2}+1\right)}$	$2R^d$

Taking just a couple of examples into consideration, it’s apparent just how huge this difference is in high dimensional space. In 10-D space, the volume of a hypercube is 400 times larger than the sphere it encloses. In 20-D space, it is 40 million times larger. With that kind of exponential growth in the size of erroneous data retrieved, R*-Trees using no other techniques to reduce dimensionality or modify the method of indexing seem to be fairly unusable for these kinds of range queries. It is obvious from the structure of the R*-Tree itself that the cost of an intersection or containment query can be worse than an exhaustive search without the use of a SAM altogether. This would make the use of the R*-Tree counter-productive in the case of datasets consisting of objects very high in dimensionality. The question for the purposes of this thesis will be as to whether or not FastMap can help solve or improve upon this problem.

5.1 Measuring R*-Tree Performance

Once again, we are interested in primarily measuring the efficiency of near-neighbor queries, which now are MBR intersection or containment queries, and also construction costs of the SAM. Also, in contrast to the static VP-Tree, R*-Trees are dynamic. An R*-Tree is initially built by a sequential series of “Insert” operations on the objects, and in the same way, new objects may be inserted into or removed from the tree at any time without damaging the tree’s intrinsic searching speed; in fact, it has been suggested that removing and re-inserting objects within an R*-Tree can actually improve its performance [BKSS90]. The dynamic nature of R-Trees is a strong advantage towards versatility, and thus has lead to a great deal of study with interest in improving R-Trees in general.

5.1.1 Near-Neighbor Query Performance Metric

In order to compare the results of a normal R*-Tree with those of a FastMap integrated R*-Tree, we must first try and define a metric that suggests the goodness or badness of range queries. Whereas VP-Trees are binary and require one true distance computation (usually a simple Euclidean distance calculation) at each node of the tree, R*-Trees are slightly different in that they perform bounds checks at every dimension of an object, which is not perfectly equivalent algorithmically or in processing time to the direct method of using Euclidean distances. Furthermore, R*-Trees are generally parameterized as to the maximum number of children for internal and leaf nodes. For every node visited, the dimensional bounds checks must be performed for every child. In this way, if the maximum number of children per internal node for an R*-Tree was 20, then it would be a reasonable approximation to state that each internal node visited in the tree would constitute the same computational cost as visiting 20 nodes of a VP-Tree formed from the same dataset.

By this reasoning, in an attempt at forming an approximate “badness” metric for a parameterized R*-Tree with **(Imax)** maximum children per internal node and **(Lmax)** maximum objects referred to by leaf nodes, it would be fair to estimate that the number of internal nodes visited plus the number of leaf nodes visited, each multiplied by their respective number of children or objects, is approximately equal in computation time to the same number of Euclidean Distance computations in a Vantage Point Tree.

In addition to these costs, we must take into account that we are retrieving the hypercube region query, and must go through every object retrieved and perform a true distance computation in order determine if it lies in the desired true sphere query set.

Our metric for measuring the total cost of an R*-Tree, then, is defined as follows, and aims at estimating “cost” as being computation cost in terms of the number of true distance computations performed in the original space.

I = (Internal Nodes visited * Imax)
L = (Leaf Nodes visited * Lmax)
O = Number of objects retrieved.

Total Cost = I + L + O

Figure 5.3: Estimated cost for an R*-Tree Intersection or Containment query.

5.1.2 Construction Costs

Construction of an R*-Tree on a collection of N objects is a sequential series of “Insert” operations on each object. It is noted that the insertion operation is very similar to B-Tree [Com79] insertions, as the references to the new object are added to leaf nodes, and node splits that occur from overflows are propagated back up to the root of the tree [Gut84]. Since this operation is approximately on the order of **O(lg(N)*Imax)**, where N is the total number objects, and Imax is the arbitrary maximum capacity of internal nodes, the order of the entire construction operation would be **O(N*lg(N)*Imax)**. This is slightly worse than a VP-Tree, unless one has specific knowledge of Imax and considers it to be a constant. Even so, with the dynamic nature of the R*-Tree, a construction cost of this magnitude is generally acceptable, as the primary motivation is query efficiency.

5.2 Applying FastMap as a Filter to a R*-Tree

FastMap by definition maps objects as points in k-d space, so our R*-Tree is now under the constraints that it consists of points in k-d space, and all near-neighbor queries will be MBR containment queries for these points. From this point, the only caveat is how to form a hypercube for the containment query which will contain all the desired images, that is, all the points within distance "R" from the image of the query object. As is illustrated in Figure 3.5, we require a perfect bounding box of the sphere of radius R. Given this bounding box as the containment query region, our query will return the superset **Q'** which will contain the true query results **Q**, plus additional corner objects.

5.3 Measuring R*-Tree Performance with FastMap Integrated

The case of FastMap for R*-Trees is simple, since as before, we consider any distance computation done in k-d space to be potentially inconsequential, and generally peripheral in the total cost. As before, this is under the assumption that **d** is large and **K** is small, so as to demonstrate the potential of FastMap to reduce costly distance computations.

5.3.1 Near-Neighbor Query Performance Metric

Since the objects indexed by the R*-Tree are points in k-d space, we retrieve our query result set **Q’** with no true distance computations necessary, but a number of distance computations in k-d space (KDCs) that is calculated by the metric given in Figure 5.3. Once again, we take the set **Q’** and perform true distance computations between every object referenced by it and the image of the query object, such that we may attain the true set **Q**. The total cost (the number of true distance computations performed) of an R*-Tree near-neighbor query when using FastMap, then, is simply the size of the set **Q’**.

5.1.2 Construction Costs

To construct an R*-Tree of size N is simply to perform an Insert operation on the N objects sequentially. This process remains unchanged, but now we are in k-d space, such that the order of the insertion algorithm may or may not still be of consequence, depending on what K is. This cost, as stated in section 5.1.2, is **O(N*lg(N)*Imax)**. However, we now have the added cost of FastMap itself, which is always **O(N*K)**. This represents true distance computations, and is the primary residual construction cost.

6. Experimental Results

Our experiments required implementations of FastMap, Vantage Point Trees and R*-Trees. We implemented FastMap and VP-Trees in Java 5.0 on a Windows XP machine, and also in C++ on the same machine, to double-check the results. The R*-Tree implementation was provided by Marios Hadjieleftheriou's "Spatial Index Library" (http://www.cs.ucr.edu/~marioh/spatialindex/) also compiled and used with Java 5.0 in Windows XP.

Our experiments had two groups. In the first group, we aimed at collecting data from a typical vantage point tree tested on various types of datasets, which are described below, some synthetic and some real. Two different methods of querying were used, the first being to choose query objects as random points in vector space, and the second being to use a random existing object as the query object. This provides information as to the performance based on the amount of dissimilarity between the query object and the existing data. We then proceeded to take aggregate samples of these parameterized queries (with multiple repetitions for varying radii) and compared the querying efficiency to the same exact query sets using FastMap as a filter for the VP-Tree.

The second group of experiments was identical, but now utilizing the R*-Tree instead of the VP-Tree. After presenting the results of each group of experiments, we go on to provide comparisons between the VP-Tree and the R*-Tree (with and without FastMap) with regard to their performance as the number of dimensions increases. In this last case, we only use synthetic datasets of uniformly distributed points in vector-space, but for both the major groups of experiments, all of our synthetic and real datasets are utilized and the results are expounded upon.

Dataset Descriptions

Dataset 1: ***Color Moments* (9-D)**
68,040 photo images from various categories.
Feature Vector Dimensions: 9 (3 x 3)
Features include Mean, standard deviation, and skewness.
(Each having an H,S,V in HSV color space)

Dataset 2: ***Co-occurrence Texture* (16-D)**
68,040 photo images from various categories.
Images are converted to 16 gray-scale images.
Co-occurrence in 4 directions is computed.
Feature Vector Dimensions: 16 (4 x 4)
Features include Second Angular Moment, Contrast,
Inverse Difference Moment, and Entropy.
(Each for horizontal, vertical, and two diagonal directions)

Datasets 3-10: **Synthetic Uniform Distributions**

Dataset 3: 10,000 points in 3-Dimensional vector space.
Dataset 4: 10,000 points in 4-Dimensional vector space.
Dataset 5: 10,000 points in 5-Dimensional vector space.
Dataset 6: 10,000 points in 6-Dimensional vector space.
Dataset 7: 10,000 points in 7-Dimensional vector space.
Dataset 8: 10,000 points in 8-Dimensional vector space.
Dataset 9: 10,000 points in 9-Dimensional vector space.
Dataset 10: 10,000 points in 10-Dimensional vector space.

6.1 Experimental Results for Vantage Point Trees

Our experiments in this section aim to contrast the performance of a typical VP-Tree with a VP-Tree using a FastMap filter. In all cases, the vantage points are chosen randomly, and the target space for FastMap is always **K = 3**, unless otherwise indicated. The distance metric is always the Euclidean Distance.

Figures 6.1.1-6.1.6 illustrate the improvements FastMap can make in reducing the number of true distance computations required for VP-Trees. **In all cases K=3, such**

that k-d space is a 3-Dimensional cube, with the exception of Figure 6.1.1 where the number of true dimensions is already 3, so we map down to 2 dimensions. The first 4 graphs describe uniformly distributed datasets in **d**-dimensional vector space, as stated below, and the latter 2 graphs describe datasets with significant clusteration.

One caveat is the method of providing query objects for these experiments. There are two preferable choices, the first being to randomly choose a value between 0 and 1 on each axis and use the resulting vector as the query point (as the datasets are all normalized such that the axes span between 0 and 1). The second method would be to randomly pick one of the existing objects in the dataset and use that as the query object. For the uniform datasets, these two methods are identical in empirical usefulness, because the data points themselves are generated randomly and uniformly; that is to say, no matter which random point you come up with, there will almost always be approximately the same number of near-neighbors within a given radius, since the data is distributed uniformly. The true issue arises in clustered datasets, which of course are more realistic and therefore more important to analyze. What happens here is that we see a huge difference in the performance of FastMap depending on the query point given, and this decline in performance increases with the number of dimensions. The problem lies in the fact that high-dimensional vector space is huge in volume, such that a random query point that is very far from any existing cluster (that is, an object very dissimilar from any object in the current dataset) will require a much larger radius in order to retrieve any objects in a range query.

As an example of this, in our experiments we have a 16-D clustered dataset (Dataset 2) that is mapped to 3-D. Using existing points in the data set as query objects, a range query with a radius of 0.37 on average would retrieve almost half of the entire dataset (32,222 objects out of 68,040). Using more than double that radius (0.77) for a random query object, the average number of objects retrieved was only 1.

This information illustrates object retrieval, but still says nothing of the performance of FastMap. To clarify this, we can look at the two cases just mentioned. With existing-object queries and a smaller radius (0.37) in 16-space, we retrieved 32,222 objects at the cost of 73,648 true distance computations, which is worse than an exhaustive search of 68,040. FastMap improved this to 36,518 true distance computations, which is about half the cost of an exhaustive search. In contrast, using random query objects and a larger radius (0.77) in 16-space, we retrieve an average of 1 object at the cost of 94 true distance computations. Using FastMap that became 3,377 true distance computations, which is far worse. These numbers degenerate rapidly as the radius increases. To simplify these results, we observe that a large radius high-dimensional space will retrieve far too many erroneous objects in k-d space after FastMap, if K is small. Thus, presenting highly random (dissimilar) objects to a clustered dataset as query objects will not be progressive in terms of reducing distance computations. FastMap in this case only seems to work well when we are sure that the query objects will be fairly similar to existing data points.

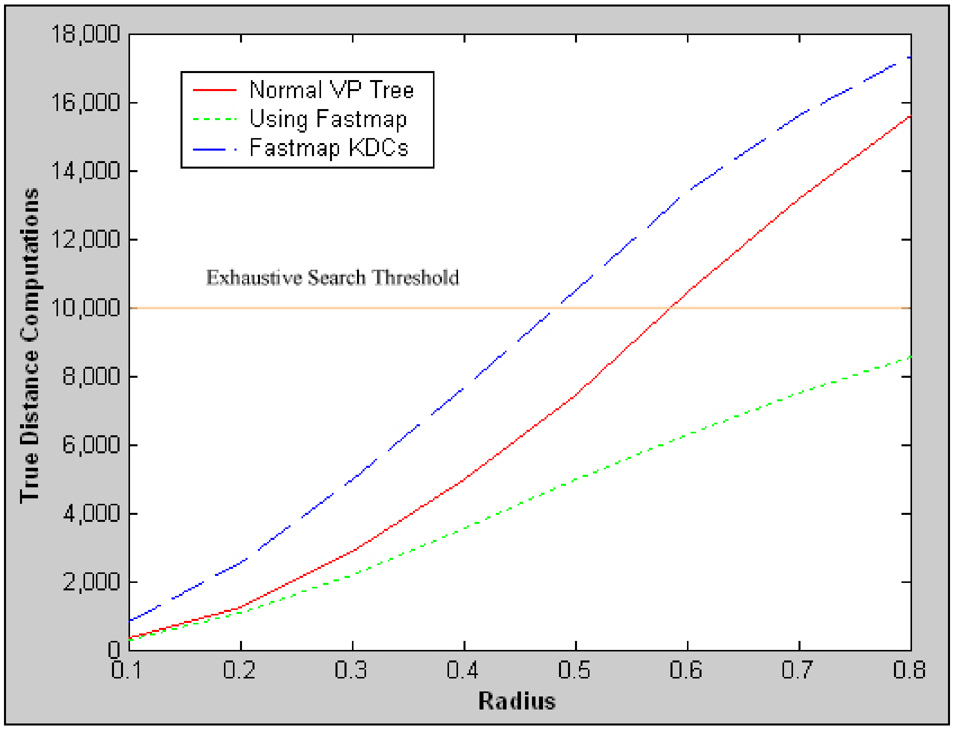


Figure 6.1.1: Uniform Distribution (3-D).

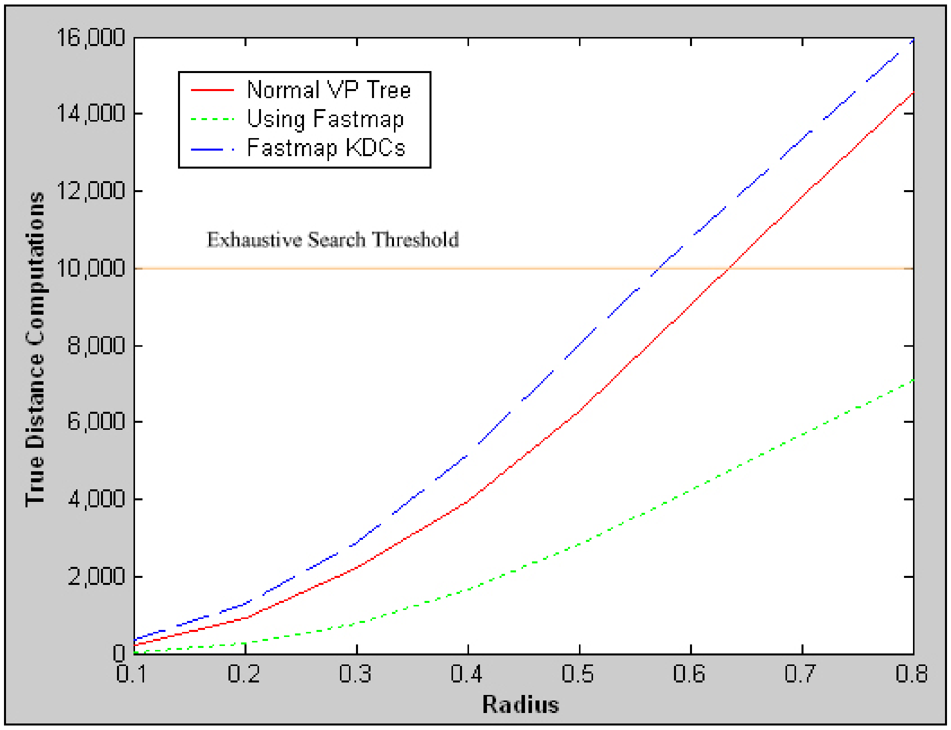


Figure 6.1.2: Uniform Distribution (4-D).

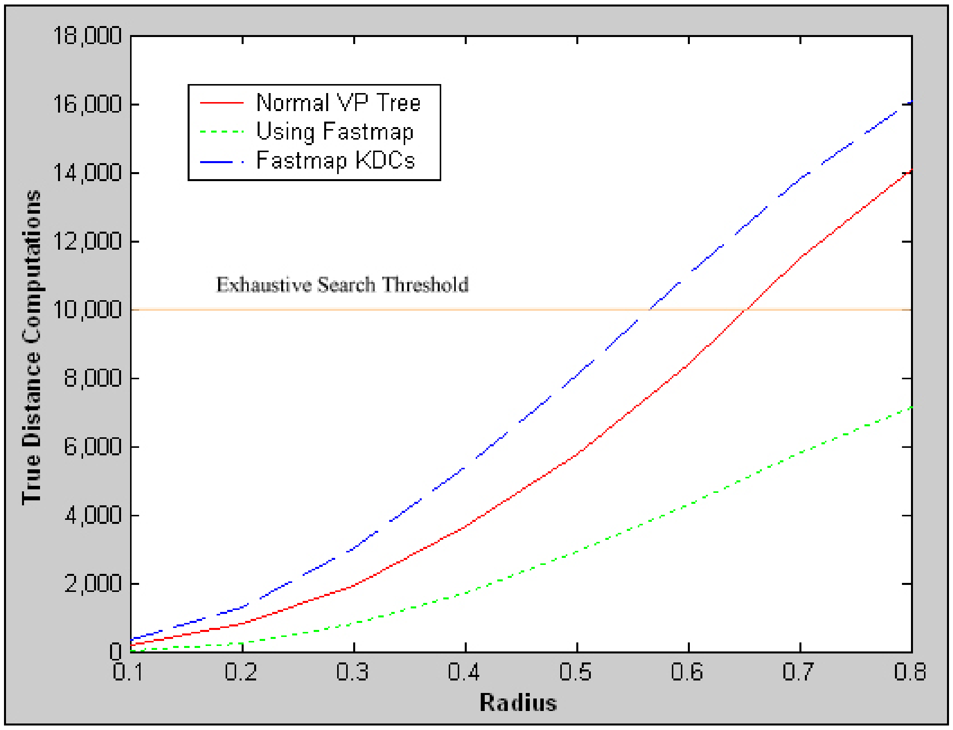


Figure 6.1.3: Uniform Distribution (5-D).

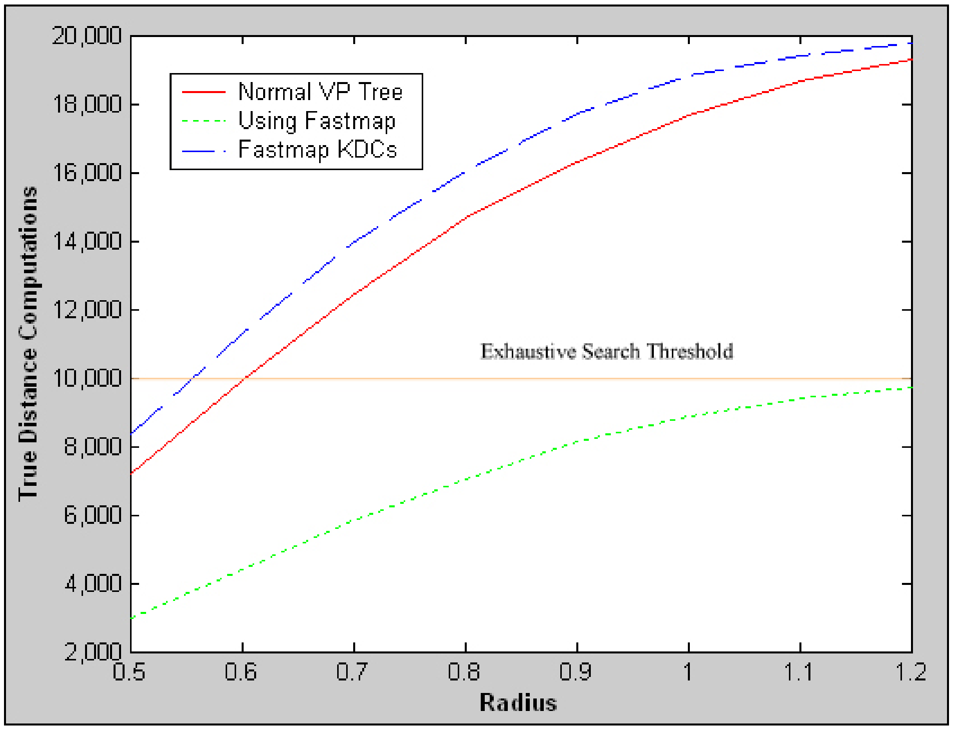


Figure 6.1.4: Uniform Distribution (10-D).

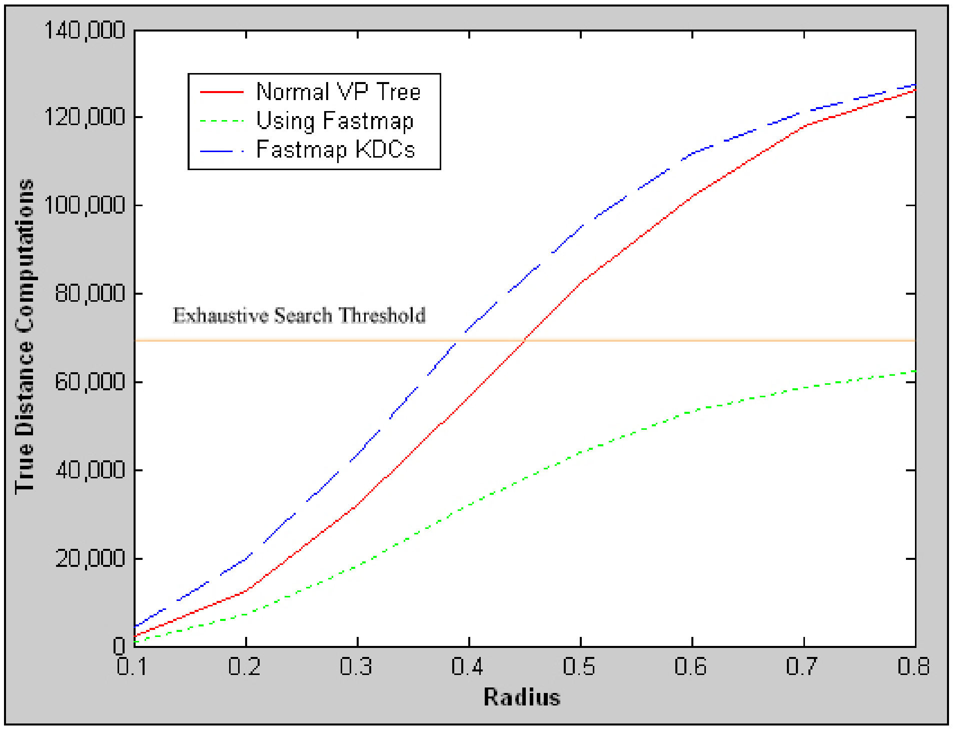


Figure 6.1.5: ColorMoments Dataset (9-D).

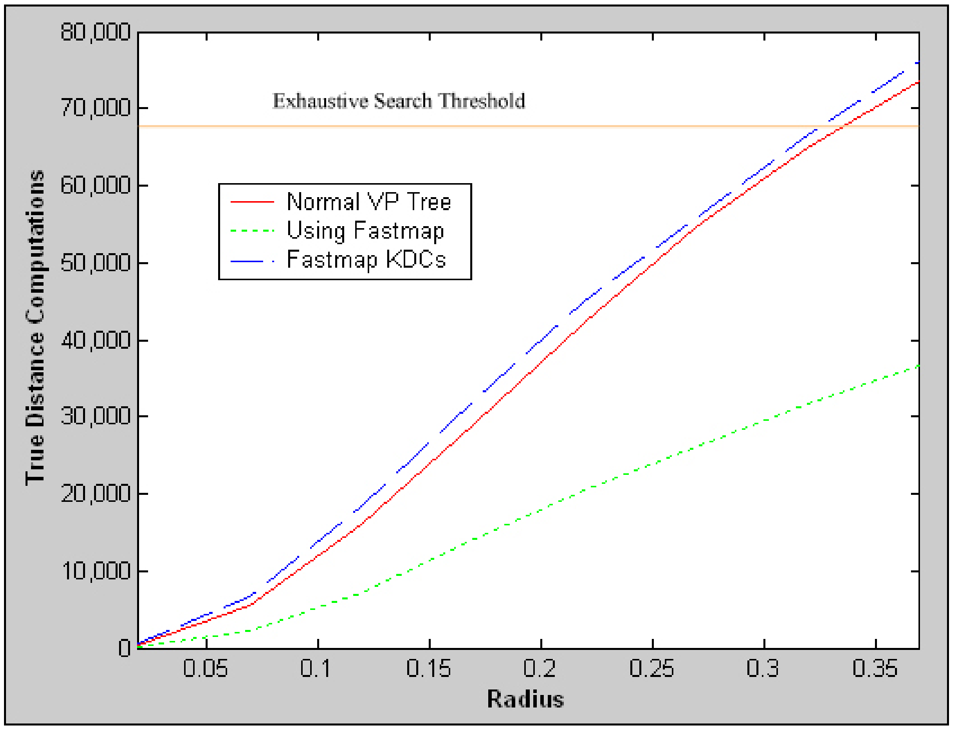


Figure 6.1.6: CO-Texture Dataset (16-D).

As opposed to the benefits of FastMap illustrated in Figures 6.1.1-6.1.6, which all utilize existing points in the data sets as query objects, we also provide Figure 6.1.7 to illustrate the poor performance of FastMap in high-dimensional space with clustered data and random query points. The random query points in 16-space require a relatively large radius to generate significant object retrieval, and thus have a propensity to retrieve an enormous amount of erroneous data in k-d space when **K** is small, which is true in this case, k-d space being 3-Dimensional.

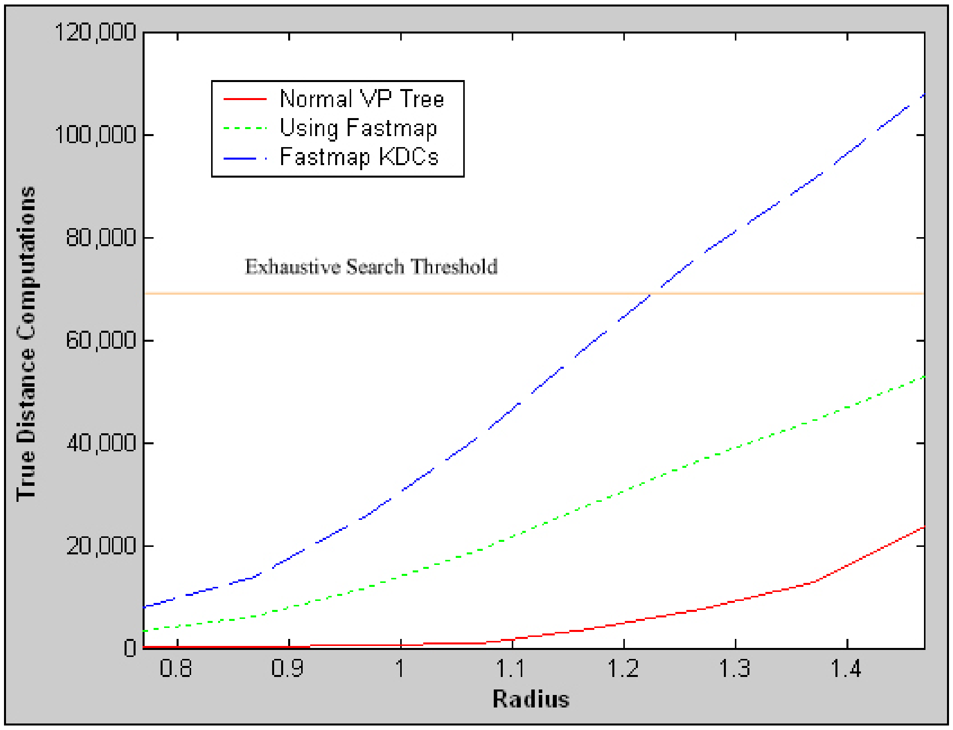


Figure 6.1.7: CO-Texture Dataset (16-D) using random query points.

To illustrate the same degeneration of performance in a lower original dimensionality, we provide Figure 6.1.8 to show how FastMap is counter-productive in the 9-D ColorMoments clustered data set when mapped down to three dimensions and using random 9D vectors as query points.

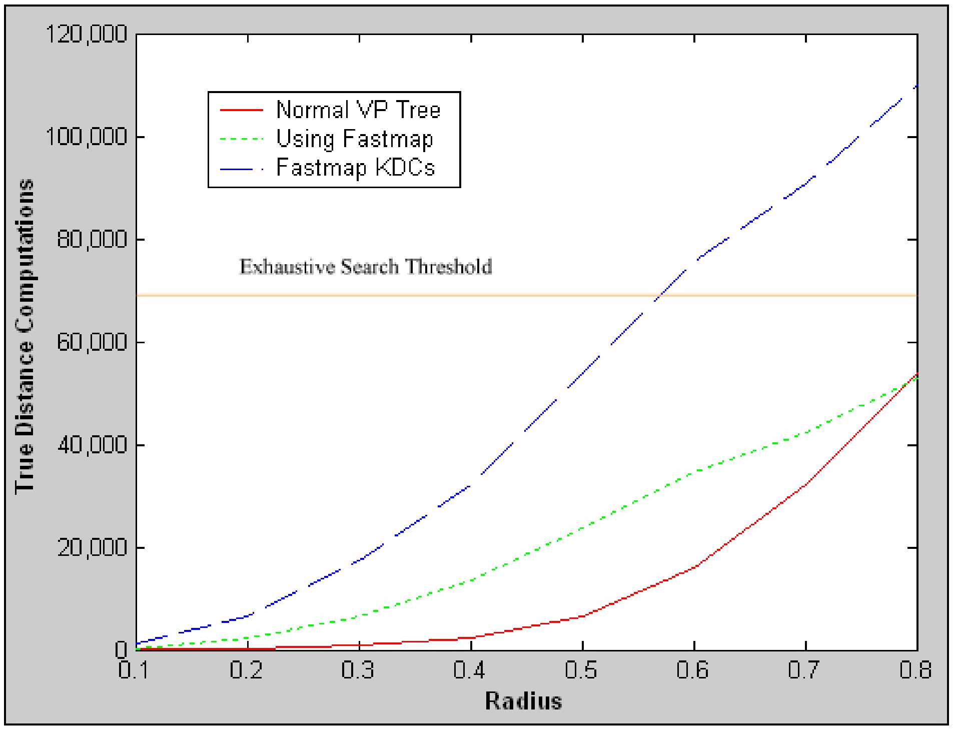


Figure 6.1.8: ColorMoments Dataset (9-D) using random query points.

6.2 Experimental Results for R*-Trees

Our experiments in this section turn to the comparison between a typical R*-Tree and an R*-Tree using FastMap as a filter. We present Figures 6.2.1-6.2.6 to illustrate these differences with our experimental results. The same datasets are used as before from the VP-Tree experiments, and the same aggregate query sets are performed. Once again for the FastMap results, **in all cases K=3, such that k-d space is a 3-Dimensional cube, with the exception of Figure 1.1 where the number of true dimensions is already 3, so we map down to 2 dimensions.**

We see similar results to the Vantage Point Tree experiments, in that the R*-Tree does not perform well for datasets with high dimensionality. In cases where the number of dimensions is 10 or higher, it's apparent that the cost of the R*-Tree range query can be worse than an exhaustive search even with a very small query radius. This is a problem that FastMap appears to improve most of the time. Of course by using FastMap, the very worst case is equivalent to an exhaustive search (since the worst case of FastMap is that you retrieve every object and must perform true distance comparisons with the query object). This is not the case with stand-alone VP-Trees or R*-Trees, as in high dimensions, range queries can be far worse than an exhaustive search, and hence severely counter-productive.

The extent to which FastMap improves the range queries in our R*-Tree experiments is similar to the improvement that we saw in VP-Trees, with one exception. That exception is how FastMap performs in the case of a very dissimilar object being used as the query object in high dimensional space containing clustered data. We recall that in this case, we need a relatively larger radius to find similar objects in the original space, since we assume it to be very large in dimensionality **(10+)** and hence a dissimilar object would be very far from any existing clusters. We still have the problem of this larger radius retrieving more erroneous objects from a FastMap query in k-d space, however this seems to be greatly offset by FastMap's ability to sidestep the necessity of a hypercube query in the original space. We recall that in 20-d space, a hypercube is 40 million times larger than the sphere which it perfectly encloses. In essence, we have the lesser of two evils, in that the necessity of a larger radius will retrieve more objects in k-d space, however that is a far less of a price to pay than retrieving a huge number of erroneous hypercube corner objects in the original space. The corner space constitutes such an overwhelming majority of the volume in high dimensional space that it seems plausible that even the poorest conditions would still yield an improvement by using FastMap.

These improvements by using FastMap even in the sub-optimal cases are illustrated in Figures 6.2.7 and 6.2.8. The drawbacks of the R*-Tree are very apparent here, as we see even the smallest of range queries in 16-space being on the verge of exhaustive, and quickly becoming even worse. It can be seen from figure 6.2.8, which is 9-D clustered data, that FastMap can possibly be detrimental to performance if the dimensionality is not extremely high and dissimilar query objects are used. But the dominant aspects of the data suggests that FastMap is an improvement in almost all cases for R*-Trees in high dimensions, even with dissimilar query objects in tightly-clustered datasets.

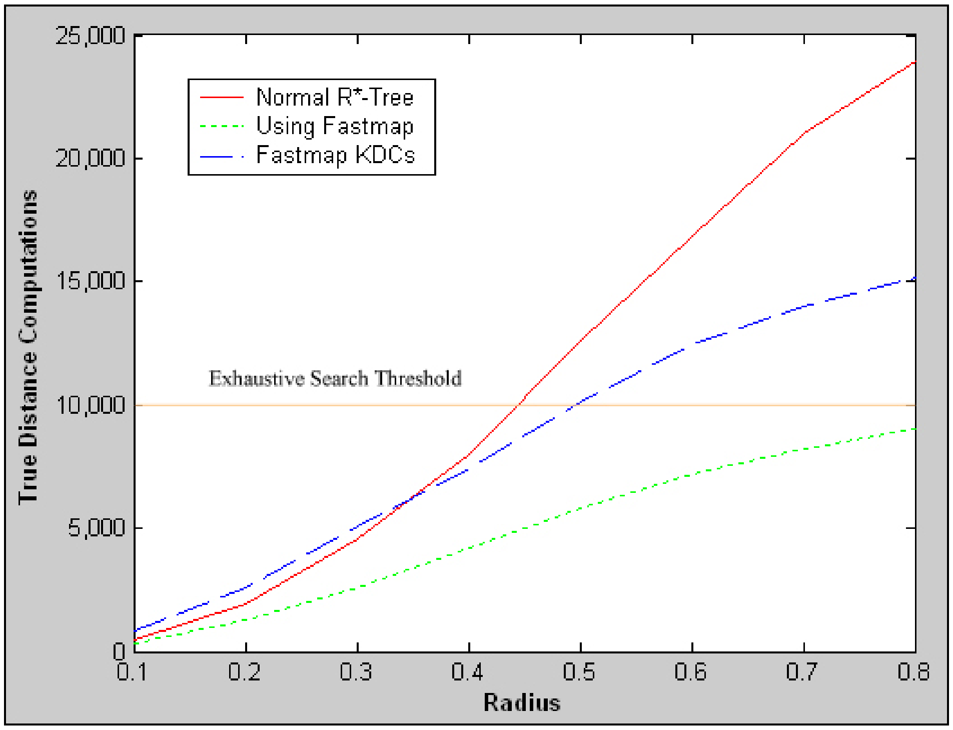


Figure 6.2.1: Uniform Distribution (3-D).

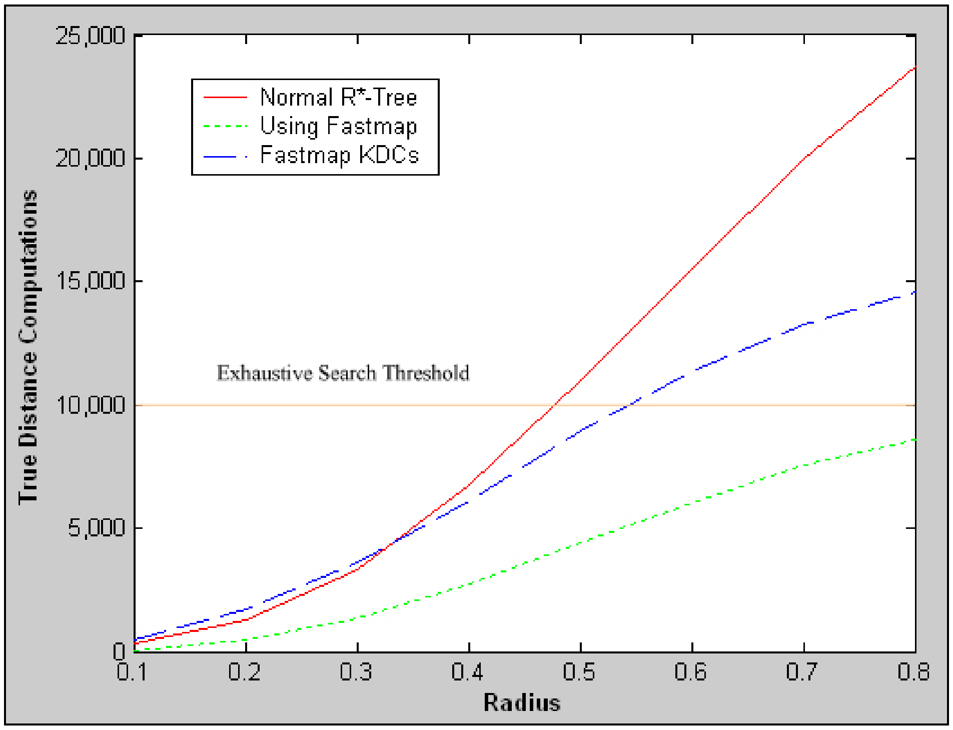


Figure 6.2.2: Uniform Distribution (4-D).

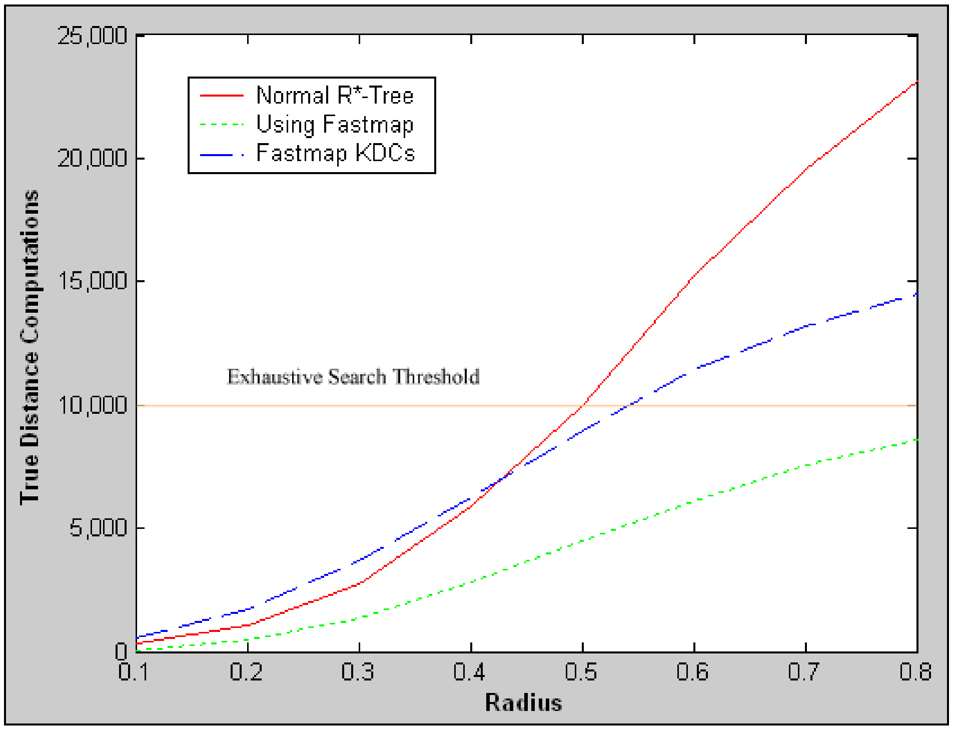


Figure 6.2.3: Uniform Distribution (5-D).

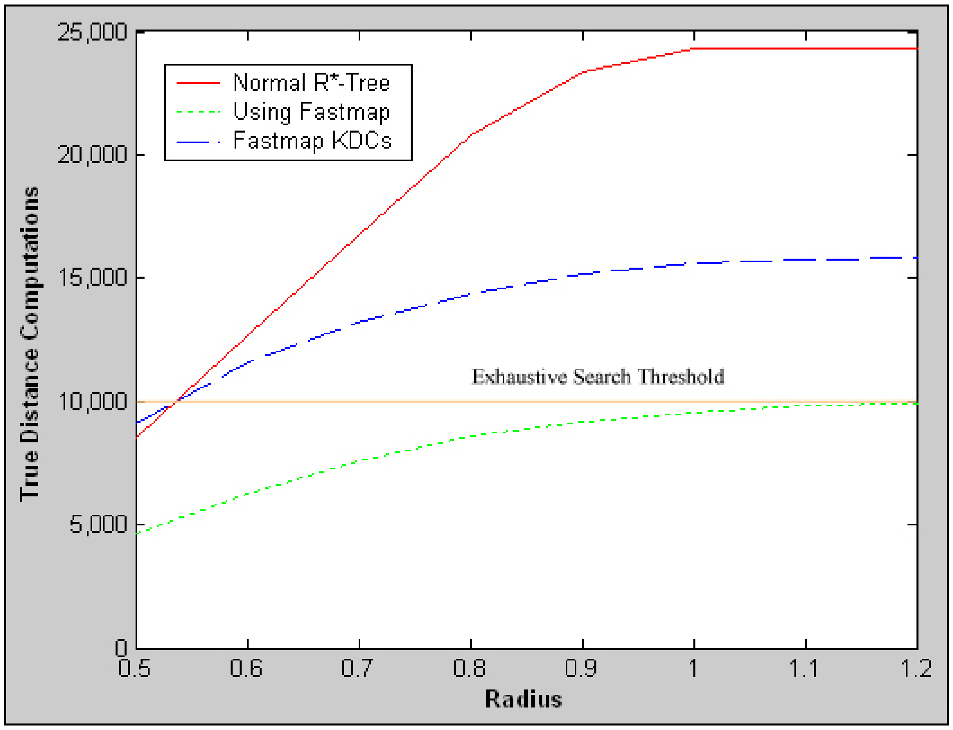


Figure 6.2.4: Uniform Distribution (10-D).

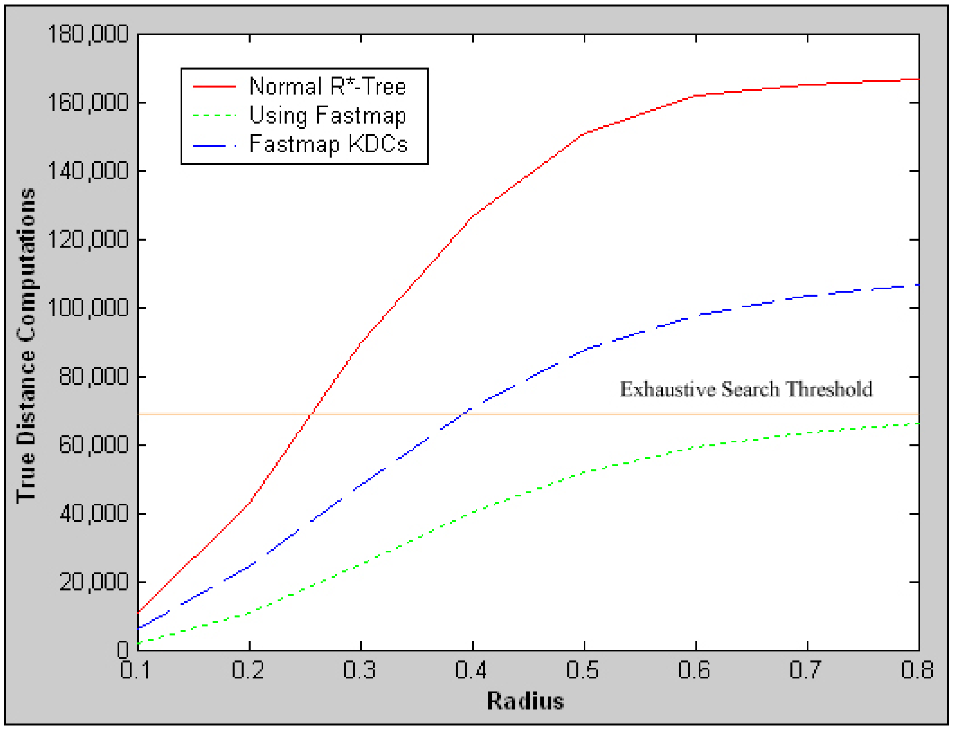


Figure 6.2.5: ColorMoments Dataset (9-D).

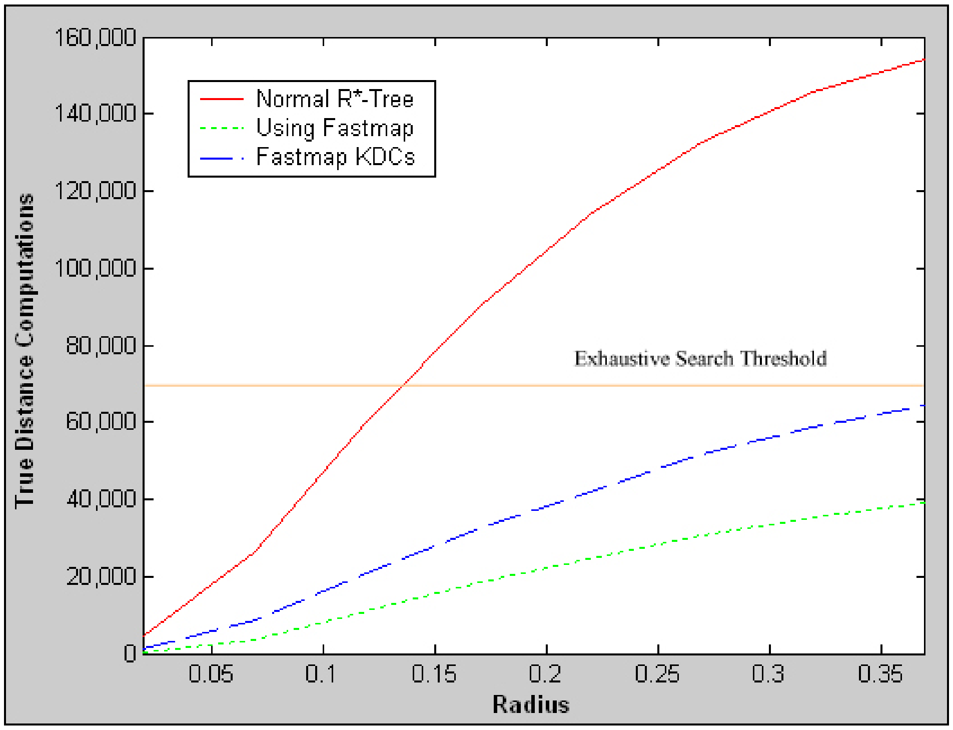


Figure 6.2.6: CO-Texture Dataset (16-D).

Figures 6.2.7 and 6.2.8 illustrate FastMap's performance for high dimensional datasets with significant clusteration and random query points (objects presented that are potentially dissimilar from any other objects in the real data). In Figure 6.2.7 we see the R*-Tree performing so poorly that it immediately becomes exhaustive, and shortly thereafter becomes worst-case R*-Tree performance, illustrated by the plateau effect.

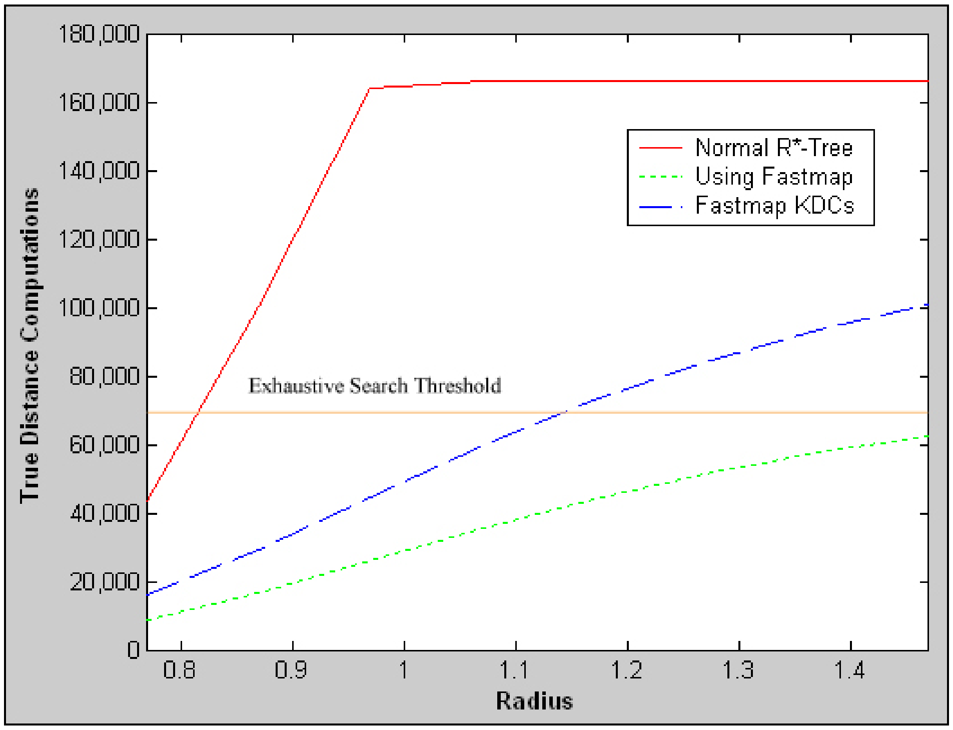


Figure 6.2.7: CO-Texture Dataset (16-D) using random query points.

Figure 6.2.8 demonstrates how the trade off between corner objects in an R*-Tree query and erroneous FastMap objects in k-d space can sometimes balance out if the number of dimensions is not extremely high (less than 10). As it can also be seen, as the query radius increases, FastMap starts to outperform the R*-Tree anyway, since the hypercube in 9-space starts to grow, and this exponential increase in the number of erroneous corner objects begins to severely outweigh the problems with extra object retrieval in k-d space.

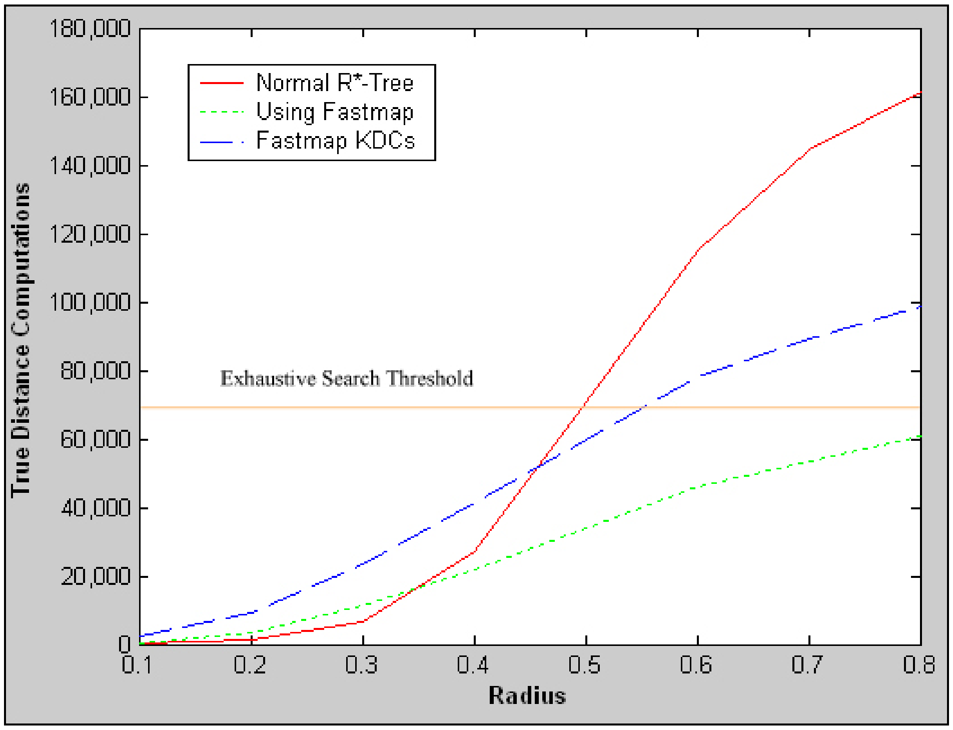


Figure 6.2.8: ColorMoments Dataset (9-D) using random query points.

6.3 Performance Comparisons between VP-Trees and R*-Trees

We present Figures 6.3.1 and 6.3.2 to compare the performance of VP-Trees and R*-Trees with the same range queries on our uniform datasets of size 10,000. A sample radius in each case (0.2 and 0.4) is used, and then tested with different amounts of dimensionality reduction, from 4-10 dimensions in the original set. In all cases, we map down to 3 dimensions. Shown in each figure is the amount of true distance computations performed by the SAMs, the corresponding amount of distance computations needed when using FastMap for those same SAMs and the extra distance computations performed in k-d space by FastMap (KDCs). We chose radii that would be low enough keep the distance computations a fair amount below an exhaustive search, but high enough to keep the number of objects retrieved in the search non-zero.

It is important to note that the radius is fixed. An artifact of this is that in the following visualizations, there are decreasing query cost values as the number of dimensions increases. The reason for this is that the radius is not changing. In 4 dimensions, a radius of 0.4 will retrieve more objects than in 5 dimensions, because the volume of the space is smaller and the objects are less distant and scattered. Since the number of objects retrieved is more, we will inherently have a situation where the cost to retrieve those objects is more. Keeping this in mind is essential to interpreting the holistic nature of the data illustrated in figures 6.3.1 and 6.3.2.

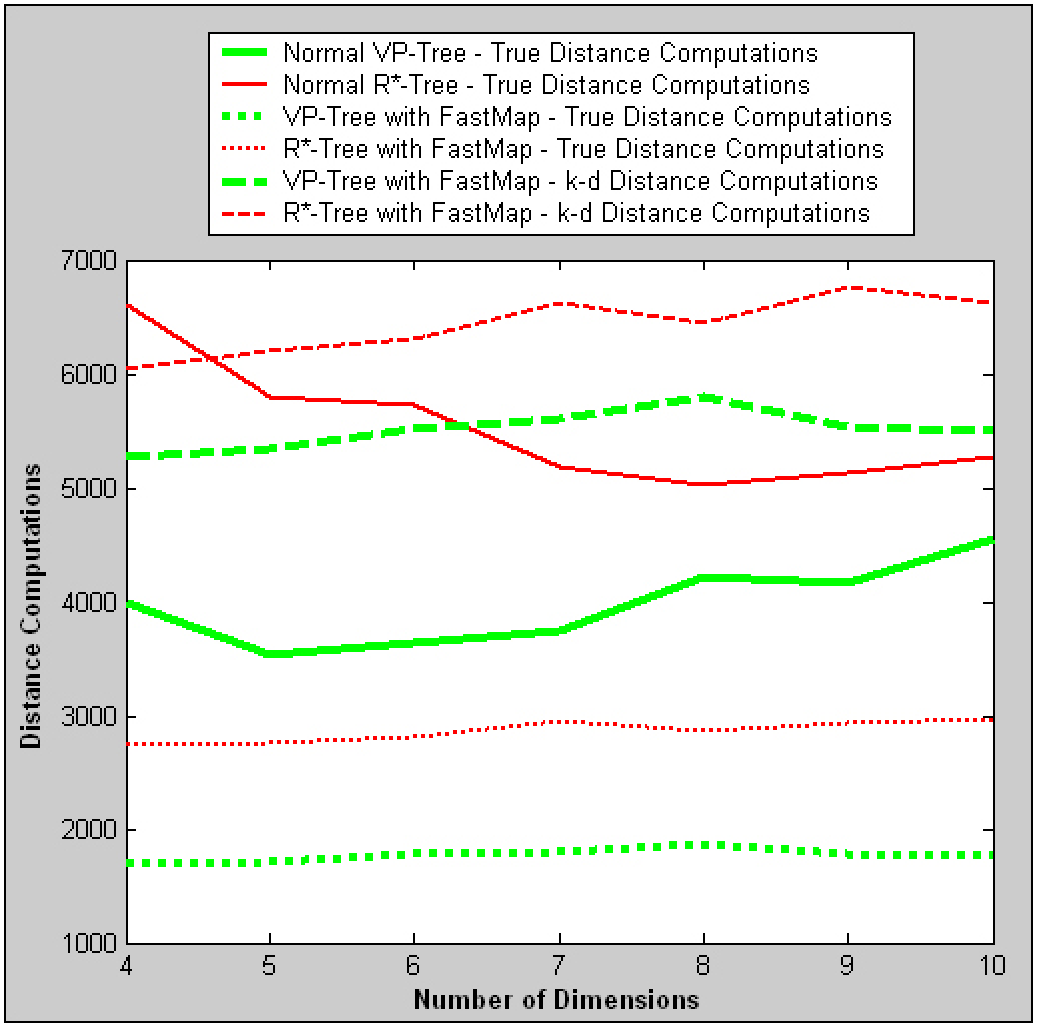


Figure 6.3.1: Uniform Dataset with varying dimensions. Radius = 0.4.

It is apparent from the data that the VP-Trees are, for all our experiments, always performing better than the R*-Tree. At first this may be counter-intuitive since we are using the most primitive form of VP-Trees by always choosing random vantage points, and also because R*-Trees are a much newer construct. Therefore it is important to note that a huge advantage of the R*-Tree over the VP-Tree is that it is a dynamic SAM, as opposed to the static nature of the VP-Tree. Also, R-Trees support for spatial data is one of the goals of its design; the VP-Tree is a bit more specialized for points in vector-space.

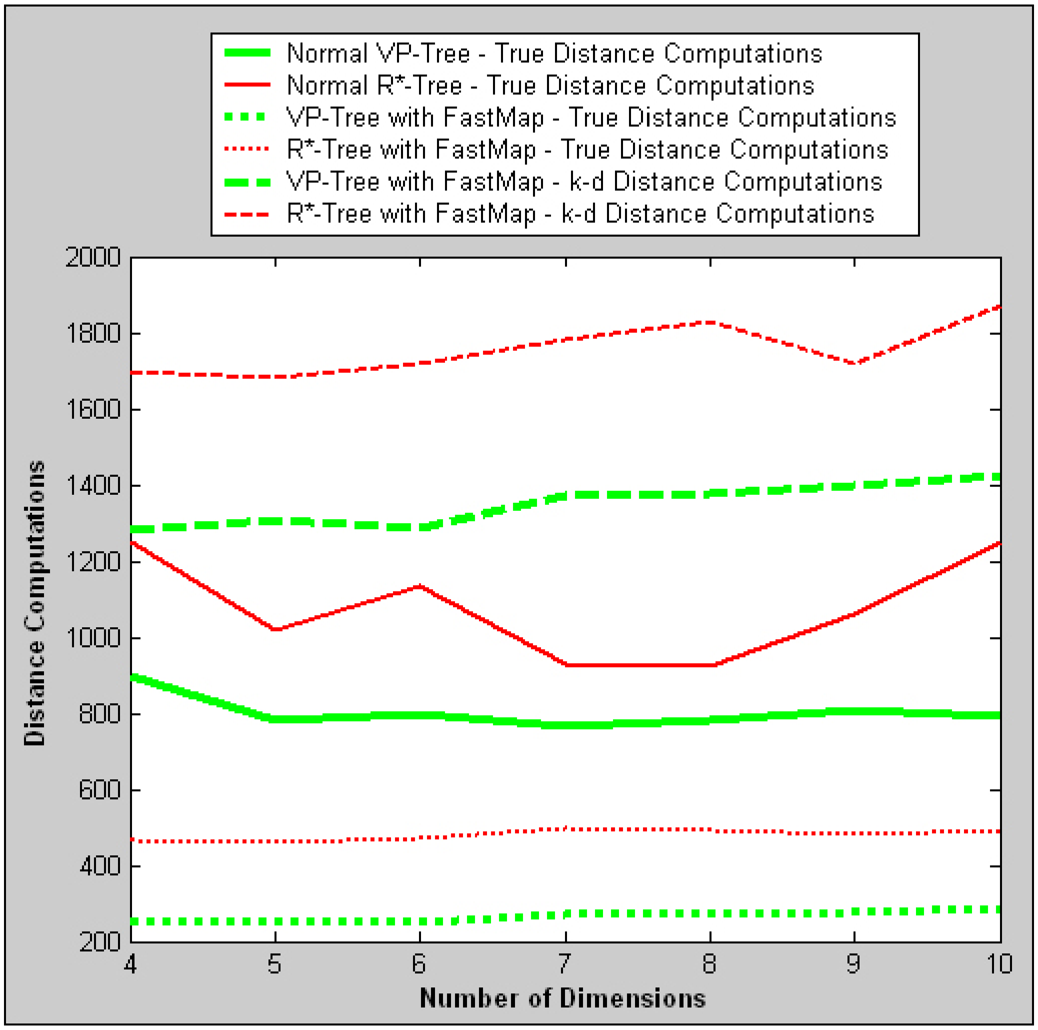


Figure 6.3.2: Uniform Dataset with varying dimensions. Radius = 0.2.

In Figure 6.3.2 our second chosen radius (reduced from 0.4 to 0.2) provides a fairly efficient query by contrast. The number of distance computations never exceeds 2000, where 10,000 is exhaustive. Again, in all cases the VP-Tree performs better and FastMap is an improvement by the criteria of lowest amount of true distance computations. One must keep in mind that FastMap's total cost is the sum of both the true distance computations and the KDCs, although they are not combinable under one measure. It should be kept in mind, then, that although the figures show FastMap's cost as being the least by far, that the KDCs are potentially the most costly by far if K is high. Once again, this illustrates that FastMap is most useful when **d** is large and **K** is small.

It is also useful in these situations to visualize the amount of query objects being retrieved in each case. Figures 6.3.3 and 6.4.4 show the true query results and cube query results for the previous two radii (0.4 and 0.2, respectively). The true sphere query result is important here, because as it declines, so does the proportion of objects retrieved to distance computations required. This is a key aspect of analysis, because a large number of distance computations may be the result of a large query result, not poor SAM performance. Figure 6.3.3 shows that our first radius (0.4) was about the right size to keep the objects retrieved within the realm of 1-700, staying positive yet fairly small.

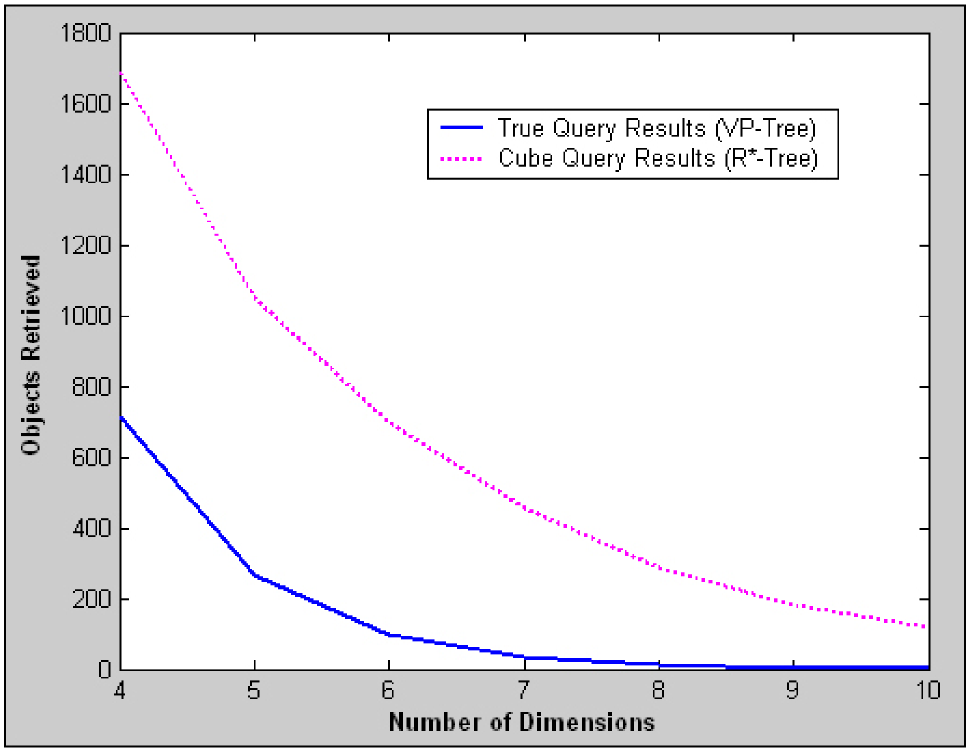


Figure 6.3.3: Number of Objects Retrieved for Radius 0.4.

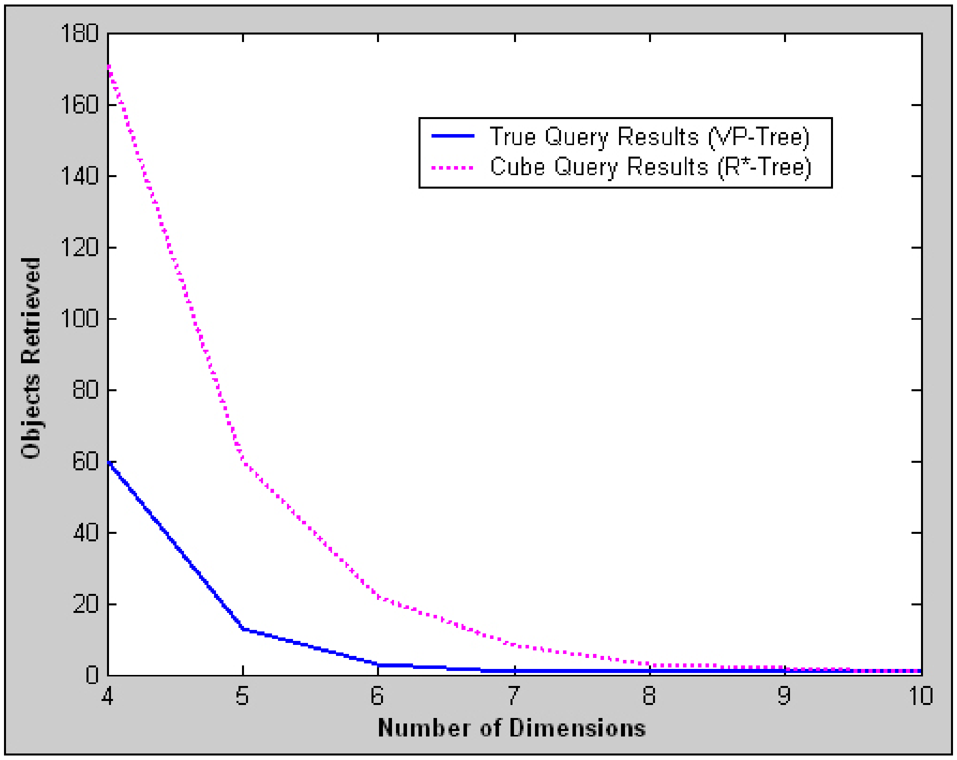


Figure 6.3.4: Number of Objects Retrieved for Radius 0.2.

With a smaller radius (0.2) the size of the true query result set is smaller still, becoming about 1 object for any dimensionality higher than 6-d. For a database of size 10,000 though, retrieving 1-180 objects is fairly realistic. As such, the previous results in Figure 6.3.2 are certainly useful, and are validated in both their comparison of the two access methods as well as their demonstration that FastMap has the potential to increase query efficiency.

7. Conclusions

Our experiments have shown that FastMap is indeed a useful tool for visualization and retrieval in large databases, and can improve the querying efficiency of a variety of point and spatial access methods by acting as an index filter. One large advantage of using FastMap is that any dataset can be mapped as points in arbitrary k-dimensional space given no other information besides a metric distance function. Other techniques would require feature extraction functions for data, which are often difficult to design. An additional convenience of this is that we now have the flexibility to choose any indexing method we want, as opposed to being confined to distance-based indexing in the case of non-dimensional data; our dataset after using FastMap consists of points in vector space, which all access methods can index, regardless of their classification as multidimensional or distance-based.

Utilizing FastMap as a filter for access methods can greatly improve query efficiency, especially when the objects are high in dimensionality or very complex, and the value of **k** is small. Without utilizing a dimensionality reduction technique, conventional PAMs and SAMs quickly become worse than an exhaustive search when the number of dimensions becomes large (10-15). By using FastMap, we guarantee a worst case search performance of **O(N)**. The range query performance in general is greatly improved, moving most of the dissimilarity computations into k-d space instead of costly computations in original space; because of this, queries that were once made exponentially worse because of the curse of dimensionality may now be realistic to perform. These encouraging results hold true depending on the attributes of the dataset and query objects; there does not appear to be any degradation in performance given that query objects are not hugely dissimilar from the objects in the dataset being searched. In addition to this, construction costs become a flat rate on the order of **O(N*K)** regardless of what PAM or SAM we are using to index our data in reduced space. All dissimilarity computations performed by our indexing method during construction are moved into k-d space.

Future work on this topic may be to test the results of other dimensionality reduction techniques in comparison to FastMap, or to expand upon the study of which types of near-neighbor queries do not produce improvements in querying efficiency. In our experiments, we identified one such scenario, in that highly dissimilar query objects in a dataset with high dimensionality will produce too many false matches in FastMap's filtering phase. There may be other scenarios where FastMap performs poorly, and any study to uncover these would be useful. Also, our experiments only utilized two particular access methods; the VP-Tree and R*-Tree; any additional experiments with other search tree structures would be beneficial.

References

[Bel57] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[Bent75] J. L. Bentley. *Multidimensional Binary Search Trees Used for Associative Searching*. Communications of the ACM, Vol. 18, No. 9, pages 509–517, 1975.

[Bent79] J. L. Bentley. *Multidimensional Binary Search in Database Applications*. IEEE Trans. Software Eng. Vol. 4, No. 5, pages 333–340, 1979.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger. *The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles*. ACM SIGMOD, pages 237-246, May 1990.

[BO97] Tolga Bozkaya, Meral Ozsoyoglu. *Distance-Based Indexing for High Dimensional Metric Spaces*. Proc. ACM SIGMOD, pages 357-368, 1997.

[BO99] Tolga Bozkaya, Meral Ozsoyoglu. *Indexing Large Metric Spaces for Similarity Search Queries*. Association for Computing Machinery Transactions on Database System, pages 1-34, 1999.

[Brin95] Sergey Brin. *Near Neighbor Search in Large Metric Spaces*. Proc. 21st VLDB Conference, pages 574-584, 1995.

[CNBM01] Edgar Chavez, Gonzalo Navarro, Ricardo A. Baeza-Yates, Jose L. Marroquin. *Searching in Metric Spaces*. Proc. ACM Computing Surveys, Vol. 33, No. 3, pages 273-321, 2001.

[Com79] D. Comer, "*The Ubiquitous B-tree*," Computing Surveys, Vol. 11, No. 2, pages 122-137, June 1979.

[CPZ97] Paolo Ciaccia, Marco Patella, Pavel Zezula. *M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces*. Proc. 23rd VLDB Conference, pages 426-345, Athens, Greece, 1997.

[DH01] Richard Duda, Peter Hart. *Pattern Classification, Second Edition*. John Wiley & Sons, New York, NY, 2001.

[Fal95] Christos Faloutsos, King-Ip (David) Lin. *FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets*. Proc. ACM SIGMOD Vol. 24, No.2, pages 163-174, June 1995.

[FGA83] H. Fuchs, E. D. Grant, G. D. Abram. *Near Real-Time Shaded Display of Rigid Objects*. Computer Graph. Vol. 17, No. 3, pages 65–72, 1983.

[FKN80] H. Fuchs, Z. Kedem, B. Naylor. *On Visible Surface Generation by A Priori Tree Structures*. Computer Graph. Vol. 14, No. 3, 1980.

[Gae95] Volker Gaede. *Optimal Redundancy in Spatial Database Systems*. Proc. 4th International Symposium on Spatial Databases, pages 96-116, Portland, Maine, 1995.

[GG98] Volker Gaede, Oliver Gunther. *Multidimensional Access Methods*. ACM Computing Surveys, Vol. 30, No. 2, pages 123-169, June 1998.

[Gutt84] Antonin Guttman. *R-Trees: A Dynamic Index Structure for Spatial Searching*. Proc. ACM SIGMOD, pages 47-57, June 1984.

[HAK00] Alexander Hinneburg, Charu Aggarwal, Daniel Keim. *What Is the Nearest Neighbor in High Dimensional Spaces?* Proc. 26th VLDB Conference, pages 506-515, Cairo, Egypt, 2000.

[HS03] Gisli R. Hjaltason, Hanan Samet. *Index-Driven Similarity Search in Metric Spaces*. Proc. ACM Transactions on Database Systems, Vol. 28, No. 4, pages 517–580, December 2003.

[KM83] I. Kalantari, G. McDonald. *A Data Structure and an Algorithm for the Nearest Point Problem*. IEEE Trans. Software Eng. Vol. 9, No. 5, pages 631–634, September 1983.

[NVZ92] H. Noltemeier, K. Verbarg, C. Zirkelbach. *Monotonous Bisector Trees--A Tool for Efficient Partitioning of Complex Scenes of Geometric Objects*. In Data Structures and Efficient Algorithms, pages 186–211, Berlin, Germany, 1992.

[OM84] J. Orenstein, T.H. Merrett. *A Class of Data Structures for Associative Searching*. Proc. 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, pages 181-190, 1984.

[Sam84] H. Samet. *The Quadtree and Related Hierarchical Data Structure*. ACM Computing Survey, Vol. 16, No. 2, pages 187–260, 1984.

[SRF87] Timos Sellis, Nick Roussopoulos, Christos Faloutsos. *The R+-Tree: A Dynamic Index for Multi-Dimensional Objects*. Proc. 13th VLDB Conference, pages 507-518, Brighton, England, September 1987.

[STMO03] S. Cenk Sahinalp, Murat Tasan, Jai Macker, Z. Meral Ozsoyoglu. *Distance-Based Indexing for String Proximity Search*. In *IEEE Data Engineering Conference, 2003*.

[Uhl91] J. K. Uhlmann. *Satisfying General Proximity/Similarity Queries with Metric Trees*. Inf. Process. Lett. Vol. 40, No. 4, pages 175–179, November 1991.

[Yian93] Peter Yianilos. *Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces*. Proc. ACM-SIAM SODA, pages 311-321, 1993.

[ZWYY03] Xiangmin Zhou, Guoren Wang, Jeffrey Xu Yu, Ge Yu. *M+-Tree: A New Dynamical Multidimensional Index for Metric Spaces*. Proc. 14th Australasian Database Conference, pages 161-168, Adelaide, Australia, 2003.