ABSTRACT

Title of dissertation: NEW APPROACHES
TO SIMILARITY SEARCHING
IN METRIC SPACES

Cengiz Celik, Doctor of Philosophy, 2006

Dissertation directed by: Professor David Mount
Department of Computer Science

The complex and unstructured nature of many types of data, such as multimedia objects, text documents, protein sequences, requires the use of *similarity search* techniques for retrieval of information from databases. One popular approach for similarity searching is mapping database objects into feature vectors, which introduces an undesirable element of indirection into the process. A more direct approach is to define a distance function directly between objects. Typically such a function is taken from a *metric space*, which satisfies a number of properties, such as the triangle inequality. Index structures that can work for metric spaces have been shown to provide satisfactory performance, and were reported to outperform vector-based counterparts in many applications. Metric spaces also provide a more general framework, and for some domains defining a distance between objects can be accomplished more intuitively than mapping objects to feature vectors.

In this thesis we will investigate new efficient methods for similarity searching in metric spaces. We will first show that current solutions to indexing in metric

spaces have several drawbacks. Tree-based solutions do not provide the best trade-offs between construction time and query performance. Tree structures are also difficult to make dynamic without further degrading their performance. There is also a family of flat structures that address some of the deficiencies of tree-based indices, but they introduce their own unique problems in terms of higher construction cost, higher space usage, and extra CPU overhead.

In this thesis a new family of flat structures will be introduced, which are very flexible and simple. We will show that dynamic operations can easily be performed, and that they can be customized to work under different performance requirements. They also address many of the general drawbacks of flat structures as outlined above.

A new framework, *composite metrics* will also be introduced, which provides a more flexible similarity searching process by allowing several metrics to be combined in one search structure. Two indexing structures will be introduced that can handle similarity queries in this setting, and it will be shown that they provide competitive query performance with respect to data structures for standard metrics.

NEW APPROACHES TO SIMILARITY SEARCHING IN METRIC SPACES

by

Cengiz Celik

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

Advisory Commmittee:

Dr. David Mount, Chair/Advisor
Dr. Lise Getoor
Dr. Hanan Samet
Dr. Yavuz A. Oruc
Dr. Amitabh Varshney

© Copyright by
Cengiz Celik
2006

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

Computers excel at storing vast amounts of information. Many applications in computer science depend on efficient storage and retrieval of data. As the cardinality of data sets increases, it is important to be able to retrieve data based on certain criteria. One important example of this is *similarity search*, where a query object is given, and similar objects are retrieved by the system. Two important queries used in similarity matching are *range queries* and *k-nearest neighbor queries*. A *range query* is given a query object q and a number r, and returns all the objects that are within distance r of q. A *k-nearest neighbor query* returns the k closest objects to a given query object, for some given positive integer k. A few applications of the similarity search include: audio and image databases [20], video, text files, fingerprints [31], face recognition [30], and protein sequences [15].

Although they support essentially the same interface, similarity search systems differ dramatically in how they model objects and how distances are defined. In particular, most of the research to date has focused on objects represented as coordinate vectors. In these systems, it is assumed that the objects can be decomposed into or represented as vectors over some multi-dimensional space, and distances are measured using geometric distance functions like standard Euclidean distance. A large number of index structures have been created based on this framework. The

coordinate vector approach is too restrictive for many classes of objects. An alternative direction for research had been similarity search in the more general setting of *metric spaces*. A metric space is defined to be a set of objects X together with a distance function d on pairs of objects that satisfies the following properties for all $a, b \in X$

- Positivity: $d(a, b) \geq 0$.
- Symmetry: $d(a, b) = d(b, a)$.
- Triangle Inequality: $d(a, b) + d(b, c) \geq d(a, c)$.

The metric space model has the capability of capturing a large variety of similarity search applications. In many cases, it is the most natural manner in which to approach a problem. For example, the distance between two character strings may easily be determined by the *edit distance*, which is a metric [28].

Metric-based methods have the simplicity of not needing to make any assumptions about the internal structure of the objects. All they require is a distance function that can be invoked on any pair of objects. This high level of abstraction, however, limits the flexibility of index structures. For example, vector-based methods can enhance efficiency by processing the dimensions of the vector one at a time. An example of this is incremental distance computation [4], where the distance of the query object to a bounding box is computed one dimension at a time. Another example is the TV-tree [29]. In the TV-tree, new dimensions are introduced only as they are needed.

The triangle inequality dictates that the distance between two objects is closely related to their distances to a third object. Metric-space indexing structures exploit this fact by appointing a small set of objects to represent the whole population. These objects are called *pivots* or *vantage points*. The distances between the pivots and a set of database objects are pre-computed and stored in the index structure. At query time, the distance between some of the pivots and the query object is computed. Using the triangle inequality, the distance between a regular database object and the query object can be bounded by their distances to the pivots. If the lower bound of the distance between a database object and the query object is greater than the query radius, it follows that the object is outside the query range, and the object can be eliminated from consideration. In a similar fashion, if the upper bound of the distance is less than the query radius, it follows that the object lies within the range. We call this operation *pivoting*. Objects that have been classified in this manner are said to be *eliminated*. Database objects that are not eliminated must have their distances to the query object computed explicitly. The efficiency of an index structure is directly related to the fraction of database objects that can be eliminated through pivoting.

Almost all existing index structures for metric similarity search are built around the concept of pivoting. They differ in the way they select pivots, decide which objects should be represented by each pivot, and how the pivot distances will be organized. These differences also affect how the querying process will be carried out.

In this thesis we will focus on cases where the distance computation is relatively

expensive and dominates the overall cost of a query. Although we will also address other issues like space and additional query time overhead, we will use the number of *distance computations* as our primary measure of the cost of answering a query. We will use the terms *CPU overhead* and *computational overhead* when referring to other organizational and bookkeeping operations performed by the index.

Throughout the thesis will focus primarily on distributions that we characterize as being difficult. Informally, a distribution is *difficult* if it does not allow a pivot to eliminate many objects. This occurs when distances tend to be concentrated around a small number of values. When this happens a large number of objects of the database are at roughly the same distance to a typical pivot. As a result the process of elimination through pivoting is least efficient. The phenomenon of distance concentration is known to occur for uniformly distributed point sets in vector spaces of high dimension. Although there is not a widely agreed upon definition of dimension for metric spaces, we will sometimes refer to such difficult distributions informally as being "high-dimensional". We also characterize a query as being *difficult* if many database objects are at a distance from the query object that is close to the range's radius. Again, this definition is operational because it represents a situation where pivoting will be least efficient in eliminating objects.

Before discussing the contributions of this thesis, let us briefly review some of the existing work in this area. (A more detailed review can be found in Chapter 2.) Much of the previous work in similarity search in metric spaces has focused on structures residing in main memory. They usually depend on a hierarchical organization of the pivots and objects. Among these, the *vp-tree* [42] stands out

as one requiring only a small amount of memory and being able to be constructed efficiently. However it is inferior to others in terms of query performance, including methods like the *mvp-tree* [8], and *GNAT* [9], both of which improve performance at the cost of greater space and construction time.

The *M-tree* [17] and *Slim-tree* [23] are disk-based structures, and support dynamic manipulations on the index while maintaining the balance of the tree. In order to be able to efficiently handle split and merge operations, however, they keep less precise data than the comparable GNAT structure. This results in poorer query performance.

Tree structures typically only allow an object to have as many pivots as the height of the tree. This may not be satisfactory for difficult distributions and queries. For this reason tree-based structures are not flexible enough to provide greater elimination power when needed. In contrast, vantage point structures like *LAESA* [34], *Spaghettis* [12] and *FQA* [13] represent another family of solutions. They use more space and construction time, but provide greater efficiency at query time. Although other tree structures also have some parameters that can be adjusted, their improvements are not as pervasive or as dramatic. The shortcomings of vantage points-based methods are the extra computational overhead that they incur, higher construction costs, and higher space usage. If they are allowed to use a sufficient number of pivots, these methods have been shown to outperform other methods in terms of the number of distance computations performed. Some of the structures in this family offer some improvements to the common problems of high space and construction time. The Spaghettis structure reduces computational overhead but

uses more space than the common approach. The FQA also reduces overhead, but it uses less precision in the distance information it stores, resulting in reduced performance in terms of the number of distance computations.

In this thesis, we study a number of practical improvements to data structures and algorithms for similarity searching in metric spaces. Because of our interest in practical performance, our analysis will be primarily empirical in nature. A recurring theme in our methods will be the observation that some distance information is more important than others. More specifically, we will show that a pivot is most effective for objects that are either very close or distant to it.

We will first introduce a method called *prioritized vantage points*, which reduces the computational overhead of vantage point-based methods. The basic idea is to only process pivots that are either close to or far from the query object. Unlike similar solutions proposed before, this method does not compromise on the quality of the distance information. (This is one of the principal weakness of the FQA structure). Also, our method does not introduce extra space overhead (as does the Spaghettis structure).

In the following chapters we will introduce a series of index structures that provide successive improvements to each other, in the sense that each structure will accumulate all the benefits of the previous ones and make additional enhancements. The first is the *Kvp* structure. This structure is unique since it improves both the storage and computational overhead of the classical vantage-points approach. The Kvp structure offers a number of benefits:

- It is a simple data structure and can be implemented relatively easily.
- It can support dynamic operations like insertion and deletion.
- It is easily adapted for use as a disk-based structure and its access patterns minimize the number of disk-seek operations.
- Queries may be executed in parallel.

We will show that tree-based structures like the vp-tree may offer suboptimal performance, even when the given distribution and query are not particularly difficult (for example, when the distribution is of low dimension and when the query radius is small). Vantage point structures suffer even more in this respect, since the number of pivots is not bounded by the height of the tree. We introduce a new structure, called *HKvp*, which overcomes these shortcomings since it does not necessarily compute the distance of the query object to all the pivots, that is, it allows the elimination of less promising ones. Even though this feature was implemented by some of the earliest structures, they lacked the capability of limiting space and computational overheads. We will show that HKvp works more efficiently than its counterparts, while offering all the benefits of the Kvp structure.

Next we introduce a structure, called *EcKvp*, which addresses one of the major weaknesses of global pivot-based structures, namely construction complexity. Since the pivots are global, we need to compute their distances to all the objects in the database. Most of this distance information is relatively useless. On the other hand, tree structures attempt to cluster relevant objects together, and therefore a selected pivot will be representing similar objects. This will help the pivot compute its

distance against these similar objects. The EcKvp structure also avoids computing distances to all of the objects. We will show that it does a better job of finding similar objects to a given pivot. This allows EcKvp to be constructed more efficiently, but with a negligible performance penalty.

Finally, we propose a new framework for similarity searching in metrics spaces, called *composite metrics*. A composite metric is a way of incorporating several metrics into the process of distance computation. We present examples that motivate this concept in two different ways: when there are alternative definitions of distance, and when the distance is dependent on several unrelated criteria. The queries in this framework can specify relative weights for each of these metrics, which makes querying more flexible. We will introduce two new index structures that are specifically designed for composite metrics: the *c-tree* and the *c-forest*. We will show that they perform very competitively compared to similar structures that work on standard metric spaces. The c-tree uses only one of the metrics per node, and incorporates others at different levels of the tree. The c-forest is a composition of c-tree structures. It operates by executing the query independently in each of the c-trees. These c-tree structures are used for eliminating objects that do not lie within the query range. The c-forest only computes the distances to the objects that were not eliminated by any of the c-tree structures. We will present experiments that show that c-trees and c-forests provide an efficient solution to similarity searches over composite metrics.

The organization of the rest of this thesis is as follows. In Chapter 2, we will present a brief survey of the existing work, followed by a discussion of general issues

in metric space indexing in Chapter 3. In Chapters 4, 5, and 6 we will present the Kvp, HKvp and EcKvp structures, respectively. In Chapter 7 we will introduce the composite metrics framework along with the c-tree and the c-forest structures. Finally, in Chapter 8 we will offer concluding remarks.

Chapter 2

Related Work

There has been considerable work on similarity searching in metric spaces from different disciplines, sometimes unaware of each other. Two very good surveys can be found at [15] and [21].

For index structures that reside in main memory, query performance is based on two principal components: the time spent on computing distances between objects and any additional time needed to process, evaluate, and combine the results of these distance computations. Henceforth, we will refer to the latter as *CPU overhead* or *computational overhead*.

We will classify approaches into three broad categories: Clustering-based methods, local pivot-based methods, and vantage points-based methods.

2.1 Clustering-Based Methods

The basic theme behind clustering-based methods is the use of a hierarchical, tree-based decomposition of the space, where the subtrees are designed to group close objects together. We also observe that each subtree is represented by a single object from the database that is ideally located near the center of the group of objects stored in this subtree.

J. Uhlmann [42] defined the *gh-tree*, short for *generalized hyperplane tree*, as

one of the first examples in this category. The idea is to pick two objects from the current subset as representatives, and partition the rest of the set into two classes, depending on which representative is closer.

The *GNAT* tree, presented by S. Brin [9] is a generalization of the gh-tree, where there are more than two representatives. A simple algorithm is given to pick the representatives. According to the algorithm, if we are to select k representatives, we first pick $3 \cdot k$ points randomly. Then, starting with an initial set consisting of one random representative, we incrementally grow the set by adding the point that maximizes the minimum distance to the other representatives.

In addition to its representatives, each node can also maintain the radius of the associated region, that is, the maximum distance of the objects inside a representative's region. This method was used in the M-tree (described below). Another enhancement would be to include the distances between the representatives as well. An even more precise way is used in the GNAT tree. Every representative stores the minimum and maximum distances to the objects in every other subset.

The performance of GNAT was compared to another structure, the *vp-tree*, which will be introduced later. In the best case, the GNAT has been reported to make more than a factor of 6 fewer distance computations than the vp-tree, while requiring about a factor of 14 more in distance computations in its construction. However, it was reported to be worse than the vp-tree in some cases. The original study [9] also showed that GNAT was outperformed by a variant of the vantage points structure, although no data was given about the parameters used in the construction of this structure. Recent experiments presented by [15, 13] show indeed

that GNAT performs consistently worse than variants of vantage points structures in terms of number of distance computations, while it consumes less space and has less computational overhead.

The *M-tree* [17] is designed to be a dynamic structure, with emphasis being paid to the structure's ability to perform queries efficiently and to optimize I/O performance after a sequence of data insertions. Similar to *SS-tree* [46], it keeps the distance to the farthest object in a sub-tree. Maintaining the radius of the representative objects allows it to easily reorganize disk blocks. Splitting a node involves selecting two new representatives and redistributing objects associated with this node among these two new nodes. The M-tree considers all possibilities for a split and chooses the one with tightest covering radius.

The *Slim-tree* [23] employs a more efficient splitting method. The minimum spanning tree of the objects is generated and the longest arcs of the spanning tree is removed partition the set of objects into two subsets.

2.2 Local Pivot-Based Methods

The structures in this category are also tree-based, however, the partitions are based on the distances of objects to either one or two selected objects called *pivots*. Objects that have similar distances to the pivots are put inside the same subtree, but that does not necessarily mean they are in close proximity of each other. The pivots are only used within their subset, and this is why we call them *local pivots*. W. Burkhard and R. Keller [10] suggested selecting a random object in the data

set and partitioning the rest such that every object having the same distance to the pre-selected object is placed in the same subset. The tree construction continues recursively on the subset of points at the same distance. Since their application domain produced discrete distance values, it was possible for many points to be at the same distance.

An adaptation of the same basic idea to continuous distance values is the vp-tree, which was introduced by J. Uhlmann [42]. Such a tree is defined by a branching factor. In order to construct a vp-tree with a given branching factor k, at a given node, one of the objects is selected as the vantage point, and the distances from the other objects to this vantage point are calculated. Then these objects are partitioned into k groups of roughly equal size based on these distances. In this way a node can have k branches with each subtree having roughly m/k objects, where m denotes the number of objects for that node. The only information that needs to be stored is the vantage point itself, and the $k-1$ distance values, denoted as $\mathit{cutoff}[1..k]$, defining the ranges of distances for each subtree.

A range query of radius r centered at a query point q is answered as follows: at any given node, the distance d between q and the node's vantage point is calculated. If d is smaller than r, the vantage point is added into the result set. For every subset i of the node defined by the cutoff values, if the interval of the subset, $[\mathit{cutoff}[i-1], \mathit{cutoff}[i]]$ intersects the interval $[d-r, d+r]$, then subset i is searched recursively.

A nice feature of the vp-tree is that it is possible to divide the space into many divisions through a single distance calculation. As a result, when doing a search, we

need only perform one distance calculation per node. However, as the dimensionality of the data distribution grows, it is well known that for many distributions, the objects tend to cluster around a single distance value [7]. As a result, almost all of the objects are at the same distance to the vantage point. Thus, the distance to the vantage point loses its discriminating power with respect to the objects. Another common way to describe the situation is to visualize the situation in a 3 dimensional space, where the median spheres dividing the branches have very similar radii, subdividing space into thin spherical shells. As a result, objects that are grouped under same subtree tend to be spread around the space rather being close to one another.

The *mvp-tree* [8] uses two vantage points per node. After partitioning the points with one primary vantage point, the partitions are further divided by using the second vantage point. This way, if we divide the space into m different regions by first vantage point, we will have a total of up to m^2 subsets. It should be noted that the second vantage point uses different cutoff values for each partition of the first vantage point. This allows the tree to maintain balance by assigning approximately the same number of points to each subset. This occurs at the cost of more space consumption per node. The value of this partitioning approach is that, instead of dividing the space into very thin shells, it strives to produce more tightly clustered subsets, while still achieving the same fanout. Figure 2.1 illustrates the contrast between the space decompositions of the vp-tree and Mvp-tree.

The mvp-tree stores distances to two vantage points at the leaf nodes, making it a hybrid of the vantage-point structures. It is reported to perform up to 80% fewer

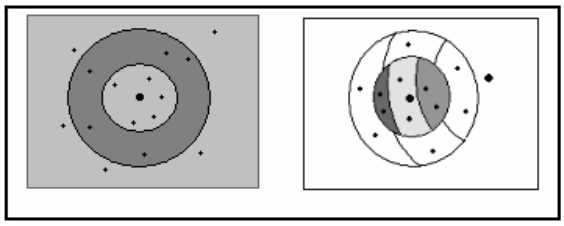


Figure 2.1: Space decomposition by the vp-tree (left) and the mvp-tree (right)

distance calculations compared to the vp-tree. Experiments in [15, 13] show that the non-hybrid version of Mvp-tree is consistently outperformed by vantage-point variants in terms of the number of distance computations.

2.3 Vantage-Point Methods

In vantage-point methods the pivots are used to control processing for the entire set of objects instead of having local scope as they do in the previously described methods. A subset of the objects are selected as vantage points. The distance between the pivots and the rest of the objects are computed at initialization time and stored in the database. At query time these precomputed distances are used to eliminate candidates in a way that is similar to local methods. If there are k vantage points, then the basic method performs $k \cdot n$ distance computations at construction and keeps $k \cdot n$ distance values in the index structure. A range query accesses these distance values to determine which objects can be eliminated based on their distances to vantage points. Finally, a pass through all objects not eliminated by use of pivots is performed.

Note that vantage-point methods require extra processing compared to local

methods, where determination of the partition at a node is done only for the objects covered by the node. Local methods require storage that is only linear in the database size, whereas vantage-point methods require $O(k \cdot n)$ storage.

A powerful aspect of these methods is that it is possible to use as many pivots as desired at the cost of construction time, which results in higher storage requirements and extra preprocessing time. Nonetheless, this additional effort and space can yield progressively better query performance in terms of the number of distance computations.

To the best of our knowledge, the first vantage-point structure that appeared in literature was *LAESA* [34], as a special case of *AESA* [43]. There have been some improvements over the basic LAESA algorithm, such as keeping distances to the vantage points sorted and doing binary searches to identify which objects can be eliminated from consideration [36].

The *TLAESA* structure [33] was proposed as a hybrid method between the LAESA and the gh-tree. The pivots are organized as in a gh-tree, but a distance matrix is also used to provide lower bounds for the distance of the query object to the node representatives. Their experiments were performed in low dimensions, and although were superior to LAESA in terms of total CPU cost, it was inferior in terms of the number of distance computations.

The *Spaghettis* structure [12] was introduced as a method designed to further decrease computational overhead. Here the distances are sorted in a similar fashion. In addition, every distance has a pointer for the same object's distance in the next array of distances. As done in the case of sorted distances, the feasible ranges are

computed for each array using binary search. For each point, its path starting from first array is traced using the pointers. Once the object falls out of range in any of the arrays, we may infer that the object cannot lie within the query region.

The *Fixed Queries Array* (FQA) [13] is one of the recent global pivot-based methods. It sorts the points according to their distances to the first vantage point, then on the second, and so on. It decreases the precision with which distances are measured, for otherwise the points effectively would be sorted only in their distance to the first pivot. Using this sorted structure, the query algorithm performs binary searches within each distance range. The first pivot is processed as in the sorted-array approach, after that, for each range of objects that has the same discretized distance to the first vantage point, we perform a binary search to find the range that is valid for the second pivot. The search continues in this fashion performing binary searches within ranges.

FQA is unique among vantage-point methods that are designed to reduce computational overhead in that it does not require any additional storage. However it does not work very well if too many bits are used for the distance values, since this would require that the structure be sorted only by the first pivot. This creates an additional trade-off between the number of bits used for distance storage and extra CPU processing time needed. This comes in addition to the trade-off between number of bits and query performance in terms of distance computations. Their experiments show great improvements in low dimensions, but for 20 dimensional data for a database of one million objects, they estimate FQA would take only 37.6% less time than a naive approach.

Chapter 3

Issues in Metric Space Indexing

In this chapter, we will mention about some of the factors that influence indexing in metric spaces. Specifically, we will analyze the pivoting operation in more detail, list some of the criteria we may use to evaluate different indexing methods, briefly summarize some of the previous work to model query cost, and present the datasets we will use in later chapters.

3.1 Pivoting Operation

Recall that metric space indexing methods commonly use the pivoting operation to eliminate objects without having to compute their real distance to the query object. Given a pivot object p and any database object o, where the distance between p and o, d_{po}, is pre-computed, we first compute the distance between p and the query object q. Based on this distance d_{pq}, we can deduce that the distance between q and o has to be in the range: $[|d_{po} - d_{pq}|, d_{po} + d_{pq}]$. Figure 3.1 depicts this situation in a 2-dimensional setting.

Sometimes the pivoting operation is performed on a subset of objects. In this case, the pivot usually maintains the minimum and maximum distances to the subset. This approach is used, for example, in the vp-tree and the GNAT structures. Alternatively, just the maximum of the distance is stored, as done in the M-tree and

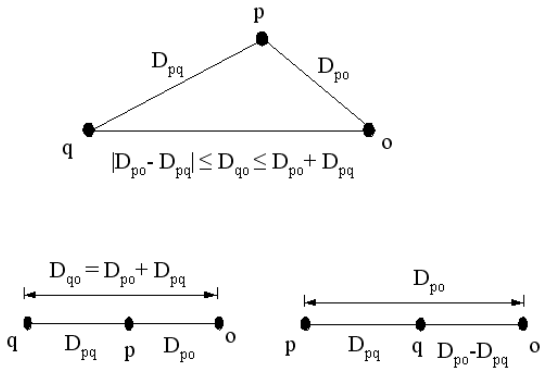


Figure 3.1: A general case and two extreme examples of distance relations between 3 objects.

the slim-tree. Of course, unless the subset of objects is very tightly clustered, upper and lower bounds for a subset will be weaker than bounds computed for an individual object.

For a given database object, there will be a number of pivots that can be processed. The overall lower bound of the distance between the object and the query object is the maximum of the lower bounds acquired from all the pivots. Similarly, the overall upper bound will be the minimum of upper bounds acquired from all the pivots. In order to avoid computing the actual distance of an object to the query object, either the object's overall lower bound should be greater than the query radius, or the upper bound should be less than the query radius. In other words, for a given pivot p, we want the quantity $|d_{po} - d_{pq}|$ to be as large as possible. Recall that for difficult distributions and high-dimensional spaces, distances tend to cluster, and so objects tend to be at about the same distance from each other. Assuming that the query object follows this expected pattern, in order for pivoting

to be effective we need to either have a high or low value of d_{po}. This observation will be supported by experimental results later.

Tree structures attempt to cluster relevant objects together, so that d_{po} is minimized. When we have a symmetrical distance distribution, we will show that high values of d_{po} are as important for object pruning as low values. The new structures that we will introduce will utilize this observation.

3.2 Evaluating Indexing Methods

The performance characteristics of similarity searching in metric spaces are affected by a number of factors. It is almost always possible to contrive a situation that favors any one index structure over another. The first question that needs to be asked is whether it is worth using an indexing structure at all. In the paper "When is "Nearest Neighbor" Meaningful?" [7], the authors rightly argued that, in high dimensions, many of the published index structures actually perform worse than brute-force sequential scan. They give the example of relational database optimizers that refuse to use an index unless it can filter out at least 90% of the records. This is based on the fact that seek operations on disk are much more expensive than sequential reads. Although metric structures are usually meant to handle large and complex objects, the point is still valid to a degree given the extra cost of using an index structure.

There are a number of factors that can affect how well a particular scheme may perform. The distributions of objects, the range of query, and the computational

cost of distance computations can all have different effects on various methods. Some of them can handle objects in low dimensions better, and some may work better for small ranges. The computational overhead (elements of the search other than distance computations) of an algorithm becomes less important as the computational cost of distance computations increases. Factors that determine the efficiency of a structure include construction time complexity, storage requirements, and how costly it is to process dynamic manipulations.

3.3 Modeling Query Cost

Here we will offer a theoretical analysis of the query performance of global vantage points-based methods. Our analysis assumes that the distance distribution is known. The actual distance distribution can be approximated by taking a sample of object distances.

Given a database object o, pivot p, and query radius r, we define the function $PF(o, p, r)$ to be the probability that p fails to eliminate object o for a query of radius r. Let $f()$ denote the distance probability distribution and let $F()$ denote the cumulative distribution function. the distance between o and p as d_{op}, and assuming that all objects conform reasonably with the distribution function, we can approximate this failure probability as the ratio of objects that are expected to have distances between $d_{op} - r$ and $d_{op} + r$ to p, that is

$$PF(o, p, r) = \int_{d_{op}-r}^{d_{op}+r} f(x)dx = F(d_{op} + r) - F(d_{op} - r) \tag{3.1}$$

In order to generalize Equation (3.1) to any object and any pivot, we assume that pivots and objects are distributed independently, which is the case for the classic vantage points approach. Under this assumption, the expected probability of the failure of the elimination of an object, $EPF(r)$, is as follows.

$$EPF(r) = \int_{-\infty}^{\infty} f(x) \cdot (F(x+r) - F(x-r))dx \tag{3.2}$$

If a vantage points approach uses k pivots, then the total cost of a query, $qcost(r)$, involves computing the distance of the query object to the pivots and then computing distances to the objects that have not been eliminated by the pivots. Thus we have

$$qcost(r) = k + n \cdot (EPF(r))^k \tag{3.3}$$

The M-tree structure has a cost model that also depends on the distribution function [16]. There also has been some work to approximate distance distributions. It was argued that most of the real data follows an exponent rule [22] where the number of objects that are within a distance of r can be approximated by r raised to a constant. However, this assumption may be too general, especially for clustered datasets. Another approximation of the distance function was performed based on the mean and standard deviation of the distance distribution. This approach is also insufficient where the distribution has several peaks or when the peaks are not symmetrical.

To the best of our knowledge, Equation (3.2) has not been used for query cost

estimation before. Equation (3.3) have been used [14] to successfully predict the query cost of global vantage points-based methods that have a suitable distribution for their model.

3.4 Experiment Data

In this section we will introduce the different types of real and synthetic data distributions we have used for our experiments. The primary means of understanding this data will be through their distance distributions.

The majority of our experiments were performed on uniformly distributed vector spaces. Each of the dimensions had values between 0 and 1, so different dimensions had different distance ranges. Figure 3.2 shows the distance distribution in two different ways. In the regular version, we see that the mean distance value increases as dimension increases. When the distance values are normalized to the range [0, 1], the mean of the distributions is exactly the mid point, and the distance values are scattered into narrower ranges as we increase dimensionality.

Other types of vector dataset we used are as follows.

Uniform(s, e) coordinates are uniform over the range $[s, e]$.

Gaussian coordinates are drawn from a Gaussian distribution of zero mean and unit variance.

Laplace coordinates are drawn from a Laplacian distribution of zero mean and unit variance.

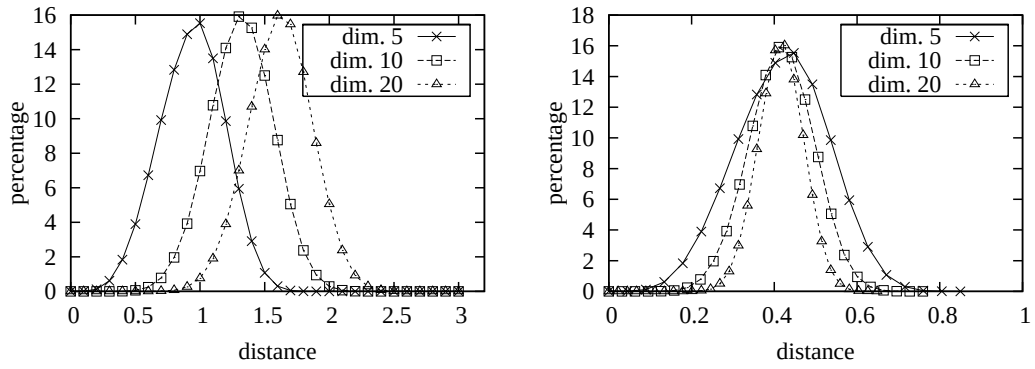


Figure 3.2: Distance distribution in unit hyper-cube for 5,10 and 20 dimensions (left) and distances normalized to 1.0 (right) using a bucket width of 0.1

Clust_Gauss($nclust$, $cdev$) coordinates are partitioned into $nclust$ number of clusters whose centers are distributed uniformly in a unit hypercube, and within each cluster the coordinates are drawn from a Gaussian distribution having the cluster center as mean and a standard deviation of $cdev$.

Figure 3.3 shows the histogram of distances for these distributions. Note that the clustered Gaussian distribution has two separate peaks; the smaller and closer one representing the distances within the cluster, the other one representing the distances between clusters.

Figure 3.4 provides more configurations for clustered data. We see that as we increase the number of clusters, the number of objects per cluster drops, and the peak that represents the objects in the same cluster becomes smaller.

We also experimented with real metric data sets. One dataset we employed involved a set of about 20,000 English words collected from the Unix operating system. We used the edit distance as our metric. The distribution is shown in

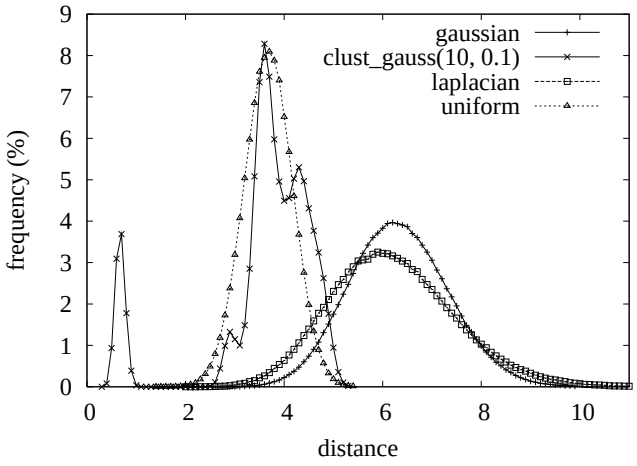


Figure 3.3: Distribution of various types of vector data for a bucket width of 0.1 in 20 dimensions.

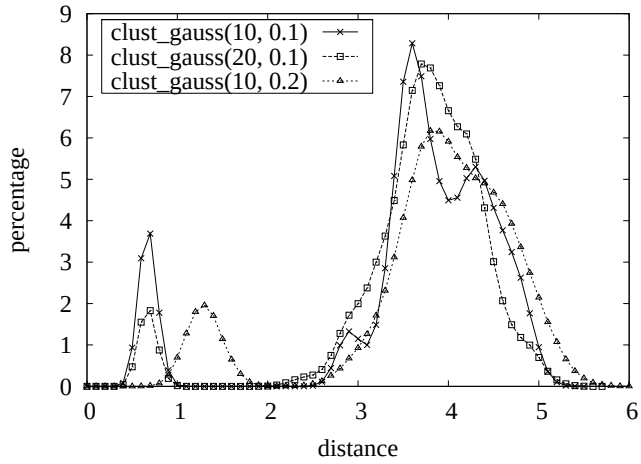


Figure 3.4: Distribution of various types of clustered vector data in 20 dimensions for a bucket width of 0.1.

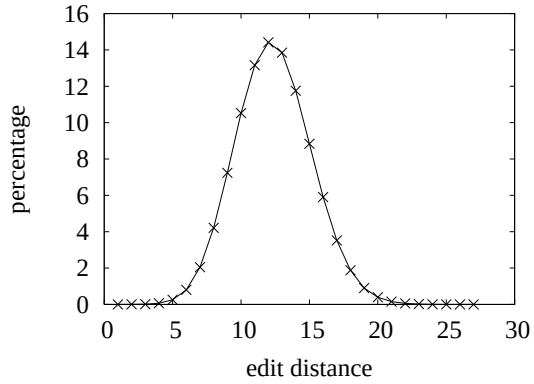


Figure 3.5: Distance distribution of English words under edit distance.

Figure 3.5. Even though words of similar spellings would be expected to demonstrate some clustering properties, the clusters are too small to be evident in the graph. The distribution looks very much like our synthetic vector data, having a fairly symmetric form.

Another dataset we used was a set of 1800 images having a resolution of 128 by 96 pixels taken from the Corel collection [1]. For each image there is a 64-dimensional color histogram and a 62-dimensional feature vector generated by Gabor texture filters [32]. Figure 3.6 illustrates the distance distribution of these images. We again note that these distributions are very much like others we have presented earlier.

Finally, we considered a document data set consisting of a set of web pages obtained through Google Inc. [2]. The web pages were converted into term vectors, and cosine distance [38, 27] was used as a metric. We see in Figure 3.7 that this distribution is very difficult in the sense described in Chapter 1 because of the high concentration of distances near the maximum value.

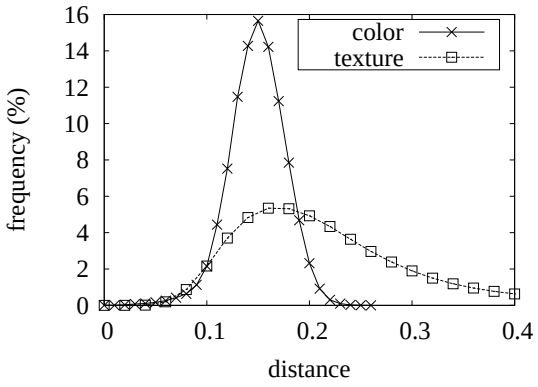


Figure 3.6: Distance distribution of image data under two separate metrics.

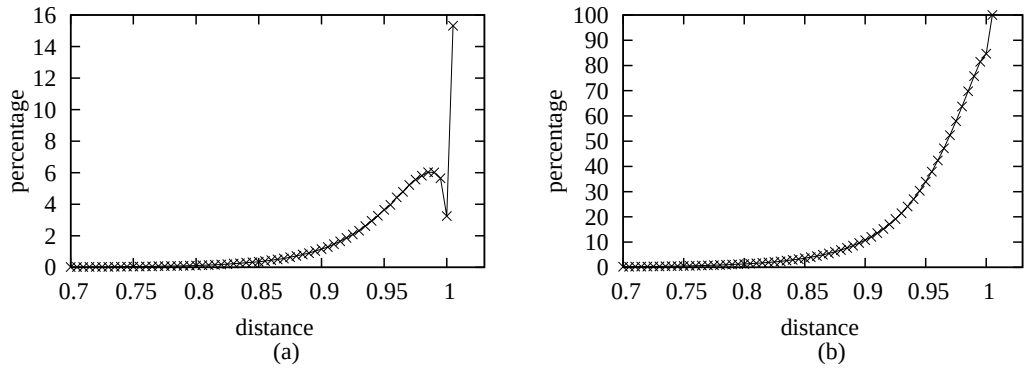


Figure 3.7: The density function (a) and cumulative density function (b) of web page data under cosine distance using a bucket width of 0.005

Chapter 4

Pivot Prioritization Schemes

In this chapter, we introduce a new approach to improve the performance of pivot-based methods, which we call *prioritized vantage points*. Recall that query processing in similarity search is in essence a process of identifying which database objects lie within the query range and which do not. Pivot-based methods operate by using precomputed distances from database objects to a collection of distinguished objects called pivots. A database object is said to be *eliminated* by a pivot if this object can be classified as lying inside or outside the query range based solely on its distance to one or more pivots, without explicitly computing its distance to the query object. The *effectiveness* of a pivot is defined to be the expected ratio of database objects this pivot succeeds in eliminating. We will show that pivots have varying degrees of effectiveness in eliminating objects depending on their distance to the query object. Based on this observation, prioritized vantage points structure only processes a subset of the pivots that look promising. This results in a reduction in CPU overhead at the cost of making (ideally relatively few) more distance computations.

Research to date on pivot-based methods has focused on structures that reside in main memory. This has limited the number of pivots used in experiments. The FQA structure [13] overcomes this by using fewer bits to encode distance informa-

tion per pivot per database object, but using fewer bits also reduces the accuracy of pivots. Rather than uniformly reducing the number of bits of precision, we introduce a new approach, called *Kvp*, which intuitively seeks to keep the most effective distance information. The Kvp structure is designed to prioritize pivot distance data to reduce space requirements. The Kvp structure is an enhancement of prioritized vantage points. In addition to saving space, we will see that it also provides savings in CPU overhead.

4.1 Prioritized Vantage Points

The usual vantage points-based method, which we will call *vps* for short, stores $k \cdot n$ pieces of information, where k is the number of pivots and n is the database size. As mentioned above, we present ways to prioritize or ignore some of this data to improve space or CPU overhead at the cost of relatively small number of additional distance computations.

As mentioned above, a pivot's effectiveness in eliminating database objects depends on its distance to the query object. Figure 4.1 shows the number of objects eliminated as a function the pivot's distance to query object. The data set consists of 100,000 uniformly distributed points in 20-dimensional space for a query radius of 0.6. Observe that the pivot is most effective when it is either close to or far from the query object.

Figure 4.2 illustrates the intuition behind this observation. Here we have range queries of radius 1 on a database of English words. As the query object is closer to

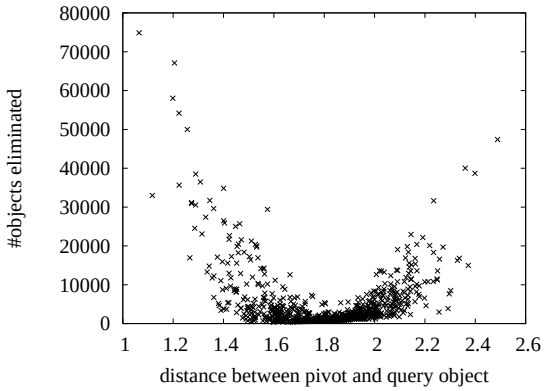


Figure 4.1: Distribution of the number of objects eliminated by pivots based on object's distances to the pivots for a query radius of 0.6 on a database of 100,000 objects uniformly distributed in 20 dimensions.

the bulk of the population, the area under the curve grows larger, therefore more database objects are saved from elimination.

In the basic vantage points approach, the distance between a query object and all of the pivots are computed, and all of these pivots are processed in an arbitrary order. Instead of processing all of the pivots, we prioritize their use based on their distances to the query object, such that close or far pivots are processed earlier than others. Note that this does not add any extra distance computations to the process.

Figure 4.3 illustrates the resulting reduction in the size of the (unpruned) search space. Here, the y-axis shows the number of objects not eliminated from the candidate list. As more pivots are processed, this number decreases.

We see that it is possible to eliminate database objects much more quickly with priority vantage-points. We can use a large number of pivots for our structure, but only use a portion of them at query time depending on the query object. Like

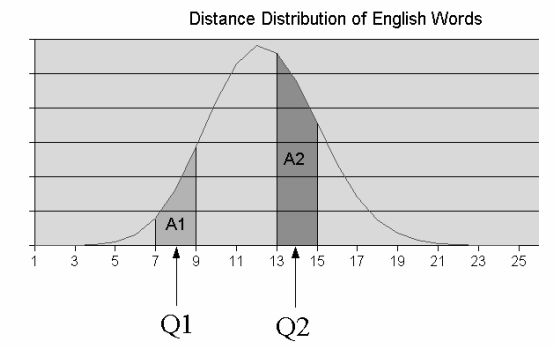


Figure 4.2: Query objects Q1 and Q2 have distances of 8 and 14 to the pivot. The areas represent the set of objects that cannot be eliminated by this pivot.

the FQA structure, this scheme does not need any additional structures other than the $k \cdot n$ distance values to the pivots. Unlike FQA, in order to achieve its CPU efficiency it does not need to reduce the number of bits of precision for the distance values, and so it is possible to use as much precision as desired.

4.2 The Kvp Structure

In this section we introduce the Kvp structure. As mentioned above, it is desirable to use pivots that are particularly close to the query object. In a similar fashion, we can expect a pivot to be more effective for objects that are close to or distant from it. At index creation time, we can find such pivots, and choose to keep only the distances to these promising pivots. We will call the result a *Kvp structure*.

With priority vantage-points, we do not know where the query object will be in advance, and so we have to keep all pivot distances. However, with Kvp the distance relations between the pivots and database elements are computed beforehand at construction time. In addition to reducing CPU overhead by first processing the

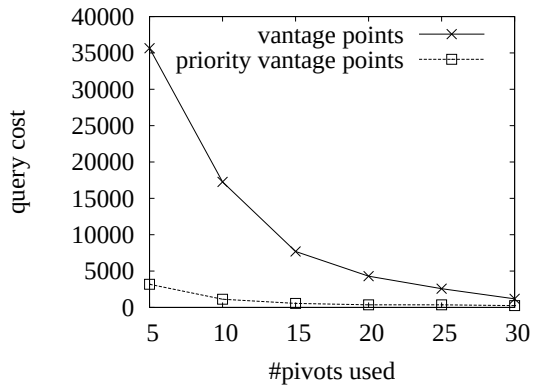
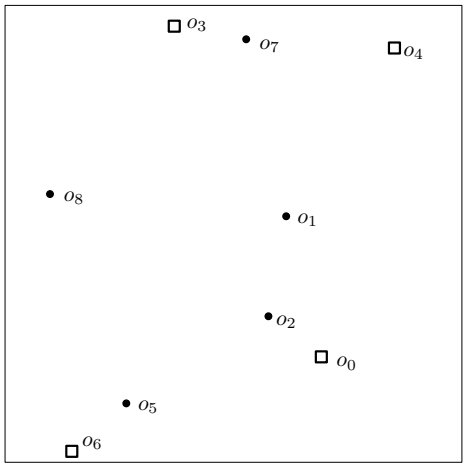


Figure 4.3: The comparison of the rate of the object elimination by the regular and prioritized vantage points approach for a query radius of 0.4 on a database of 100,000 objects uniformly distributed in 20 dimensions.

most promising pivots, we can eliminate distance computations to the less promising pivots, thus decreasing the space requirements. There are two ways this can be implemented. One way would involve the usual layout, where every pivot stores an array of distances to all the database objects. The object distances can be sorted so that binary search can be used to quickly determine set of objects that are eliminated. Another way to implement the basic idea is to have a collection of object entries, where each object entry stores the distances to its selected pivots. The benefit of this latter approach is that it is very easy to insert or delete objects from the database, since there is no global data structure that keeps information about the objects. We preferred to implement Kvp using the second approach. Figure 4.4 shows an example.

Other than the fact that Kvp only stores a subset of pivot distances, the way it processes queries is identical to the classical global pivot-based method. For each



(a)

	o_1	o_2	o_5	o_7	o_8
o_0	0.32	0.16	0.46	0.74	0.73
o_3	0.50	0.68	0.86	0.16	0.45
o_4	0.45	0.67	1.01	0.33	0.83
o_6	0.72	0.54	0.16	1.02	0.60

(b)

obj. id	pivot id	distance	pivot id	distance
1	0	0.32	6	0.72
2	0	0.16	3	0.68
5	6	0.16	4	1.01

(c)

Figure 4.4: A sample database of 9 vectors in 2-dimensional space, and an example of the Kvp structure on this database that keeps 2 distance values per database object. (a) The location of objects. Boxes represent objects that have been selected as pivots. (b) The distance matrix between pivots and regular database objects. For each object, the 2 most promising pivot distances are selected to be stored in Kvp (indicated by using gray background color). (c) The first three object entries in the Kvp. Each object entry keeps the id of the object, and an array of pivot distances.

database object it maintains a lower and upper bound for the distance to the query object. Each pivot is used to attempt to tighten these bounds. After processing all possible pivot distances, if the bounds are good enough to either discard the object as out of the query range, or prove that it is within the query range, we avoid computing the actual distance between the object and the query object. Otherwise we compute this distance. Figure 4.5 shows the query performance of Kvp as a function of the number of pivots stored for a query radius of 0.4 in 20 dimensions. The results that are labeled as "random" choose the next pivot to be used randomly, simulating a classic vantage-points structure. Kvp methods first process close and distant vantage points. For example, assume we have a Kvp structure that has a pool of 50 prioritized vantage points, which we refer to as Kvp 50. In the sorted array of pivot distances 0 through 49, the processing proceeds in the order: 0, 49, 1, 48, 2, and so on. As the number of pivots in the pool is increased, the chances of finding a better suited pivot also increases. Varying the number of pivots provides flexibility to improve query performance by spending more time at construction time without increasing space and CPU overhead.

We can see from the graphs that Kvp, like priority vantage-points, can eliminate database objects much faster than the classic approach. Even though it uses less space than priority vantage-points, it produces similar query results. The reductions in CPU overhead and space are very closely related. For example, if we were to store and process two pivots instead of ten, we would be saving a factor of 5 in both storage and computational overhead. In the case of radius 0.4 for 20 dimensions, we observed the same query performance in terms of the number of

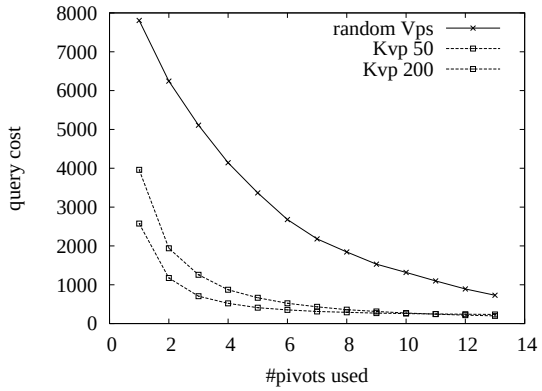


Figure 4.5: Query performance of the Kvp structure, for vectors uniformly distributed in 20 dimensions.

distance computations.

4.3 Secondary Storage

Access patterns of pivot-based structures are targeted toward minimizing CPU time, but they are not always suitable to be stored on disk. For example, performing binary search in secondary storage is expensive as it involves many seek operations. Disks are much better at performing sequential scans. The Kvp structure is quite amenable for data that are stored on disk. It only requires a sequential scan of distance values. It does not involve a heavy processing burden, so processing time does not dominate over I/O time. It requires relatively little memory, since only the vantage objects, the query object, and the distance vector of the processed object is needed.

4.4 Memory Usage

To the best of our knowledge, Kvp and its variants HKvp and EcKvp which will be introduced in later chapters are the only structures to store fewer distance values than the classic vantage-point methods. As with FQA, another way of storing less information is by discretization, so that fewer bits are used for the distance values. Consider its simplest form where the intervals have equal width, using b bits in a metric space where the maximum distance is D_{max}. This will map distances into buckets of width D_w where

$$D_w = \frac{D_{max}}{2^b}$$

Since all the distances in the same bucket will be assigned the same distance value, we will have a maximum error of D_w per distance value. Assuming query objects are distributed uniformly, we can approximate the error to $D_w/2$. Therefore, if a pivoting operation with radius r extends its range of objects that cannot be eliminated into $r + \frac{D_w}{2}$, we will be able to simulate a query process where only b bits of distance information are available.

Experiments were performed where the number of bits used per distance value varied, and the number of pivots per object was adjusted accordingly to keep the total size fixed. In the following figures, a label $\langle r, b, Vps \rangle$ stands for classical vantage-points algorithm using b bits per database object for a query radius of r, and $\langle r, b, Kvp\ k \rangle$ stands for Kvp with k pivots.

We can see from Figure 4.6 that using more bits is more effective than using more pivots for values below 7 bits. Also observe that using more bits is almost as

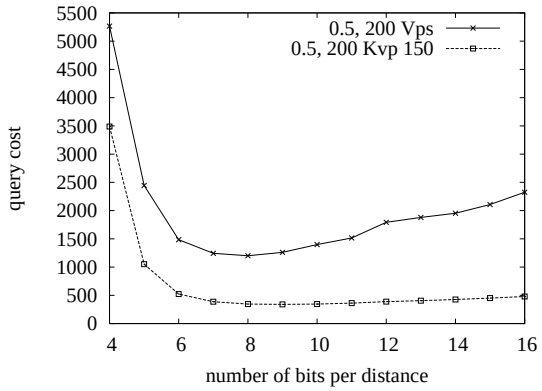


Figure 4.6: Query performance in terms of distance computations when the total memory usage is kept at a constant and the number of bits per distance value and the number of pivots stored is varied. The database has 10000 uniformly distributed vectors in 20 dimensions.

important as using more pivots even after that point.

One of the results that emerged from these experiments is that, ignoring construction cost, for a given Kvp structure and a metric space, there is an optimal number of bits. This is due to the trade-off between the number of pivots and the number of bits. Another important result is, using a precision that is less than a critical value (such as 6 in this case) results in very high performance penalties. There is a sharp decline in query costs as we increase the number of bits. This explains why a structure like the vp-tree, which effectively uses very few bits of precision by not actually storing the distances but a limited number of distance ranges, is significantly inferior in performance. As we increase number of bits above this critical value and decrease number of pivots we see that the performance degrades, but not as dramatically.

Radius	#Bits	Cost Ratio (Vps/Kvp)
0.7	200	2.56
0.5	100	5.22
0.5	200	3.53
0.5	400	1.73
0.4	200	1.84

Table 4.1: A sample of improvements Kvp provides over Vps

Typical tree-based structures rely on the partition information implicit in each internal node to prune out subtrees from search. Since nodes are divided into limited number of subtrees, this amounts to using insufficient information for pruning. Although vantage-point algorithms are very simple in structure, it is their ability to use a large number of bits of distance information that gives them such good performance.

Figure 4.7 compares two methods under varying radii and memory parameters. Observe that the Kvp structure performs consistently better than Vps. Table 4.1 summarizes the speedups that Kvp provides over Vps in terms of the number of distance computations.

In general, vantage-point structures need more space and construction time than the relative lightweight vp-tree. The Kvp structure adds flexibility that allows the index structure to better fit a given domain. In contrast the vp-tree does not have this flexibility to tradeoff space for better query time. We ran some experiments

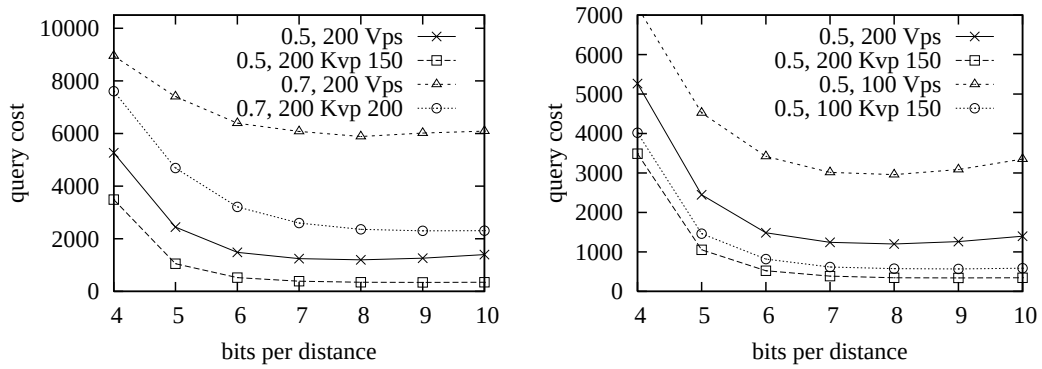


Figure 4.7: Effects of changing search radius and amount of total memory on performance. The database has 10000 uniformly distributed vectors in 20 dimensions.

to see how this space-query time tradeoff works for the Kvp structure. We assume a balanced tree is stored in the typical "heap layout" in an array, such that a node at index i has its subtrees at index $2i$ and $2i + 1$ [26]. Therefore each internal node does not require any pointers, and only needs to store a distance value using 64 bits. Since the leaves of the tree do not need to store anything other than the object itself, this would amount to using 32 bits per object. Given a database of size 10,000, the vp-tree performs approximately $\log_2 10,000 \approx 13$ distance computations per database object at construction time.

To produce a fair comparison, we created Kvp structures that use various multiples of 13 pivots and 32 bits of distance information. The construction cost and space are the independent variables. The construction cost dictated how many pivots the Kvp structure could use, and for a given number of pivots, the space dictated how many bits we could use per database object, which should be less than bits per distance value times the number of pivot distances stored per object. For a given space limitation of B bits per object, we varied the number of bits per pivot

distance, b, and calculated the corresponding number of pivot distances to store as $\lceil B/b \rceil$.

Figures 4.8 and 4.9 summarize the experiment results. The x-axis, *construction cost*, is measured as the ratio of the construction cost used by Kvp to a construction cost of 13 pivots per database object. The y-axis, *space*, indicates the multiple of 32 bits that was used per database object. The z-axis, *query improvement*, is the ratio of vp-tree query cost to the particular Kvp query cost in terms of distance computations.

Figure 4.8 shows a query improvement of more than 70-fold in Kvp over the vp-tree using 8 times the storage of the vp-tree and 2 times the construction cost of vp-tree for vectors in 20 dimensions, whereas we have an almost 50-fold improvement for English words. When we use approximately the same resources as the vp-tree, we obtain query cost improvements up to a factor of 2.59 for vectors and 6.97 for English words.

Note that Figures 4.8 and 4.9 do not show optimal results, since we see some performance decrease even as we increase construction cost. Using more pivots does not help if we do not have the space to store them. Since Kvp computes distances to all the pivots, we have a problem when optimal number of distance computations is below the number of pivots used by Kvp. For a given space bound, there is an optimal construction cost that Kvp should use, and this trade-off obviously is not reflected on our graphs. In the next chapter, we will introduce a new structure, called the *HKvp*, that can overcome this problem.

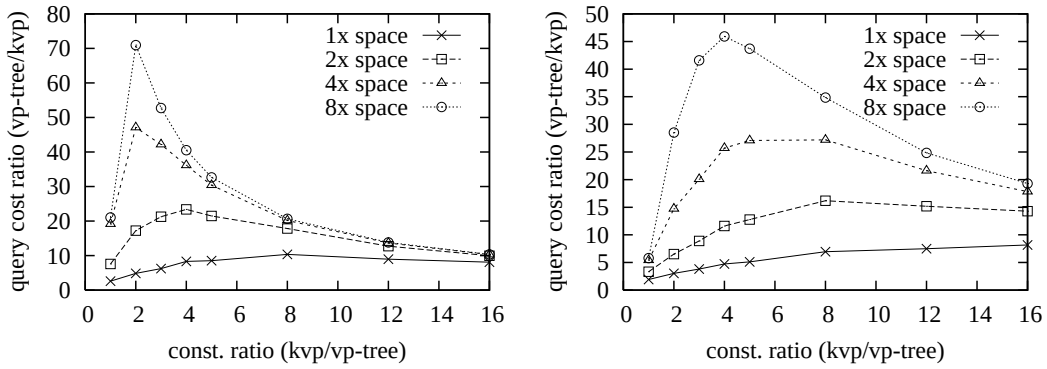


Figure 4.8: Comparison of Kvp and the vp-tree in 20 dimensional vector space for query radius of 0.3 (left) and 0.4.

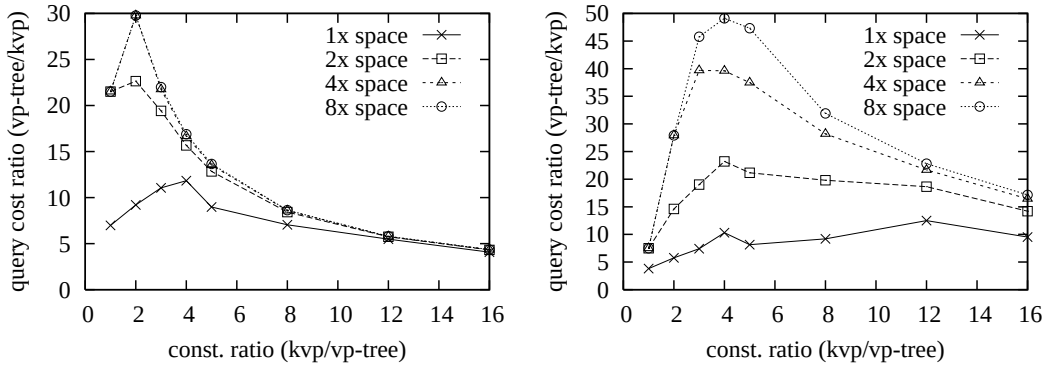


Figure 4.9: Comparison of the Kvp and the vp-tree for queries performed in the metric space of English words for query radii 1 (left) and 2.

4.5 Comparison of Kvp and Tree-Based Structures

Using a Kvp structure, one can easily vary a number of parameters, including the construction cost, the number of pivots used per object, the number of pivots stored per object, the number of pivots processed at query time per object, and the number of bits used per distance value. It is possible to observe how these changes affect performance.

In a sense, it is possible to view most of the existing structures as variants of the vantage point-based methods. For example in a vp-tree with a branching factor of k, there is one pivot per node, all the objects in subtrees can be eliminated with their distances to this pivot, and number of branches have an affect similar to the number of bits used. For a database object, there are approximately as many pivots as the height of the tree. This view explains why changing k in the vp-tree has little affect on query performance, since as k increases and pivots become more precise (which is similar to using more bits), the height of the tree becomes shorter and there are fewer pivots per database object. A major problem with the vp-tree is that the only data that is used are the cutoff values. The individual distances of objects to the pivots are computed but then discarded.

From the perspective of vantage points, it is also easier to see why GNAT with branching factor k improves on the vp-tree. In GNAT, there are k pivots per node, and the distance ranges of k subtree to these pivots are stored. One slight disadvantage of GNAT is that ranges of distances to a pivot can overlap. However, instead of having just one pivot per one, objects in GNAT make use of k pivots.

Tree-based methods have two advantages over the classical vantage points methods. Whereas a pivoting operation involves one object in Vps, it usually involves groups of objects in tree-based structures. As explained before, this is something that only improves on the CPU overhead, and has a negative impact on the number of distance computations. Secondly, tree-based methods attempt to divide the space into clusters in order to benefit from the locality of pivots. This is similar to what priority vantage points and Kvp try to accomplish. While tree structures have varying degrees of success in clustering similar objects together, Kvp takes a direct approach and precisely computes the closest pivots. In addition, Kvp properly makes use of far pivots as well.

4.6 Conclusions

In this chapter we presented two new structures, the Priority Vantage Points and the Kvp. We showed that pivots are more effective if they are close to or distant from the query object. The Priority Vantage Points structure utilizes this fact to only process promising pivots to save from the CPU overhead. Kvp takes this idea further, by only storing and processing relevant distances between database objects and the pivots. We showed that there is a performance penalty involved in terms of number of distance computations when we ignore some of the distances, but it is not significant thanks to our prioritization schemes that process more promising pivots earlier.

We also showed that the Kvp structure can support dynamic insertions and

deletions. It can also be stored on secondary memory very easily and only requires sequential scans.

Chapter 5

The HKvp Structure

One of the ways to evaluate metric space indexing methods is to compare their query performance in terms of the number of distance computations versus their construction costs, ignoring the space consumption. We shall see that, while some index structures can take advantage of added preprocessing to improve query time, others either cannot or produce only minor improvements. We will show that global pivot-based methods like LAESA, Spaghettis, FQA, Omni Sequential and Kvp take advantage of their preprocessing time more effectively than tree-based structures. Although LAESA is one of the earliest pivot-based methods, it remains as the best performer by this criteria. In part this is because more recent methods have focused on reducing the relatively high space and extra processing times that arise with global pivot-based methods.

In this chapter we introduce a new index structure, called *HKvp*. We will show that this structure can reduce the query times by up to 49% compared to LAESA. HKvp is based on the Kvp structure so it also has the ability to decrease both the space usage and the extra processing time, which LAESA lacks.

5.1 Tradeoffs for Existing Index Structures

Before introducing the HKvp structure, we will take a closer look at the relationship between the construction cost and query performance for three different structures, which represent the three different families of metric space index structures as described in Chapter 2. GNAT was chosen as standard example of clustering methods since others in the same family sacrifice the pivot precision for allowing dynamic operations. We chose Kvp as the representative for global pivot-based methods. The various methods within this class that we have introduced usually have identical query performance in terms of the number of distance computations; they only differ in their space and computational overheads. For local pivot-based solutions, there were only two alternatives for continuous distance values, the vp-tree and the mvp-tree. The vp-tree was chosen over the Mvp-tree because of its simplicity and the fact that a majority of other algorithms use it as a benchmark for comparison. Also, the mvp-tree puts some distance information in leaf nodes, making it a hybrid between the vp-tree and global pivot-based methods [15].

The basic assumptions underlying the working principles of global pivot-based methods are that we have a very large database with a high intrinsic dimension and a very expensive distance function. Therefore, it is assumed that the number of pivots is quite limited compared to the database size. There are cases when this assumption does not hold, however, for example when the database size is limited or the application requires a high number of pivots to meet a given set of performance requirements. In other words, the ratio of the number of pivots to the database size

is not always as low as assumed.

Given a query object q, a typical pivot-based structure begins the search process by computing the distances between all k pivots and the query object q. Assume that the average probability of a pivot not eliminating an object for a given radius is f. In this case, after processing k pivots, there would still be f^k objects that remain not eliminated. Therefore the total cost of the query would be:

$$Cost(q, r) = k + n \cdot f^k$$

It can be argued that this function has an optimal k value for a given query object and radius. This observation is supported in Figure 5.1. The cost of a query using k pivots is at least k, if there is a solution that would require fewer distance computations, typical pivot-based structures will fail to find it. Figure 5.1 also shows that after the optimal number of pivots is reached, the cost is dominated by the number of pivots.

In the vp-tree, the vantage points in the nodes govern only the associated subtree. The out-degree of the nodes of the tree determines its height and therefore determines the total number of pivots. The higher the tree, the more internal nodes will be produced, resulting in more pivots, more efficient query processing, but higher construction costs. On the other hand, increasing the branching factor results in narrower distance ranges, increasing the effectiveness of the pivots as discussed earlier. Figure 5.2 shows the query performance of the vp-tree based on the construction cost for different branching factors. We see that there is a strong correlation between the construction cost and query performance. Similar to Kvp,

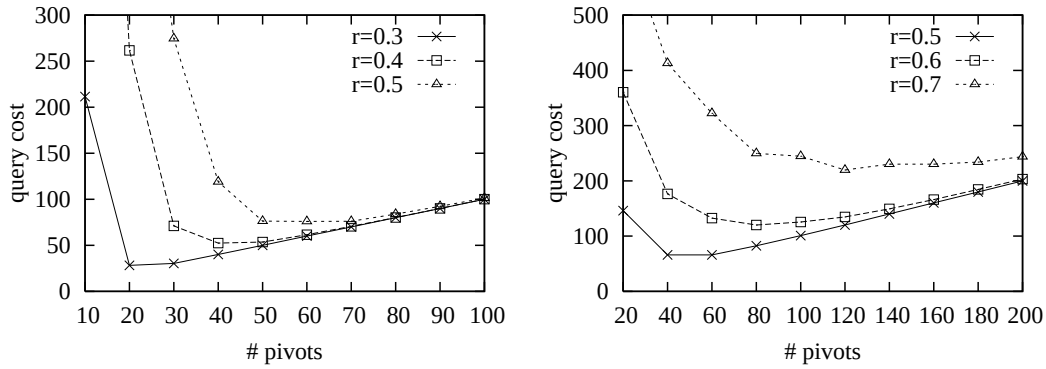


Figure 5.1: Query performance of Kvp in terms of the number of distance computations for varying numbers of pivots. We see that there is an optimal setting for the number of pivots when either the query is relatively easy (left) or when the number of pivots is high (right). For both figures, the database is composed of 10,000 vectors uniformly distributed in 40-dimensional Euclidean space.

there is an improvement on query cost as we increase the construction cost. The reason there is not much difference between different configurations is probably because the precision loss we encounter for low branching factor values, as discussed before. Observe that as construction time increases, the vp-tree does not improve query times as effectively as Kvp does. Otherwise, the two graphs are quite similar.

In GNAT, increasing the branching factor of the tree decreases the height in a similar fashion as the vp-tree. The difference is that the number of pivots per level also increases in direct proportion to the branching factor. As a result, increasing the branching factor of GNAT results in much higher construction times than the vp-tree. Figure 5.2 also offers a review of GNAT for different branching factors. We can see that the construction cost can grow as high as 50 million distance computations, compared to about 100 thousand for the vp-tree.

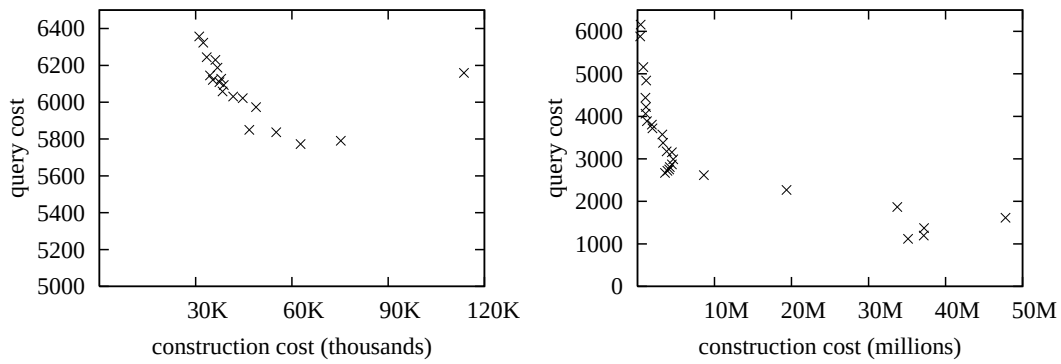


Figure 5.2: Query performance of vp-tree (left) and GNAT (right) in terms of the number of distance computations versus the construction cost for a query radius of 0.44. The database is composed of 10,000 vectors uniformly distributed in 40 dimensions.

Figure 5.3 summarizes the overall picture. In order to make the data presentable, we only showed a limited number of GNAT configurations. We can see that GNAT has very high construction times, and yields worse performance than the vp-tree in some cases even when given more construction time. Due to the nature of the vp-tree, it is not possible to freely increase the construction cost, since it achieves its maximum when the branching factor is 2. The reason GNAT outperforms the vp-tree is that it can use more pivots, thus can store more distance information. We also see that the Kvp yields much better results when using same amount of setup time as other tree methods.

As Figure 5.1 reveals, however, Kvp is not perfect in its utilization of construction time. We see that it may perform worse when allowed more pivots. To solve this problem, we need a way of eliminating pivots as well as ordinary database objects, which would result in the possibility of processing a subset of the avail-

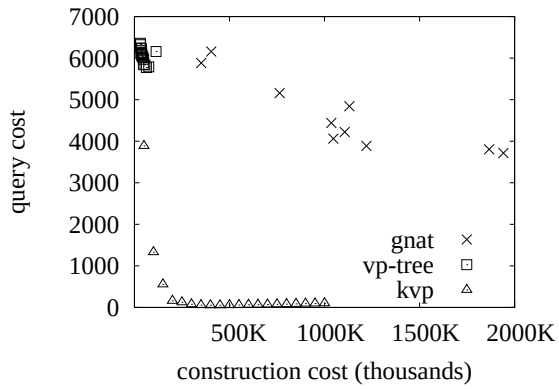


Figure 5.3: Performance comparison of three structures by their construction costs for a query radius of 0.44. The database is composed of 10,000 vectors uniformly distributed in 40 dimensions.

able pivot set. AESA is a pivot-based structure where all the database objects are selected as pivots [43]. All the distances between objects are computed, and all this information is used to successively process pivots and eliminate other pivots. LAESA is a modification of this approach in which only a subset of the database are used as pivots [35]. The basic algorithm used by LAESA, adapted to a range search for a query object q of radius r over set of objects O is presented in Algorithm 1.

The algorithm maintains a global array for objects to hold the lower bounds for the distances to q. An object can be eliminated if its lower bound is greater than r. The lower bounds are also used to choose the next most promising pivot from the unprocessed list. It was reported [35] that choosing the pivot with the least lower bound gives the best results. The criterion for eliminating pivots is based on a pre-determined ratio. LAESA is one of the earliest index structures, and so has been surpassed with the introduction of more recent pivot-based structures that use

space and processing time in a more efficient manner.

Algorithm 1 Range Query for LAESA. Note that all the distances between pivots are pre-computed.

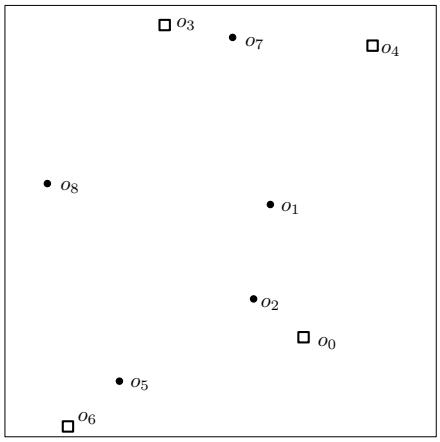
Input Query object q, radius r, set of database objects O, set of pivots P.

Output *resultSet*, set of objects that qualify for the query.

```
toProcess ← O
choose an arbitrary object s ∈ P
while toProcess ≠ {}
    compute D_sq = d(s, q)
    toProcess ← toProcess − {s}
    if D_sq <= r
        resultSet ← resultSet ⋃ {s}
    for all u ∈ toProcess
        if s ∈ P
            update u's approximate distance to q
        if u can be eliminated from search
            if u ∈ P and pivot elimination criteria holds
                toProcess = toProcess − {u}
            else if u is an ordinary object
                toProcess = toProcess − {u}
    set s to be the next most promising object from toProcess
```

5.2 The HKvp Structure

We introduce a new structure, called *HKvp*, which stands for "High performance Kvp". HKvp maintains the complete set of distances between pivots, and eliminates pivots as well as ordinary database objects like AESA. Moreover, HKvp is an extension of Kvp, therefore it exploits Kvp's capability to reduce space and processing times. Applying other well-known CPU or space reduction methods on LAESA is not straightforward. An example HKvp is shown in Figure 5.4.



(a)

	o_1	o_2	o_5	o_7	o_8
o_0	0.32	0.16	0.46	0.74	0.73
o_3	0.50	0.68	0.86	0.16	0.45
o_4	0.45	0.67	1.01	0.33	0.83
o_6	0.72	0.54	0.16	1.02	0.60

(b)

	o_6	o_4	o_3
o_0	0.62	0.71	0.82
o_3	0.99	0.49	
o_4	1.17		

pivot distance matrix

obj. id	pivot id	distance	pivot id	distance
1	0	0.32	6	0.72
2	0	0.16	3	0.68
5	6	0.16	4	1.01
7	3	0.16	6	1.02
8	3	0.45	4	0.83

object entries

(c)

Figure 5.4: A sample database of 9 vectors in 2-dimensional space, and an example of the HKvp structure on this database that keeps 2 distance values per database object. (a) The location of objects. Boxes represent objects that have been selected as pivots. (b) The distance matrix between pivots and regular database objects. For each object, the 2 most promising pivot distances are selected to be stored in HKvp (indicated by using gray background color). (c) The HKvp consists of the distances between pivots and the object entries. Each object entry keeps the id of the object, and an array of pivot distances.

LAESA only makes use of lower bounds for pivot and object elimination. HKvp also makes use of upper bounds for object elimination and choosing the next pivot to process. Another improvement of HKvp is the maintenance of distance bounds for pivots. Rather than eliminating a pivot right away, it waits until all pivots are known to be inside or outside the query range, when we have the greatest information about the bounds of a pivot, and then it chooses which pivots to explore further and calculates their exact distances to the query object. Instead of discarding those pivot bounds that are only approximate, it uses them for elimination as well.

The first phase for HKvp involves computing the distance bounds for pivots. Some of these distances to query objects are computed exactly, and the rest are only approximations based on distance relations to other pivots. In the second phase, the objects are visited and the pivot bounds are used to eliminate them. The corresponding algorithm is presented in Algorithm 2. An illustration of the first phase of the HKvp is given in Figure 5.5.

5.2.1 Pivot Selection Policy

We have seen that HKvp progressively computes distances to pivots that are expected to be promising. In this section, we explain how we determine how valuable a pivot is expected to be. Note that this discussion is not about the selection of which subset of the objects to use as pivots at the construction time.

Previous experiments have shown that the best way to select the next pivot to process is to choose the one with the lowest lower bound for the distance to the

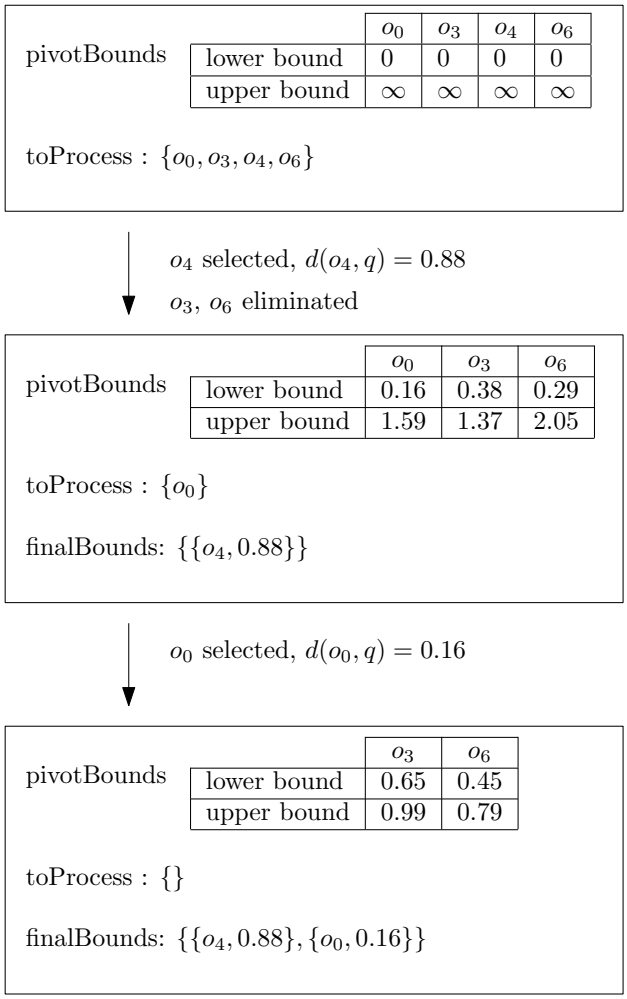


Figure 5.5: A sample run of the first phase of the HKvp range query algorithm on the database given in Figure 5.4. The query object is the vector (0.73, 0.07), and the query radius is 0.2. First, the pivot o_4 is selected for being processed. Based on their distances to o_4, the other pivots improve their bounds on the distance to q. After this step, o_3 and o_6 are eliminated and removed from the processing queue. This leaves only o_0 to process. o_3 and o_6 improve their bounds even more based on their distance to o_0, since this may improve the likelihood of the elimination of regular database objects later on. After processing o_0 the range query may proceed to processing the regular database objects based on the pivot distance information compiled in *pivotBounds* and *finalBounds*.

Algorithm 2 Range Query for HKvp. Note that all the distances between pivots are pre-computed.

Input Query object q, radius r, set of database objects O, set of pivots P.

Output $resultSet$, set of objects that qualify for the query.

initialize all pivot's bounds as $[-\infty, +\infty]$ in $pivotBounds$
$toProcess \leftarrow P$
$finalBounds \leftarrow \{\}$
while $toProcess \neq \{\}$
    get the most promising pivot p based on $pivotBounds$
    compute $D_{pq} = d(p, q)$
    if $D_{pq} <= r$
        $resultSet \leftarrow resultSet \bigcup \{p\}$
    $toProcess \leftarrow toProcess - \{p\}$
    remove the bounds of p from $pivotBounds$ and put into $finalBounds$
    for all bounds $[S_v, E_v] \in pivotBounds$
        update $[S_v, E_v]$ based on D_{pq} and $d(v, p)$
        if $E_v <= r$
            $resultSet \leftarrow resultSet \bigcup \{v\}$
            $toProcess \leftarrow toProcess - \{v\}$
        else if $S_v > r$
            $toProcess = toProcess - \{v\}$
$ncompute \leftarrow (1 - pivotDropRate) \cdot |pivotBounds|$
for $ncompute$ times do
    get the most promising pivot p based on $pivotBounds$
    compute $D_{pq} = d(p, q)$
    if $D_{pq} <= r$
        $resultSet \leftarrow resultSet \bigcup \{p\}$
    $toProcess \leftarrow toProcess - \{p\}$
    remove the bounds of p from $pivotBounds$ and put into $finalBounds$
    update bounds of other pivots
put the rest of the elements in $pivotBounds$ to $finalBounds$
process the regular database objects like Kvp by using $finalBounds$

query object [40, 35]. This chosen pivot is then compared against the query object, and based on this distance, the bounds for other pivots are updated.

We explore two new approaches for selecting the next pivot: the pivot with the highest upper bound, and the pivot with the greatest difference between its lower and upper bounds. We also consider a combination of these policies. The idea is that using several policies at the same time will improve the evaluation of the pivots that have different strengths over each other. In order to control the relative degree of importance of these policies in more detail, we make use of two parameters. *FarToCloseRatio* dictates the ratio between "presumed" far pivots to close pivots. Likewise, *WideToCloseRatio* is used to control the proportion of "wide" candidates. In the following figures, a ratio $a :: b$ denotes a FarToCloseRatio of a and WideToCloseRatio of b. Using 1::1 makes use of three policies in equal amounts, whereas 0::0 only uses the close pivots, inf::0 uses only far pivots, and 0::inf uses only wide pivots. Active pivots are stored in a processing queue. As each pivot is processed, it eliminates some other pivots from the processing queue. In the following figures we plot the number of pivots remaining to be processed as a function of the number of pivots already processed.

Figure 5.6 shows results for 500 pivots chosen from a uniform distribution in 20-dimensional space. We get the worst results from processing far pivots, whereas wide pivots provide a slight improvement. The best of the pure policies is using close pivots, which is supported by prior research [35]. There is still room for improvement however, as shown by the equal combination policy denoted by 1::1. When we mix the policies more proportional to their individual performances, such as 0.7::0.9, we

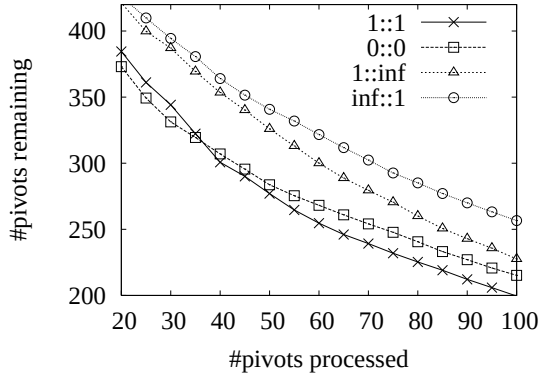


Figure 5.6: Pivot reduction in dimension 20 for a query radius of 0.9 and 500 pivots obtained and improvement of up to 10% over the original idea.

Our results suggest that, especially in relatively high dimensions, selecting the next pivot based on a combination of factors gives the best results. These ratios can be tuned further for a given radius, dimension and pivot number, but we do not explore the matter in further detail here, given that using an equal combination policy seems to provide acceptable improvements.

5.2.2 Drop Rate

In the first phase, pivots are processed until everything is known about their inclusion or exclusion in the query range. If all the objects were also pivots as in AESA, this step would be sufficient to complete the query processing. However, there are some ordinary database objects that still need to be explored. At one extreme we have Kvp, which computes all the distances to pivots first. At the other extreme we can process the ordinary objects using only what we know about pivots from the first phase. This decision is controlled by a parameter called the *drop rate*.

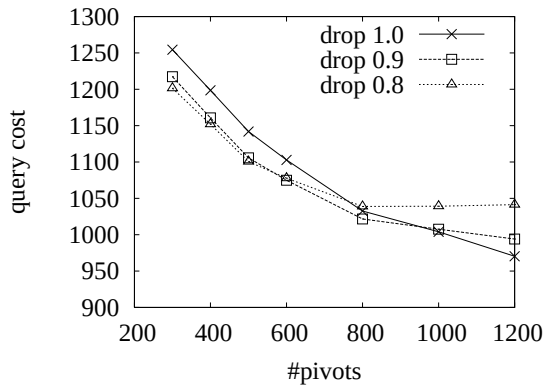


Figure 5.7: Query Costs for Varying Drop Rate values for a query radius of 0.9 in 10 dimensions

When the first phase is executed, we have a set of pivots PP that have their exact distance to query object calculated, and the rest of the pivots PR know their bounds with sufficient accuracy to be excluded or included in the query range. We continue processing the set PR to obtain better bounds for pivots, but this time we restrict this to a ratio of the cardinality of set PR. We drop a portion of PR and process the rest of the pivots in PR.

Earlier experiments showed that for a fixed number of database objects, as the number of pivots is increased, more of them should be dropped for better performance. The probability that a pivot will be useful for elimination of ordinary objects increases as there are more objects per pivot. Figure 5.7 compares the query performance of HKvp in 10 dimensions for a radius of 0.9 for drop rates 1.0, 0.9, and 0.8. Other settings yielded similar results, and are not presented here.

As we see, as the ratio of pivots decreases we need to make use of more pivots. The point where the performance of the drop rate value of 1.0 equals to that of

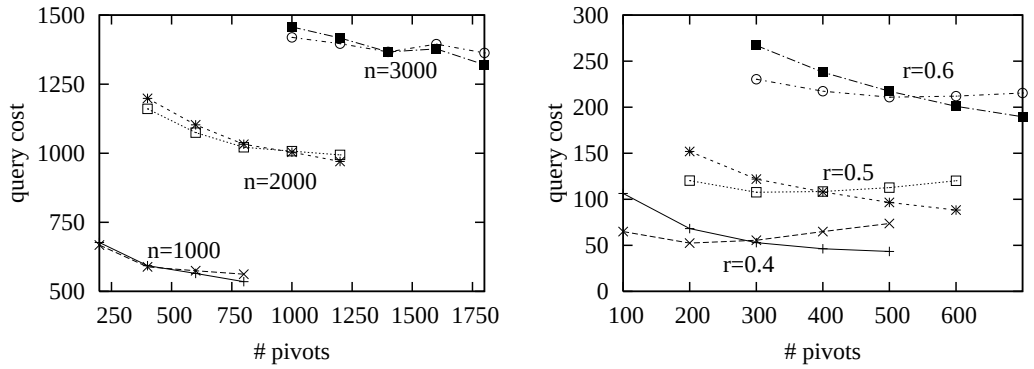


Figure 5.8: The crossing point between drop rates of 1.0 and 0.9 in 10 dimensions for various database sizes (left) and query radius values (right).

0.9 is particularly interesting, since this is when there is a substantial need to start adjusting the drop-rate parameter to use more pivots. We ran some experiments to see how the size of the database and query radius affects this critical point. Figure 5.8 summarizes our results. As expected, we see that larger databases need to start dropping pivots later than smaller ones. We also see that as we increase the search radius and therefore make the query more difficult, we need more pivots to process, therefore the point where we should stop eliminating all pivots goes toward that direction.

5.2.3 Indirect Elimination

One of the improvements of HKvp over its predecessors is that it uses pivots without a direct distance to the query object for elimination of ordinary database objects. Recall that after the pivot elimination step, we will have lower and upper bounds for distances of some pivots. We make use of these bounds to deduce distance

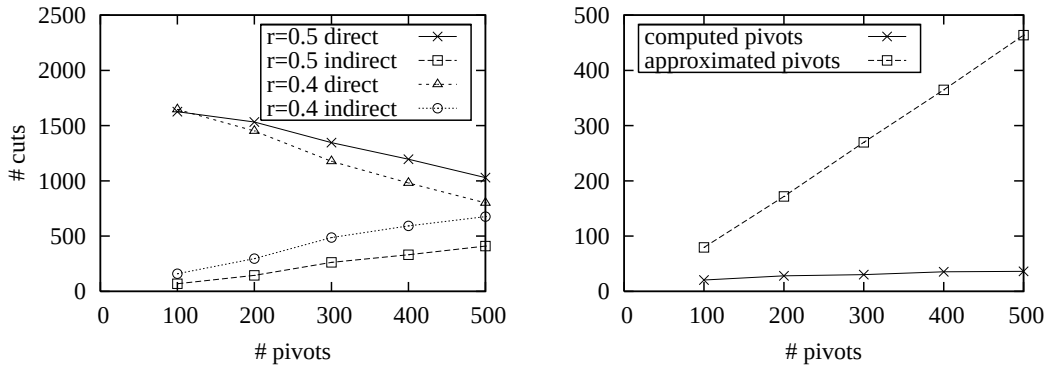


Figure 5.9: **(a)**Comparison of direct cuts and approximate cuts in HKvp (left). **(b)** The number of approximate and direct pivots for HKvp (right). For both graphs, the database consists of 2000 points uniformly distributed in 10 dimensions.

bounds from the ordinary objects to the query object.

In order to see the extent to which indirect elimination contributes to query performance, we ran some experiments where we counted the number of indirect cuts compared to regular cuts. Figure 5.9(a) summarizes the results for a 10-dimensional database with 2,000 objects. We see that approximate cuts provide a fair amount of object elimination. As the number of pivots grows, we see that approximate cuts are of greater value. Figure 5.9(b) provides an explanation. Here we plot the size of PP versus the size of PR. We see that the former number grows much slower, and the ratio of approximate pivots increases as we increase the total number of pivots.

Figure 5.10 compares indirect versus direct cuts for varying radius values using 500 pivots. We see that direct cuts are dominant for difficult queries since they provide greater certainty; however, as the queries become easier, approximate elimination becomes quite important.

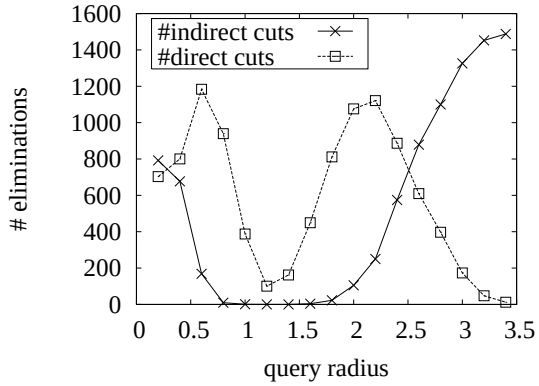


Figure 5.10: Breakdown of indirect and direct elimination of objects for varying query radius in 10 dimensions on a database with 500 pivots and 1500 ordinary objects.

5.2.4 Pivot Limit

The *pivot limit* is a parameter of Kvp that indicates the number of pivot distances stored and used per database object. As an extension of Kvp, HKvp can also adjust this parameter. Rather than storing the distances to all pivots, keeping only the most relevant pivots reduces both storage and processing times simultaneously. Figure 5.11(a) shows the result of adjusting the pivot limit on query times for a database of size 4,000 in 10 dimensions for different radius values. A standard drop rate of 1.0 was used in these experiments, which means that only the necessary pivots were computed against the query object. The total number of pivots was 500. We see that the pivot limit parameter is more helpful for lower radius values. For radii of 0.5 and 0.6, storing only 300 pivots produced results very close to that of using full 500 pivots.

Although the results look similar to typical Kvp experiments, there is one slight

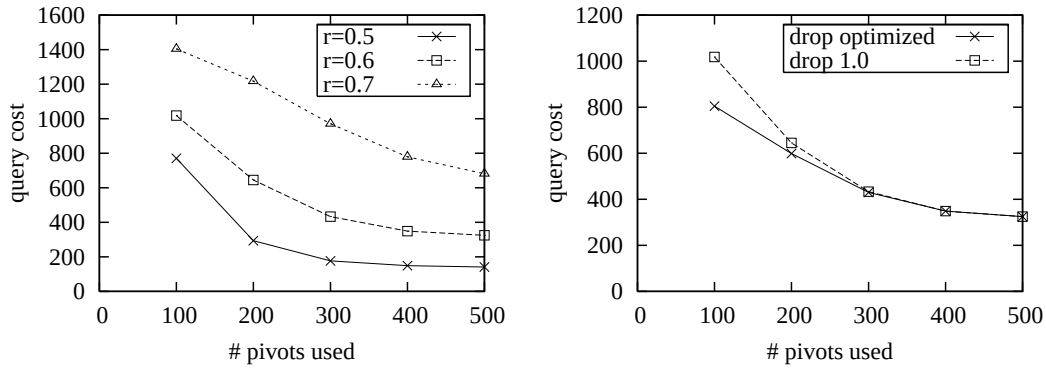


Figure 5.11: **(a)** Using different Pivot Limit parameters (left). **(b)** Optimizing the drop rate for better pivot limit utilization (right). The database size is 4000, The objects are uniformly distributed in 10 dimensions.

difference: the pivot limit does not necessarily reflect the actual number of pivots used per database object. Some of the pivots are not usable since they have been eliminated in the first phase of HKvp. By adjusting the drop-rate parameter, we can increase the number of pivots that are actually used, and the optimal drop-rate parameter will be different for different pivot-limit values.

Figure 5.11(b) explores the affect of the drop-rate parameter on the pivot-limit parameter. Varying drop rates with increments of 0.1 were applied and the best one was taken as the optimized result. We see that with a good choice of drop rate, we can decrease the performance penalty that results from limiting the number of pivots.

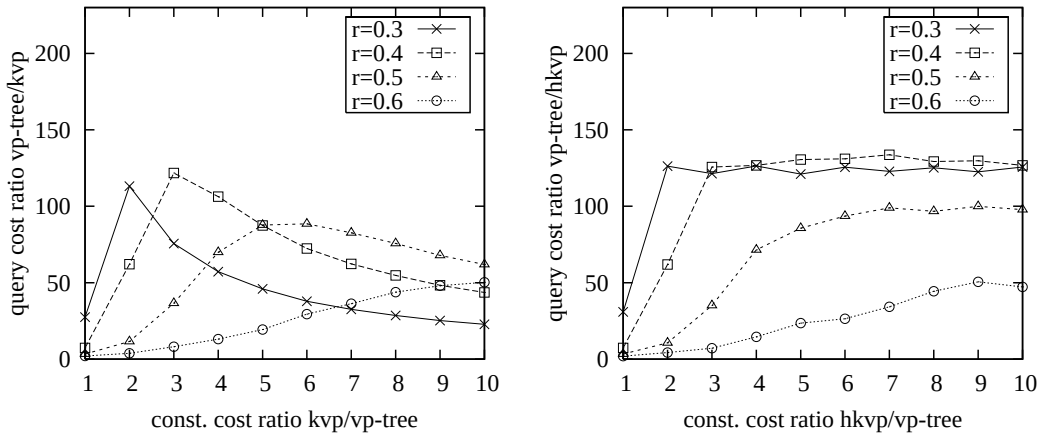


Figure 5.12: Ratio of the costs of vp-tree to Kvp (left) and vp-tree to HKvp (right) for various construction cost ratios on a database of 10,000 uniformly distributed random vectors in 40 dimensions.

5.3 Query Performance of HKvp

We mentioned earlier that the number of pivots in a pivot-based structure has an optimal value. Figure 5.12 compares the Kvp with the vp-tree for three different query radius values. We see that Kvp can perform up to 120 times better using 3 times the construction cost of the vp-tree on a database of 40-dimensional uniformly distributed vectors. Even though the improvement we obtain in terms of performance by increasing the construction cost is impressive, we see that the performance of Kvp begins to decline once the optimal number of pivots is reached. The optimal number of pivots increases as the query radius increases, because more pivots are needed for queries that are more difficult.

We compared the performance of HKvp versus Kvp to determine the performance gain due to its better use of pivots. Figure 5.13 summarizes our results on a

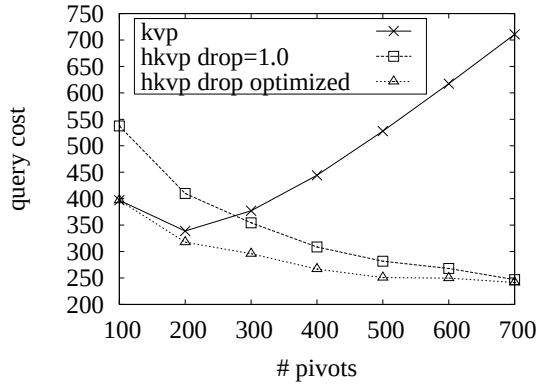


Figure 5.13: Comparison of HKvp with Kvp on a database of 1000 vectors.

database of 1,000 vectors using varying number of pivots. We see that HKvp with a drop-rate parameter of 1.0 improves on Kvp when the number of pivots is high, but it produces poorer results otherwise. The reason is that the pivot elimination process is targeted at minimizing the number of pivots being processed, but the unprocessed pivots decrease the likelihood of the elimination of ordinary objects. This shortcoming can be rectified by adjusting the drop-rate parameter according to the ratio of pivots to the database size. The figure also shows the result of HKvp where the drop-rate parameter is optimized. In this case it outperforms the two extreme cases in all our tests.

Figure 5.12 also shows a comparison of HKvp with the vp-tree. In this figure, Kvp was seen to perform worse as more pivots were used. We see that this shortcoming is eliminated in HKvp.

Our experiments that compared HKvp to LAESA showed that it performs up to 49% fewer distance computations. Some of our results are summarized in Figure 5.14. In all experiments, the same greedy algorithm was used to determine the

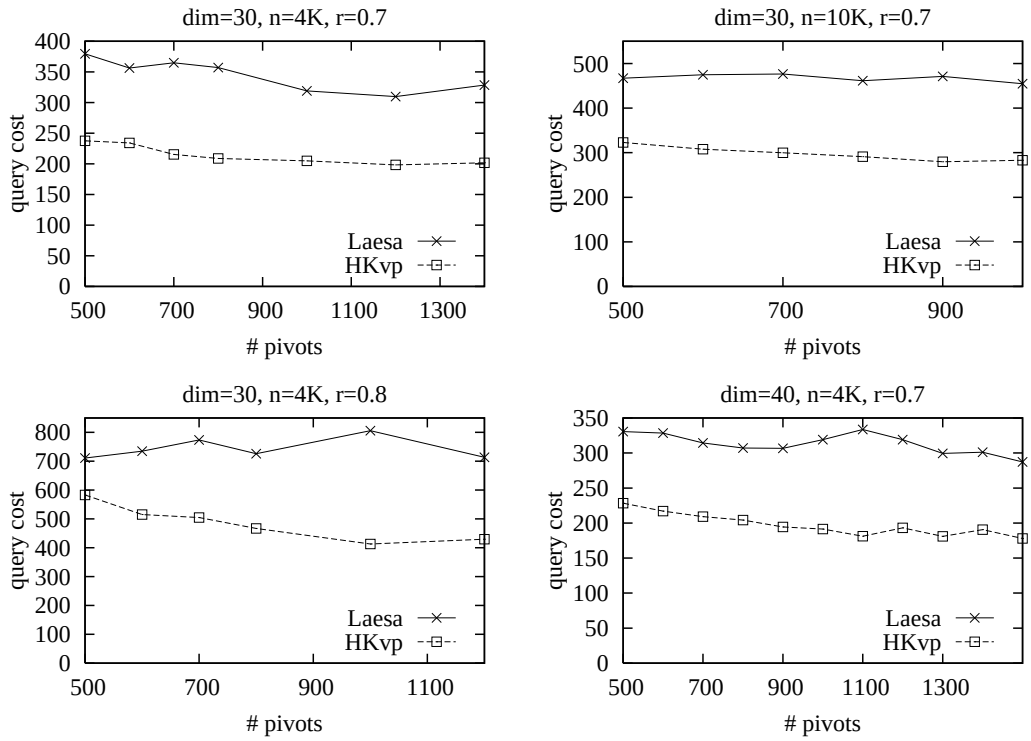


Figure 5.14: Hkvp compared to LAESA for various settings. The drop parameters were optimized for both structures.

optimal value for the drop-rate parameter for both structures. In order to obtain a better understanding of the query cost improvements, Figure 5.15 presents two of the settings in Figure 5.14 in terms of the ratio of the query costs. Observe that when the number of pivots is low, both structures work very similar to Kvp and use all the pivots available. This explains why HKvp does not provide improvements over LAESA for very low pivot number settings. Otherwise, we see that there is a consistent improvement over LAESA for cases the HKvp was designed for.

We chose a pivot selection policy of 1::1 for the experiments which yielded query costs that are up to 43% less than LAESA. Optimizing this policy provided up to 6% further improvements on the query cost.

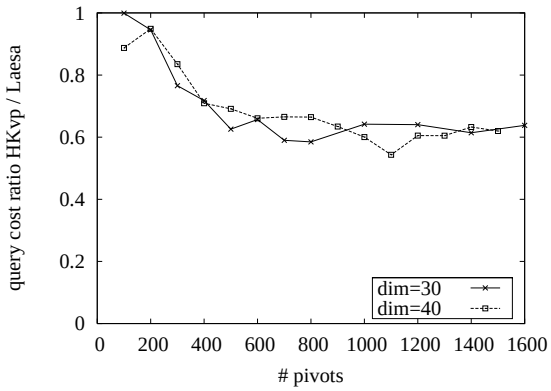


Figure 5.15: Ratio of the costs of LAESA to HKvp for query radius 0.7 on a database of 4,000 uniformly distributed random vectors.

5.4 Summary

The efficiency of a metric space indexing algorithm may be measured by a number of factors including space usage, construction cost, processing time and the number of distance computations to process a query. In this chapter, we focused our attention on the effect of construction cost and the number of distance computations. We evaluated three major structures from this point of view, and showed that the performance of these structures is highly correlated with their construction times. Our experiments clearly show that global pivot-based methods offer a better performance to construction cost ratio.

We also showed that the optimal number of pivots varies with the choice of method and query radius. Easier queries, that is, queries executed on databases having lower dimensions or queries having small query radius values, tend to require fewer pivots than more difficulty queries. A pivot-based method may perform worse even when given more setup time because it has more pivots to process than needed.

Although it lacks some of the features of its modern descendants, LAESA has the ability to eliminate pivots as well as ordinary database objects, and produces better results when the ratio of pivots to the number of ordinary database objects is high.

We also introduced a new structure, called HKvp, which performs significantly better than LAESA while providing greater flexibility. HKvp achieves this flexibility by trading off query performance (in terms of distance computations) against lower overheads in space and processing times.

The major drawback of HKvp is the need to adjust the drop-rate parameter to produce the best results. A possible solution is to modify the HKvp search to try different drop-rate values and train itself on which drop rates work best for different query radius values. This is purely a query-time adjustment and varying the drop rate would not result in any structural changes on HKvp.

Chapter 6

The EcKvp Structure

In this chapter we introduce a new structure, called *EcKvp*, which is based on the Kvp structure. It shares with Kvp the advantages of low space and query time, but with only a small performance penalty, it offers significantly lower preprocessing time. It achieves lower construction costs by storing the pivots in an index structure, and retrieving distance values between objects and pivots by querying into this structure rather than computing all the distances between them.

6.1 Introduction

In order to better understand how EcKvp works, let us consider a typical global pivot-based structure. Given a set of objects O, we first choose P as global pivots. For each object o, we compute the distances to all the pivots in P and create a *pivot distance vector* PD_o containing these distances. Thus, the index structure is basically a distance matrix between P and $O - P$. Let us consider a range query centered about a query object q with radius r. Recall that our basic strategy is to *eliminate* database objects from consideration without explicitly computing the distance between the database object and the query object. We first compute the distances between q and all the pivots in P. For each object o, we use the information in PD_o to attempt to avoid computing the actual distance between o and q. Let E

denote the set of such eliminated objects. Given that PD_{op} contains the distance between each pivot p and each non-pivot o, it follows from the triangle inequality that o can be eliminated if its distance to p is less than $d(q,p) - r$ or if its distance to p is greater than $d(q,p) + r$. That is,

$$o \in E \Leftrightarrow \exists p \in P((PD_{\text{op}} < d(q,p) - r) \vee (PD_{\text{op}} > d(q,p) + r)) \tag{6.1}$$

Observe that this test does not require that any distances be computed at query time other than between pivots p and thee query object q. Only when we fail to eliminate an object based on all the stored pivot distances in PD_o will we compute the actual distance between o and q to determine whether $d(q,o) < r$. Thus, the total cost of the query in terms of the number of distance computations performed is $|O - E|$.

Variants of this basic idea differ in the way they store the distance information and the way they process that information to eliminate objects. For instance, Kvp purges the less effective distances in PD_o to avoid storing and processing the complete distance matrix.

The EcKvp structure introduces a new method for creating the pivot distance vector. It makes use of an internal index containing only pivots to acquire this information. We call this structure the *pivot index* or the *inner index*. The pivots of the global structure will be regarded as ordinary database objects inside the inner index. The database objects of the global structure will be used as query objects at construction time. By running a query on the inner index we will discover distance

relationships between pivots and regular objects. We call this an *inner query*.

The structure to use as the inner index and the type of the inner query to be executed inside the inner index are the parameters of the EcKvp structure. Because of the special needs of the EcKvp construction process, it is important to augment the information returned by similarity search queries applied to the inner index. Typical similarity search queries, such as range and k-nearest neighbor queries only identify which database objects meet the search criteria; it is not a requirement to return actual distances to the query object. But this distance information will be of value to the EcKvp search algorithm. Also, in the process of answering the similarity queries we may discover useful information about the distance relationships for objects that are not in the result set. This information is discarded when the search is completed, but may be of value to the EcKvp search.

Let us consider how to augment information returned from a query for the purposes of building the EcKvp structure. Assume that database object o is being inserted into an EcKvp structure, and as a part of the insertion, we run a query on the inner index to determine PD_o. Let us assume that the inner index is a global pivot-based structure. When answering the query on o, the search algorithm seeks to eliminate objects of the inner index. Let us say that in the middle of the query algorithm, we are processing the pivot pi of the inner index on the object p to decide whether p should be in the result set. Recall that p can be an ordinary object in the context of the inner structure, but it is one of the pivots of the EcKvp structure. Let D_o denote the distance between pi and o, D_p denote the distance between pi and p, and r denote the query radius. Letting D_{op} denote the distance between p

and o, we can conclude that:

$$|D_o - D_p| < D_{op} < D_o + D_p \tag{6.2}$$

In this way, we can determine upper and lower bounds for the value of D_{op}. We call this the *pivot bound* of object o to pivot p, or PB_{op}, and the set of all pivot bounds for object o are denoted by PB_o. Thus, Equation (6.1), which describes the cases when an object can be eliminated, can be expressed as:

$$o \in E \Leftrightarrow (PB_{op}.upperBound < d(q,p) - r) \vee (PB_{op}.lowerBound > d(q,p) + r) \tag{6.3}$$

If the pivot bound PB_{op} is strong enough to prove that o is either inside the query range or outside the query range, we stop the processing of o, and otherwise we move on to process other pivots. Each pivot we process will tend to improve PB_{op}, and if none of the pivots are strong enough to eliminate o, we compute the actual distance between o and p. Thus, although this process is oriented towards answering the range search with as few distance computations as possible, it produces some very useful information in the form of PB_o as a byproduct.

Recall that the typical range query discards information about objects that lie outside of the query radius. Our special query returns all the pivot bounds produced. Another modification we make is to let the query process look at all the pivots of the inner index, even if the object can be proven to be inside or outside of the result set. This way we can further strengthen the pivot bounds without incurring additional distance computation costs during preprocessing.

The query radius used for the pivot index, which we will call *inner query radius*, is a parameter. Selecting a small inner query radius value will result in low cost query executions, but the distance information it provides will also be limited. Therefore, there is a trade-off to consider in the choice of the radius parameter, which we will explore later in the chapter.

In summary, the EcKvp structure consists of an internal pivot index and PB, the set of all pivot bounds for all the objects in the database. The construction algorithm is given in Algorithm 1.The algorithm for range search is very similar to that of Kvp. The only difference is we need to take into account that we may not have exact distance of objects to pivots. The algorithm is given in Algorithm 2.

The algorithm for producing PB_o is independent of the EcKvp structure. Algorithm 3 presents the algorithm for a global pivot-based inner index PI using a modification of the range search.

6.2 The Pivot Index

The performance of the pivot index is crucial for the efficiency of the EcKvp construction. If EcKvp were to use sequential scan for the inner queries, effectively computing the distances between each database object and all pivots, it would perform exactly the same amount of work as the Kvp structure. Therefore, any index structure that works better than sequential scan will provide us with improvements on the overall construction cost. The exact amount of improvement can be quantified as follows. Given a query range, define its *cost ratio* to be the ratio of the average

Algorithm 1 EcKvp Construction

Input O, set of database objects

Output PB, the pivot bounds that make up the index.

$PB \leftarrow \{\}$

select $P \subset O$ as pivots.

construct the inner structure PI as the pivot index consisting of objects in P

for all $o \in O - P$ do

execute a query on PI using o as the query object and produce PB_o

remove the unpromising entries in PB_o

sort PB_o so that entries that are more promising appear in front

$PB \leftarrow PB \cup PB_o$

Algorithm 2 EcKvp Query

Input EcKvp structure E, query object q, query radius r.

Output RS, the set of objects within query radius.

$RS \leftarrow \{\}$

for all pivots $p \in E$ do

    Compute $d(p, q)$

for all objects $o \in E$ do

    for all $PB_{op} \in PB_o$ do

        if $r > d(p, q) + PB_{op}.upperBound$

            $RS \leftarrow RS \cup o$

            move on to the next object

        if $[d(p, q) - r, d(p, q) + r]$ and PB_{op} does not intersect

            move on to the next object

    Compute D_{oq} as $d(o, q)$

    if $D_{oq} < r$

        $RS \leftarrow RS \cup o$

Algorithm 3 Algorithm for constructing PB_o

Input Owner of the entry o, pivot index PI

Output PB_o

$PB_o \leftarrow \{\}$

for each $p \in PI$ do

$PB_{op} \leftarrow < -\infty, +\infty >$

for each pivot $pi \in PI$ do

if $|d(pi, o) - d(pi, p)| > PB_{op}.lowerBound$

$PB_{op}.lowerBound \leftarrow |d(pi, o) - d(pi, p)|$

if $d(pi, o) + d(pi, p) < PB_{op}.upperBound$

$PB_{op}.upperBound \leftarrow d(pi, o) + d(pi, p)$

$PB_o \leftarrow PB_o \cup PB_{op}$

query cost in terms of the number of distance computations for this range to the size of the index.

In this chapter, we will make the reasonable assumption that the number of pivots is significantly smaller than the size of the database, and therefore the construction cost of the pivot index will be a negligible component of the overall preprocessing costs. This means the inner index will consist of a small database of objects, and so we are willing to spend a significant amount of time in constructing the inner index in order to speed up query processing. Given these requirements, we will show that the HKvp structure is an excellent choice for the pivot index. The experiments we will perform on HKvp will be based on uniformly distributed vector spaces.

6.2.1 The HKvp Index Structure

As introduced in Chapter 5, HKvp is a variant of the Kvp structure that has the advantage of eliminating pivots as well as database objects during a search. In this way, HKvp can avoid computing the distances to all the pivots. In order to eliminate pivots based on the distance of the query to another pivot, HKvp stores the distances between pivots. Therefore, the construction cost and space of the HKvp structure grows quadratically with the number of pivots.

Recall that the HKvp query processing starts with the processing of the pivots. First, it computes the distance of the query object to some of the pivots, and based on these distances, it computes bounds on the distances to the rest of the pivots.

The important thing here is, at the end of this phase, the bounds of a pivot are good enough to infer whether it is inside or outside the search radius. In the second phase it processes the non-pivot database objects based on the pivot bounds computed in the first phase.

The first phase works in an incremental fashion. At each step we find the most promising pivot based on some policy, compute the distance from this pivot to the query object, and update the bounds of other unprocessed pivots based on this new information. The process continues until all the pivot bounds are sufficient as described above.

6.2.2 Cost of the Inner Queries

The number of the pivots, which determines the size of the pivot index, and the inner query radius, are both parameters used in the construction of the EcKvp structure that will influence the cost of the inner queries. Let n be the size of the database and p be the number of pivots selected out of these n objects. As mentioned earlier, we assume that p is significantly smaller than n. Since our inner index is an HKvp structure in which all objects are pivots, the construction of the pivot index will involve $O(p^2)$ distance computations. On the other hand, assuming that HKvp provides sub-linear query performance, the cost ratio of HKvp will improve as we increase p. Figure 6.1(a) supports this observation, showing that the cost ratio sharply improves as the size of pivot index grows.

Keeping in mind that the improvement in preprocessing time for EcKvp over

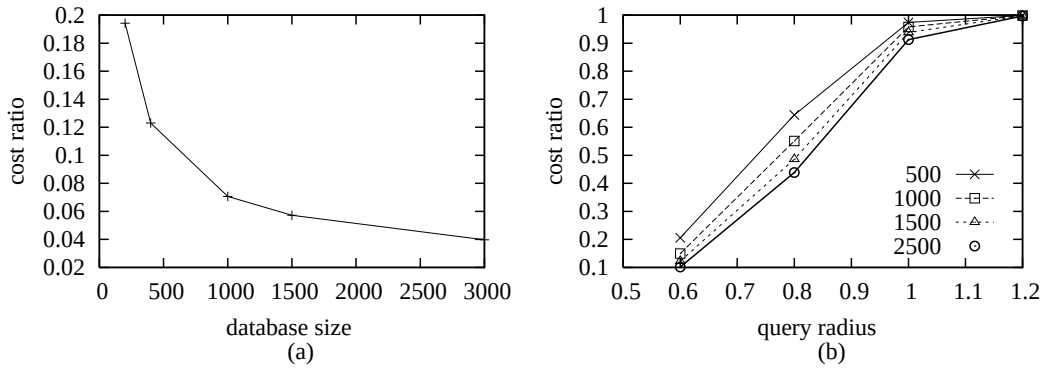


Figure 6.1: Cost ratio for various settings in 10 dimensions. (a) cost ratio by database size for a fixed query radius of 0.5. (b) various database sizes by the query ratio.

Kvp is related to the cost ratio, for the setting in Figure 6.1 we see that an improvement of 80% is possible with as few as 200 pivots. The improvements are more dramatic at the lower portion of the graph, and for this particular setting, a pivot pool size of 1000 to 1500 seems to offer a good compromise. Figure 6.1 (b) shows that the results are consistent under different inner query radius values.

6.2.3 Information Received from the Inner Queries

We have seen that increasing the size of the inner index tends to improve the cost ratio, while increasing the query radius tends to make the cost ratio worse. Although our results show that improvements in query times are possible, they do not provide a clear picture of how query performance depends on the partial information obtained from the inner queries. Query performance will depend on the quantity and quality of the information received from the inner queries.

One way to measure the quantity of information received from the inner queries

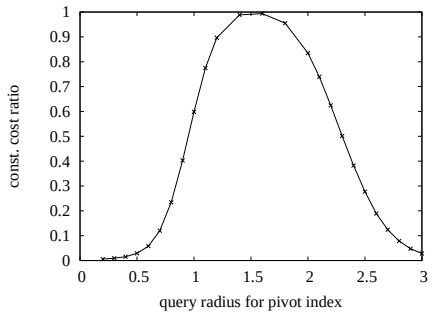


Figure 6.2: Number of exact distances by inner query radius for a pivot pool of size 1500 uniformly distributed in 15-dimensional space

is to consider the number of exact bounds, where the lower and upper bounds are equal to each other, and hence they provide the exact distance between the object and the pivot. For any index structure, this is also equal to the cost of the query in terms of the number of distance computations. Therefore, while it is true that the cost of the query is expected to grow sub-linearly with the size of the inner index, the amount of exact information also follows the same pattern.

Figure 6.2 shows the amount of the information viewed from this respect. We see that the amount of information grows rapidly as we increase the inner query radius, and that it is possible to pick a radius value to yield any desired inner query cost.

The number of exact distances is related to only the quantity of the information, but not the "quality" of this information. This is because pivots have varying degrees of success in their ability to prune database objects from consideration. Specifically, we know that pivots are most valuable when they are either close to or far from the query object [11]. To investigate this issue, we have run experiments

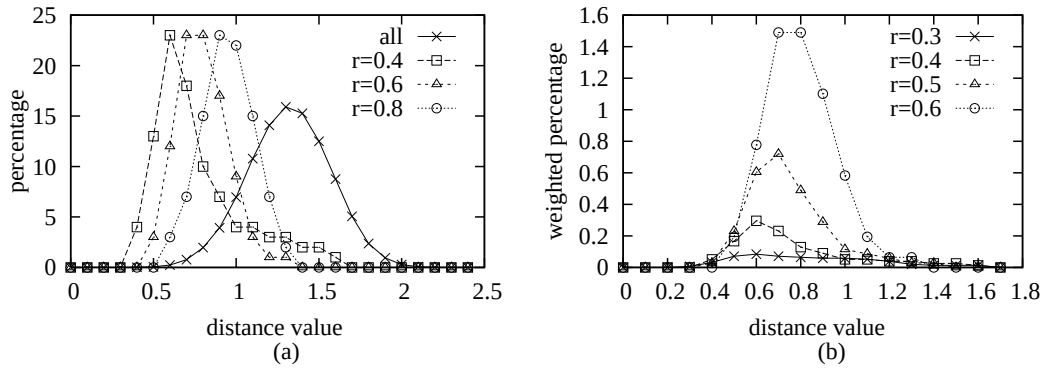


Figure 6.3: Distribution of the distance information for various inner query radii. (right) disribution weighted by the size of population

to analyze the distribution of exact distances between query objects and pivots for various inner query radius values. Figure 6.3 summarizes our results. We see that smaller radius values produce higher quality distance information, since the pivot distances are closer to the object. This behavior is symmetric, which means that as the inner query radius values approach the other end of distance spectrum we see that more restrictive queries produce information with more favorable distributions.

The discussion about the set of pivots for which we have discovered the exact distances is not sufficient to give us the whole picture. We need to somehow incorporate the rest of the pivot bounds that are not exact. We define the *efficiency* of a pivot bound PB_{op} with respect to a query radius r to be:

$$Ef(PB_{op}, r) = \text{probability that } o \text{ will be eliminated by } p \text{ for a radius of } r.$$

Given F(), the cumulative probability distribution function, we can approximate the efficiency using the following equation.

$$Ef(PB_{op}, r) \approx F(PB_{op}.lowerBound - r) + (1 - F(PB_{op}.lowerBound + r)) \quad (6.4)$$

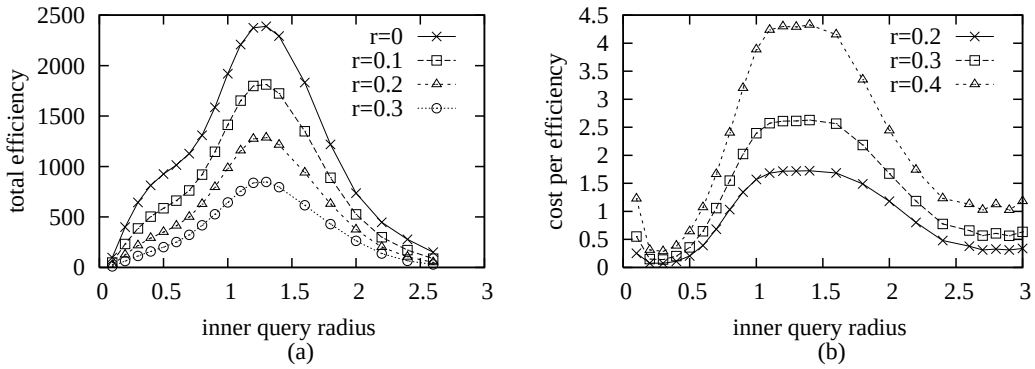


Figure 6.4: (a)Total efficiency versus the inner query radius. (b) Cost per efficiency versus the inner query radius

Note that Equation (6.4) is directly derived from Equation (6.3), and corresponds to the area which represents objects that can be eliminated by p.

When we run a query for object o on the inner index, we will get a set of pivot bounds, PB_o. We define the *total efficiency* of PB_o with respect to a query radius r to be:

$$Ef(PB_o, r) = \sum_{PBop \in PB_o} Ef(PB_{op}, r) \tag{6.5}$$

Figure 6.4 (a) shows the total efficiency values achieved through different inner query radius values. We see that the total efficiency values show similar trends for different difficulty levels. This shows that the choice of the EcKvp parameters does not depend on the query radius, which may vary and is only available at query time.

At this point, it is natural to ask how the cost per efficiency changes as a function of the inner query radius. Figure 6.4 (b) shows that there is an optimal radius value from this respect. Figure 6.4 presents evidence as to why EcKvp has

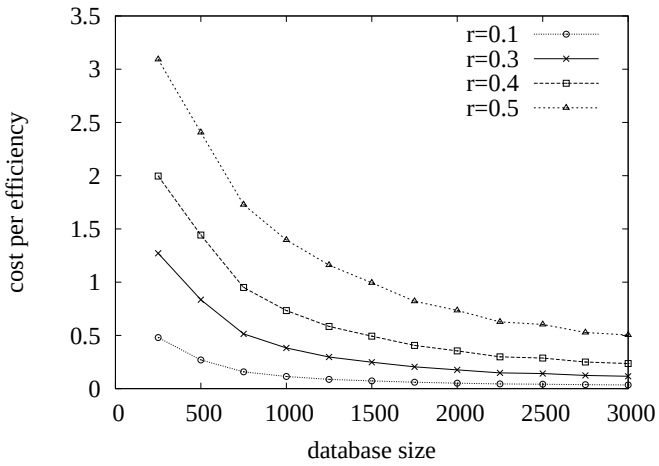


Figure 6.5: Cost per unit efficiency for various query radius values by the inner index size for an inner query radius of 0.3 in 10 dimensions.

a great potential for improving the preprocessing costs. The inner query radius value where information is most expensive is also the point where the query is most difficult. This is exactly the path taken by other global pivot-based methods. In contrast, EcKvp can generate information with significantly lower computational cost.

We know that the cost for generating exact pivot distances for varying inner index sizes is constant; each distance computation produces one exact distance. We performed experiments to determine how the total efficiency output is affected by the inner index size. Figure 6.5 shows this relationship for a constant inner query radius value of 0.2.

We see that larger index sizes produce information more efficiently than smaller ones. The more difficult the queries, the wider the gap between performance of different inner index sizes.

6.2.4 HKvp Parameters

The HKvp structure has a number of parameters whose values can be chosen to optimize performance for the pivot index. The *pivot limit* parameter controls how many pivot distances per object are stored in the database. In our experiments, we stored all possible distances, but the pivot limit value can be used to reduce space and CPU requirements of our pivot index. Another parameter is the *pivot selection policy*, which determines the order in which pivots are processed. Although several alternatives exist, we have not pursued optimizing this parameter in our experiments. As the default ratio, we used 1::1 as explained in Chapter 5.

Yet another parameter is the number of pivots to use. Even though the EcKvp parent structure may have specified the number of pivots to keep in the pivot index, the HKvp does not have to appoint all of them as pivots in its own scope. Figure 6.6 shows query results for various choices of the number of pivots. The database size was 2500 and the number of pivots was varied, so that if we have k pivots we would have 2500 - k ordinary database objects. We see that in this example the performance tend to level off after assigning 1500 pivots. Using fewer pivots for our pivot index improves the construction cost of the pivot index considerably, but at the price of slightly poorer query performance.

6.3 Performance of EcKvp

In this section, we compare the performance of EcKvp against the HKvp structure. To simplify the presentation, in our comparisons we will ignore the space

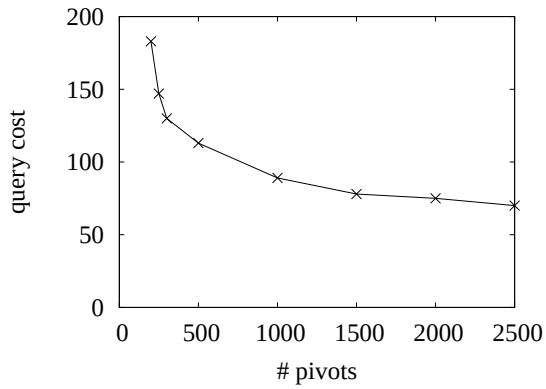


Figure 6.6: Query cost versus the number of pivots for the inner index in 10 dimensions for a query radius of 0.5.

usage and focus instead on construction costs. As we have mentioned, EcKvp uses the same space-saving techniques as Kvp.

We first compare HKvp with two well-known methods from the other two families of solutions as outlined in Chapter 2. We chose GNAT from the clustering methods family since it is a well known structure with established performance. We chose the vp-tree from the local pivot-base methods family because of its simplicity. The best performer here, the mvp-tree, was reported to make up to 80% fewer distance computations, using twice the construction cost of vp-tree. However, it is a hybrid structure since it keeps some extra distance information at the leaves. We generated several instances of vp-trees and GNAT trees by varying their branching factors. Figure 6.7 summarizes our results for one particular setting. We observe that global pivot-based methods, represented by Kvp, are far superior to others.

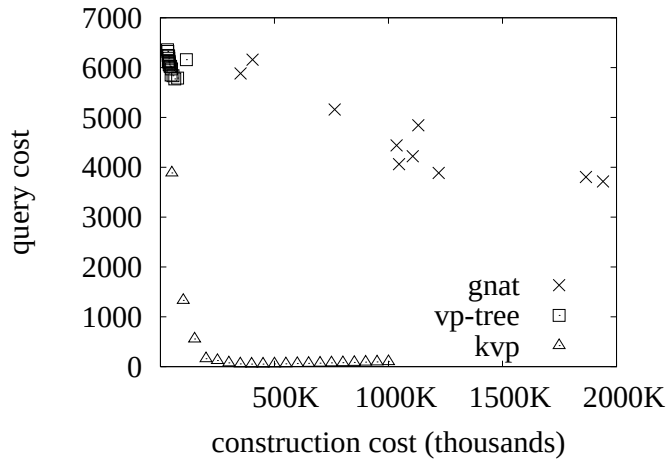


Figure 6.7: Comparison of query performances of Kvp, vp-tree and GNAT versus their construction costs in 40 dimensions for a database of size 10000.

6.3.1 The Construction Cost Ratio

Here we outline how we compare the construction costs of the EcKvp and HKvp. The preprocessing performance of EcKvp is sensitive to the size of the database. We will attempt to quantify performance through a statistic called the *construction cost ratio*, which is independent of the database size. Given a database of n objects, and choosing p of the objects as pivots, the total construction cost of EcKvp using an inner query radius of r to construct an inner index PI is as follows.

$$QC(PI, r) \cdot (n - p) + \frac{p^2}{2}$$

where $QC(PI, r)$ is the average cost of queries run against PI using radius r. The total construction cost of Kvp is given by:

$$p \cdot n - \frac{p^2}{2}$$

The ratio between these two costs is:

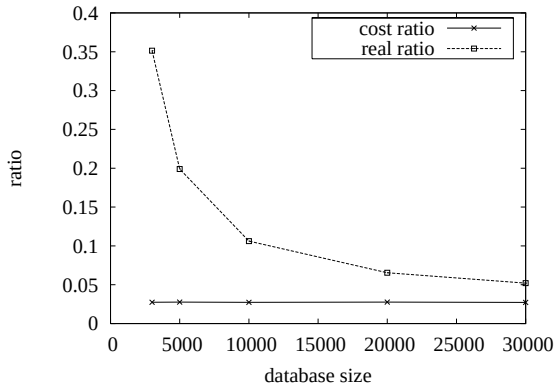


Figure 6.8: The real construction cost ratio versus projected ratio for varying database sizes and for a pivot pool of size 1500 in 20 dimensions.

$$\frac{p}{2n-p} + \frac{QC(PI,r)}{p}$$

Assuming n is far larger than p, the first term of the above sum is negligible relative to the second term. The second term is equal to what we have defined earlier as *cost ratio*. In the context of EcKvp, we will also call this the *construction cost ratio* to be more specific. From now on, we will use this term to compare the construction times for EcKvp to HKvp.

Figure 6.8 provides some justification for our decision to use the construction cost ratio as the main criterion of the evaluation of the preprocessing performance of EcKvp. We see that as the database size grows, the actual ratio of the construction costs of EcKvp and HKvp quickly decreases to a value that is comparable to the construction cost ratio. As expected, the construction cost ratio remains around the same value for different database sizes. This allows us to run our experiments for modest database sizes, yet accurately predict the cost ratios for significantly large

databases.

6.3.2 The Construction Cost and Performance Trade-off

We have mentioned that executing expensive inner queries provides us with more distance data, thus improving the query performance of the EcKvp. On the other hand, expensive inner queries increase the construction cost of EcKvp. In this subsection we will show examples of various settings where we explore the trade-off between query performance and construction cost. Throughout we compare the query performance of these two structures as the ratio of the query times of EcKvp to HKvp, called the *query performance ratio*. In theory, the HKvp structure should never perform worse than EcKvp, implying that this ratio should never be smaller than 1, but due to minor variations in the order in which pivots are processed, we sometimes observed ratios that are slightly smaller than 1.

One way to generate EcKvp structures with various construction cost ratios is to vary the inner query radius values. We ran experiments to see how varying the inner query radius influences the construction cost ratio and query performance. Figure 6.9 offers a summary for a particular setting for two different dimensions.

One point worth mentioning here is that for a given dimension, there seems to be actually two performance trends. The slightly worse trend is produced by using high inner query radius values. Figure 6.4 suggests that high radius values produce less efficient pivot bounds. Given that increasing the inner query radius rapidly increases the inner query costs, one idea that we experimented with was to run two

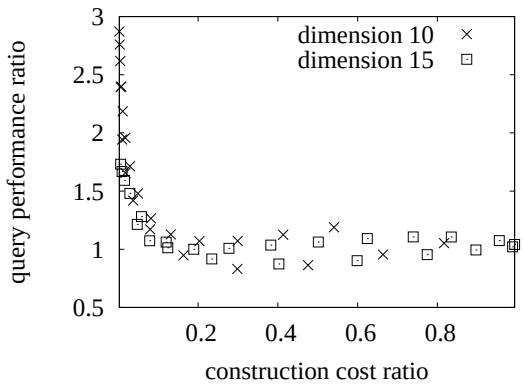


Figure 6.9: EcKvp performance for varying inner query radius values using a pivot pool of 1500.

queries, one close and one far, and then combine their outcomes. Figure 6.3 suggests that the distribution of these two inner queries are expected to be very different, so that the pivot information we obtain from these two separate relatively inexpensive queries should have very little in common, so that the extra query provides some new distance information. Unfortunately, our experiments with this approach did not yield a significant improvement.

In practice, determining a suitable inner query radius is not a trivial design issue. Because of this we implemented a revised version of the EcKvp structure that is given a target construction cost ratio, and it dynamically changes the inner query radius in order to reach this target value. Figure 6.10 shows the query performance of this implementation for various settings.

So far, all of our experiments have involved uniformly distributed data sets. Next, we performed a series of experiments to investigate how EcKvp performs for clustered data sets. We generated synthetic data by partitioning vectors into clusters

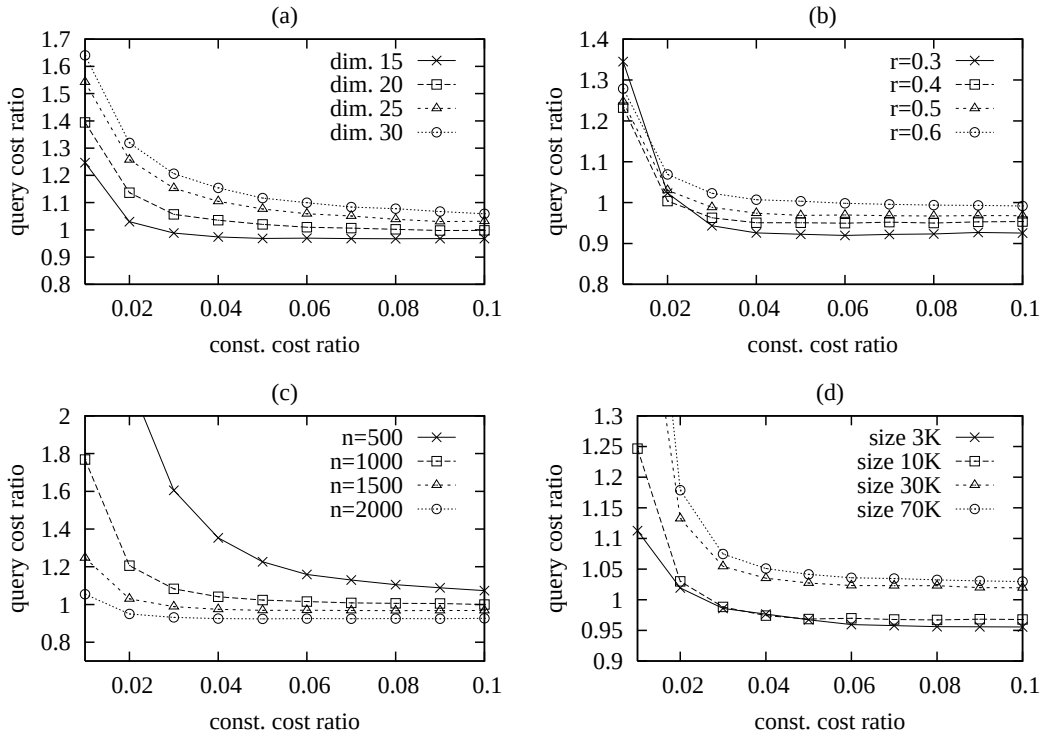


Figure 6.10: The construction cost and query performance ratios of EcKvp to HKvp in terms of the construction cost ratio and query performance ratio for different values of (a) dimensionality, (b) query radius, (c) number of pivots, (d) database size. In the experiments, unless specified otherwise, the database size is 3000, the objects are uniformly distributed random vectors in 15 dimensions, the query radius is 0.5, and the number of pivots is 1500.

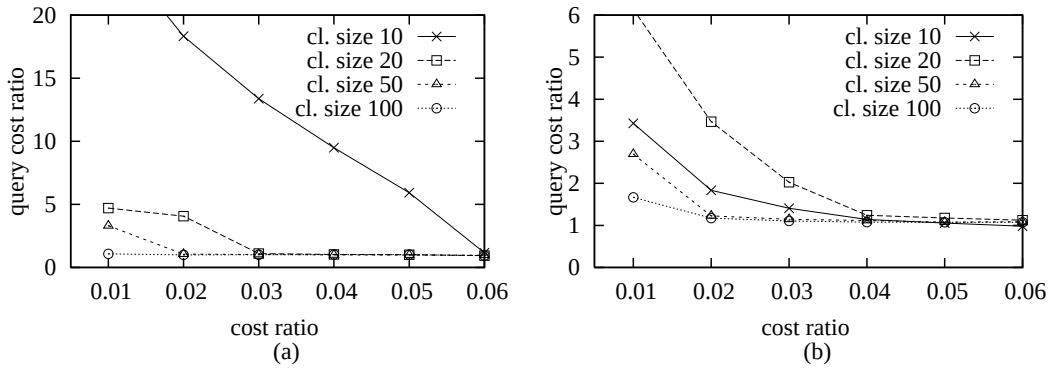


Figure 6.11: Comparison of EcKvp to HKvp for varying number of clusters for a database of 3000 vectors uniformly distributed in 15 dimensions for a cluster standard deviation of (a) 0.1 and (b) 0.2.

whose centers are distributed uniformly within a unit hypercube, and within each cluster the coordinates are drawn from a Gaussian distribution having the cluster center as mean and using a specified standard deviation. Our results are summarized in Figure 6.11. We see that as the cost ratio increases the performance of the EcKvp structure approaches the performance of HKvp, but with a construction cost of only 6% that of HKvp. In comparing Figures 6.11(a) and 6.11(b) we see that as the cluster standard deviations decrease, and the number of clusters increase, higher cost ratios are needed to match HKvp's performance. One explanation is that with clustered data finding a pivot within the same cluster as the query object provides a pivot that is very close to the query point, and by the observations made earlier about close pivots, such a pivot will have very strong discrimination power. By using all the pivots available, HKvp is more likely to find such powerful pivots.

6.3.3 Comparing EcKvp with HKvp for Similar Construction Costs

Up to this point we have viewed EcKvp as a way of achieving lower construction cost to provide comparable query performance as HKvp. Another way to look at EcKvp is that it should provide better query performance when given the same construction cost as HKvp. We ran a number of experiments to investigate this point of view. We created an EcKvp structure with a particular construction cost ratio, and then compared it with an HKvp structure that uses the same number of distance computations per regular database object. That is, given an EcKvp with p pivots and using a construction cost ratio of cr, the HKvp structure with which we compared it would use $p \cdot cr$ pivots. Note that this may not be a fair comparison when the database size is very small; the query performance will be dominated by the elimination of pivots, which in general is handled more efficiently than regular objects. Because of this, we ran our experiments for increasing values of the number of database objects to maintain control of this side effect. Figure 6.12 summarizes our results. We see that EcKvp can reduce query times by up to 70% compared to HKvp.

An important parameter of EcKvp is the number of pivots. We argued before that, in general, having a greater pool of pivots increases the efficiency of inner queries. We also ignored the construction cost of the pivot index, since the number of pivots is typically much smaller than the database size. If this cost is incorporated into the total construction cost, then the greater the pivot pool size is, the less we will be able to spend for inner queries. We ran experiments where we fixed the total

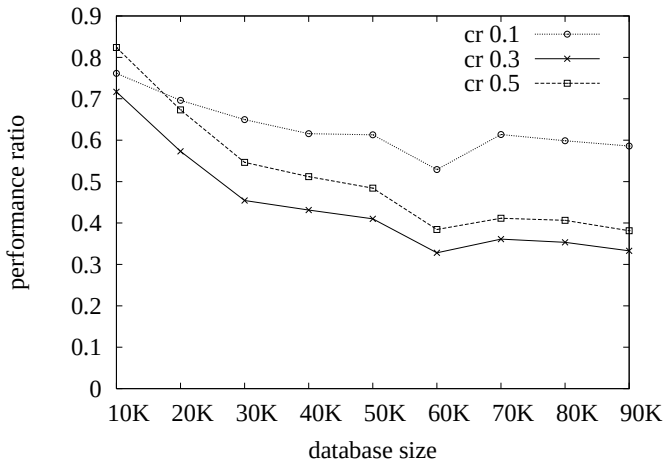


Figure 6.12: Comparison of EcKvp with an HKvp structure that uses the same construction cost ratio (cr). The pivot pool size is 2000, the query radius is 0.8, and objects are drawn from a uniform distribution in 20-dimensional space.

construction cost, including the inner index construction, and varied the number of pivots. We also compared our results with that of an HKvp structure that has the same construction cost. Let the total construction cost permitted be denoted by *ccost*, the database size be n, and the number of pivots be p. We can compute the number of pivots needed to achieve this construction cost by solving the quadratic equation:

$$ccost = \frac{p^2}{2} + (n - p) \cdot p$$

Our results are presented in Figure 6.13. We see that EcKvp can provide up to 60% improvement over HKvp in the query performance.

So far we have used only synthetically generated pseudo-random data to test our structures. In order to validate our results on real metric data, we have obtained a set of web-page data obtained through google [2]. We employed cosine distance [38]

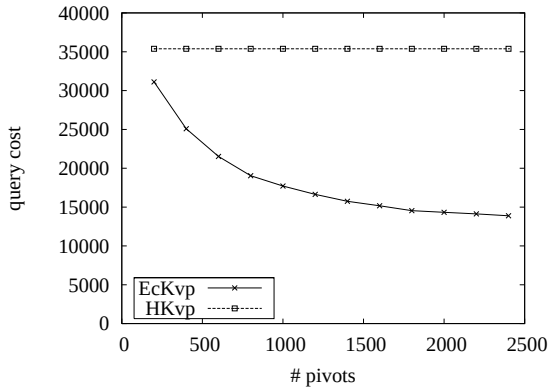


Figure 6.13: Comparison of EcKvp with an HKvp that uses the same total construction cost. The database size is 100,000, the query radius is 0.8, and objects are drawn from a uniform distribution in 20-dimensional space.

to the term vectors to carry out our range queries. Figure 3.6 shows the distribution of this data.

In a similar fashion to the way we generated Figure 6.7, we generated several different configurations of the GNAT tree and the Vp-tree by varying their branching factors. We also varied the total construction cost of HKvp and EcKvp for the same reason. Our results are summarized in Figure 6.14. We see that the results are similar to our earlier experiments. EcKvp provides performance improvement of up to 42% compared to HKvp.

6.4 Conclusions

In this chapter we have introduced a new data structure for similarity searching, called EcKvp, which retains all the positive features of the Kvp and HKvp structures but with significantly lower preprocessing times. We have shown that

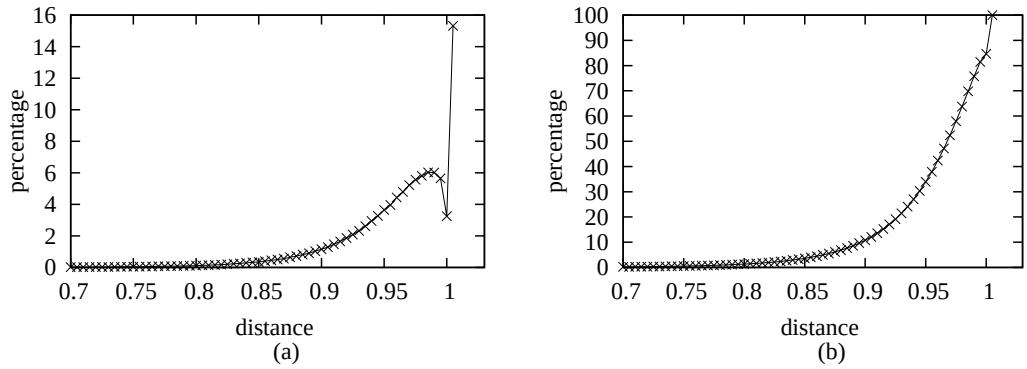


Figure 6.14: Comparison of the index structures using the web page data of size 1000.

EcKvp can also perform better than HKvp for same construction cost because it will be able to use more pivots.

EcKvp has a number of weaknesses. Although we are advised to use as many pivots as possible, when the size of the database is low, the optimal pivot pool size will be smaller than expected. We have shown that there seems to be an optimal inner query radius from the cost per efficiency point of view. This might mean that sometimes when the number of pivots is increased, the EcKvp structure might perform worse since it uses a suboptimal inner query radius. In our experiments this has not been an issue, but dealing with this issue in a more rigorous way is an interesting problem for future research.

In an attempt to better understand the EcKvp structure, one can draw parallels to tree structures. From many respects, Kvp can be considered as a bottom-up version of a tree-like structure. Tree structures require less space than do flat global pivot-based structures because pivots have local scope, and they only carry information about the objects within their own subtree. The Kvp structure eliminates

useless pivot bounds, and so in a way, a given pivot has a sort of "local scope" on objects that have found it to be useful. Tree-based structures attempt to cluster relevant objects together, so that pivots are used for similar objects. The objects that are in the same subtree as a given pivot have landed there based on a number of comparisons to other pivots starting from the root of the tree. The objects and the pivot must have compared similarly to the upper-level pivots, increasing the likelihood that they are in close proximity. This is one factor that the current global pivot-based methods are missing; they blindly compute distances between all pivots and all objects. EcKvp improves the process by explicitly performing a query on the pool of pivots. Tree structures use the same weak channels to organize both the pivots and objects, even though pivots are much more important. In contrast, EcKvp organizes the pivots in an expensive but powerful structure. As an example, if EcKvp used a vp-tree as the inner structure, and used an approximate greedy nearest neighbor search process (without the usual back-tracking) as the inner query, then it would have the same information as a vp-tree, even though organized in a different way. Instead, EcKvp is free to use a better pivot organization and a more involved inner query.

Chapter 7

Composite Metrics

In this chapter, we define a new framework called composite metrics for similarity matching. A composite metric is a weighted sum of a set of metrics. A range query in composite metrics not only specifies a query object and a radius value, but also the relative weights to be used for each metric. This enables users to adjust the definition of the distance at the query time according to their needs. This approach can be very useful when the definition of similarity is affected by a number of independent factors, or when there are multiple alternative distance functions that complement each other. To provide some concrete motivation for our framework, we cite several examples from pattern recognition and machine learning. Finally, we introduce two new structures, the c-tree and c-forest, for indexing in composite metrics. Our experiments show that it is possible to provide competitive query performance while providing this extra flexibility.

7.1 Introduction

Recall from Chapter 1 that a *metric space* is a set of objects O and a distance function between database objects that obeys the properties of positivity, symmetry, and triangle inequality [24]. Given such a distance function d, a range query $Range(q, r, O)$ is defined as:

$$Range(q, r, O) = \{o \in O | d(q, o) \leq r\} \tag{7.1}$$

Given a set of metrics, $\{d_1, d_2, ..., d_c\}$ and associated weights $\{w_1, w_2, ..., w_c\}$, we define a *composite metric* to be a linear combination of these distance metrics. That is:

$$d(a, b) = \sum_{i=1}^{c} w_i \cdot d_i(a, b) \tag{7.2}$$

where a and b are two objects from the database. It is straightforward to show that if each of these distance functions is a metric, their linear combination is also a metric.

Note that a family of non-linear combinations can also be made to work in this framework. For example, given a distance function

$$d(a, b) = \sqrt[p]{\sum_{i=1}^{c} (w_i \cdot d_i(a, b))^p}$$

the query range $Range(q, r, O)$ in this setting can be transformed into $Range(q, r^p, O)$ using the composite distance function

$$d'(a, b) = \sum_{i=1}^{c} w'_i \cdot d'_i(a, b)$$

where $d'_i(a, b) = (d_i(a, b))^p$ and $w'_i = w_i^p$.

Range searches in composite metrics can be applied to provide flexibility in the choice of the overall distance function. In addition to having a query object and a radius as input, the search can also specify custom weights to be used in the distance function. For example, we can define a generalized *weighted range query*:

$$Range(q, r, W, O) = \{o \in O | \sum w_i \cdot d_i(q, o) \leq r\}$$

where a vector of constants, $W = w_1, w_2, \ldots, w_c$ is used to assign custom weights to each individual metric. This provides the flexibility at query time to adjust the relative importance of the various components of the distance metric.

7.2 Motivation for Composite Metrics

Metric spaces are based on a distance function that is typically presented as a black box. In contrast, in vector spaces the individual components of the distance computation, the coordinates that is, are all visible. One of the motivations for composite metrics is that by having more information about the inner workings of the distance function, we can improve our index structures. For this point of view to be valid however, one would need to know the weights of individual metrics at preprocessing time.

Our goal here is to provide greater flexibility. We want to improve the usability of the index structure by allowing the user to customize the weights of individual metrics at query time. This can be very useful when the user wishes to inject an element of feedback into the search process, or when there is an underlying system that learns weights based on some pre-established success rate.It is understood that the price of this added flexibility will be some loss in the overall performance, but our goal will be to minimize this performance loss.

There are a number of reasons for considering the combination of a number of

metrics. In many applications, the measure of similarity is a function of a number of independent factors, where each of these factors is itself a similarity measure. For example, the similarity of two persons could be measured as a combination of their similarity of age, location of residence, gender, profession, interests, and so on. Another case is when there are a number of different methods to define similarity between objects. For example, image similarity can be performed by color histograms, texture, and many other different methods that provide varying degrees success depending on the specific application.

Another application where composite metrics are useful arises in machine learning. Machine learning algorithms provide a mapping from objects with a set of attributes or features to either class labels or real values. They are typically given a set of training data with known mappings, and based on these past experiences, estimate the outcome of a new instance. Instance-based learning algorithms [3] base their results on experiences that are similar to the new instance. A distance function can be used to measure the similarity between objects. A detailed discussion of distance functions is presented in [47]. In order to compute the distance between two objects, distances between each feature are combined through a set of weights [5]. When the feature weights are fixed, a typical metric space index structure can be used to speed up the retrieval of relevant training data. G. Atkeson *et al.* [5] cites some examples of systems that use the *k-d* tree [6] for retrieval, however *k-d* trees only work in vector spaces.

Some algorithms modify feature weights to improve the quality of the learner at run-time [45, 44, 39]. These dynamic cases cannot be handled by a traditional

metric space index structure without rebuilding the entire index as the weights are changed. Index structures for composite metrics, however, would be flexible enough to be used in such situations.

Multi-classifier systems in machine learning combine the results of a number of different methods [47, 18]. This was shown to produce better results than the individual learning methods. One of the reasons is that different classifiers have different regions where they perform better, and a composite approach can maximize the influence of the most effective classifiers [18]. In the literature of multi-classifier systems, using classifiers based on different feature sets is called *parallel combining*, whereas using different classifiers based on same feature set is called *stacked combining* [19].

In some applications, such as face recognition and fingerprint matching, each training instance is a class of its own, namely the owner of the face or fingerprint. In these applications, an approximation of the output probability distribution can be constructed directly by the distance of the instance that is being matched to each of the training instances in the database. If the combination strategy is based on summing the individual parts then this process can be completely captured within the framework of composite metrics. It has been shown that this method performs better than other methods [25]. X. Lu *et al.* [30] have used this approach to sum matching scores from three different methods to produce an overall matching score, and their results outperformed individual classifiers. Similarly A. Ross *et al.* [37] combined two methods using different weights for fingerprint matching. Some multiclassifier systems maintain a set of weights that are updated dynamically [41].

This is another example that motivates the custom weighted querying framework in composite metric spaces.

7.3 The C-Tree and C-Forest Structures

To the best of our knowledge, no structure exists in the literature that can answer dynamically weighted queries in the composite metrics framework. A straightforward approach to index composite metrics would be to treat the metric space as having just one composed distance function, and to apply one of the existing methods. This has the disadvantage, however, that one must fix the weights before constructing the structure. Therefore, it does not allow one to query with weights that have been chosen at query time.

In this section we introduce the *c-tree* structure for answering queries over composite metrics. It can be classified classified as a combination of the *k-d tree* [6] and the *vp-tree* [42]. We first start with a brief discussion of these two structures.

The *k-d* tree [6] is a method for indexing vector spaces, it uses one dimension per node to partition the objects and rotates the choice of partitioning plane among the various dimensions with each level of descent in the tree. The median of the coordinate values of the selected dimension is used to partition the current subset in two subsets that are associated with the left child and right child of the node.

In contrast, the vp-tree works in metric spaces. As described in Chapter 2, the construction of a vp-tree with a branching factor of k proceeds in a recursive manner, partitioning the nodes that represent a particular subset of the data. At

each node, one of the objects is selected as the *vantage point*, and the distances from the other objects to this vantage point are calculated. Then these objects are partitioned into k groups of roughly equal size, based on these distances. In this way, a node can have k branches with each subtree having n/k objects, n being the number of objects for that node. The only information that needs to be kept is the vantage point itself, and $k-1$ distance values, denoted as $cutoff[1..k]$, defining the ranges of distances for each subtree. A range query of radius r centered at a point q is performed as follows: at any given node, the distance d between q and the node's vantage point is calculated. If d is smaller than r, the vantage point is added into the result set. For every subset j of the node defined by the cutoff values, if the interval of the subset, $[cutoff[j-1], cutoff[j]]$ intersects the interval $[d-r, d+r]$, then subset j is searched recursively.

The c-tree is also a tree-based structure. It uses one vantage point per node, like the vp-tree, and partitions the objects depending on their distances using only one of the metrics. Therefore, each internal node contains a pivot, a set of ranges of distances for a given metric, and child node pointers associated with each of these ranges. Recall that c denotes the number of metrics in the composition. While traversing the tree in order to process a query, it maintains two c-element vectors to represent the set of objects rooted at the given node, denoted min and max, to help decide whether the current node can be eliminated from the search or not. The value min_i contains the minimum possible distance between the region represented by the current subtree and query object in terms of metric d_i, and max_i contains the maximum distance. We also maintain scalars Min and Max to hold the weighted

sum of *min* and *max* vectors respectively. That is,

$$Min = \sum_{i=1}^{c} w_i \cdot min_i \; and \; Max = \sum_{i=1}^{c} w_i \cdot max_i$$

We can eliminate a subtree from the range search if Min is greater than the query radius. If Max is less than the radius, this means the entire subtree is inside the query range and no further processing is necessary, other than collecting the list of objects in the subtree.

Suppose that, during the traversal of the tree, we are examining a node p having vectors min^p, max^p and the vantage point v_p that partitions the objects based on metric i. A child node u, that lies between distances s and e of metric d_i to v_p, will have min^u and max^u vectors that have the same values as min^p and max^p for each component except i, where:

$$min_i^u = max(min_i^p, [s, e] - d_i(v_p, q))$$

and

$$max_i^u = min(max_i^p, e + d_i(v_p, q))$$

The distance between a range and a distance value is defined as follows:

$$[s, e] - d_i(v_p, q) = \begin{cases} d_i(v_p, q) - e & \text{if } d_i(v_p, q) > e \\ s - d_i(v_p, q) & \text{if } d_i(v_p, q) < s \\ 0 & \text{otherwise} \end{cases}$$

One thing to note about this structure is that it makes it possible to defer defining the metric weights until query time. The structure only depends on individual metrics for organization. It is when we actually perform the query that we determine the distance between a given region and the query object.

The determination of which metrics to choose as representatives of the nodes is a parameter of the structure. Our current implementation permutes the metrics randomly and rotates among them in this order with each level of descent in the tree.

Unfortunately, the vp-tree is not considered to be one of the best performing index structures. For difficult queries where the dimensionality is high or the query radius is high, it does not carry sufficient information to eliminate a large fraction of objects from consideration. In order to address this deficiency, we also introduce the *c-forest* structure, which maintains a collection of c-tree structures and combines their results to process a query. These c-tree structures will have different sets of pivots which increases the likelihood that an object will be eliminated during search. In order to accomplish this, we implemented a special type of query on the c-tree, which, upon reaching a leaf node, does not compute the actual distance but flags the object associated with the node as *uneliminated*. The only distance computations performed are the ones made in internal nodes to navigate through the tree. After repeating this process for all trees, we compute the intersection of the sets of all uneliminated objects. As a final step, we compute the distances between query object and each element in this common intersection to determine whether they should be included in the final result set.

To avoid repeated computations of distances, the c-forest caches the distances computed during the processing of query to be used across the trees. This same optimization can also be applied for the construction of the trees, but our current implementation does not have this feature, and so the construction cost of c-forest

grows linearly with the number of trees.

7.4 Performance

Because we know of no other index structures for handling composite metrics, the only basis for comparison for the c-tree and c-forest structures is sequential scan. We use the term *cost ratio* to denote the ratio of the total number of distance computations of our structures to sequential scan, that is, the size of the database. Beyer et al. [7] point out that, due to the nature of modern storage systems, an index structure is considered to be effective if it reduces the number of candidates by at least 90%, which corresponds to a a cost ratio of 0.1. We expect objects in our domain to be larger than records in a traditional relational database, so the ratio of disk seek time to the total time should be lower than for typical database applications. Thus, cost ratio values that are higher than 0.1 could still be considered efficient depending on the size of the objects. Of course, seek times are only an issue for disk-based implementations. If the database resides in main memory, it is much easier to demonstrate improvements over linear scan.

Even though traditional index structures are not designed to work in the composite setting, we have chosen to include two well known metric space indexing methods, the vp-tree and GNAT in some of our experiments. Our approach is to let the traditional methods work with fixed weights for both the construction and query phases. Note that this is not a fair comparison for both sides, since on one hand, the c-tree and c-forest provide greater flexibility, and have to handle weights

that are not given until query time, whereas the vp-tree and GNAT have to compute the full composite distance on the whole object and cannot execute partial distance computations on the parts of the objects. Nevertheless, we would like to investigate the performance cost incurred by the additional flexibility of allowing weights to be given at query time.

7.4.1 Data Sets

We ran experiments on two types of data sets, one representing images and the other consisting of synthetically generated vectors. The images present us with the case where the metrics are defined to produce similar results, and vectors were used to model cases where the components are independent.

Our image database consists of 1800 images. For each image there is a 64-dimensional color histogram and a 62-dimensional feature vector generated by Gabor texture filters [32]. This provides us with a setting where there are two complementary metrics. Figure 3.5 presented in Chapter 3 shows the distance distribution of the images based on these two distance metrics.

The majority of our experiments were performed using random vectors. Each object had a collection of vectors sampled from the unit hypercube that were generated independently of each other. We used Euclidean distance between vectors, where distances were normalized to the range [0,1].

In order to simulate a setting where components might have varying degrees of complexity, we used distributions that we label as $v(d_0, d_1, k)$ where each object

is represented as a collection of uniformly distributed random vectors having dimensions between d_0 and d_1, and there are k instances of each vector. In other words, there are a total of $k \cdot (d_1 - d_0 + 1)$ vectors having dimensions $d_0, d_0 + 1, ..., d_1$. The normalization of the distances ensures that only the query weights determine the relative importance of components, not their dimensionality.

We examined various different distributions of vectors. For this reason, we created *multiples*, a set of datasets labeled as $m(distr, dim, mult)$ where *mult* many of vectors were sampled from a certain distribution *distr* in dimension *dim* . We used various distributions for the vectors as described in Chapter 3. Figure 3.2 shows the histogram of distances for these distributions.

7.4.2 Experiment Setting

In all of our queries the custom query weights were randomly generated and their sum was normalized to 1 in an effort to keep the difficulty of the queries constant. In general, using high weights will have a similar effect to using low radius values, thus making the query easier. For the vp-tree and GNAT, we made all the weights equal so that their sum is 1.

The major parameter of c-tree, vp-tree, and GNAT that can be adjusted by the user is the tree's branching factor (that is, the out-degree of each node). Some of our graphs are scatter plots that show these possible different settings of the same structure. Usually, a branching factor that results in more construction cost stores more distance information and yields relatively better performance. Our graphs

show the trade-off between the construction cost and the cost ratio.

The c-forest structure can use as many c-trees as desired. These c-tree structures might have varying branching factors, but for our experiments we fixed the branching factors of all the c-trees. A label "*c-forest* $\langle m \rangle$" in our graphs represents the set of c-forests that use m as the branching factor of its internal trees. We obtained different construction cost settings by varying the number of trees inside the c-forest.

7.4.3 Experimental Results

In this section we outline the performance characteristics of our structures under different conditions.

The difficulty of a composite metric query is determined both by the number of components and their individual complexity. This is explored in Figure 7.1. The cardinality of the database is another important parameter that determines the query cost. When the cost of the query is sublinear, the cost ratio values are expected to improve as the size of the database grows. Figure 7.2 summarizes our results.

We see that the results show similar characteristics across different radius values. The rest of our experiments assume a fixed query radius to emphasize other factors. We have also seen that larger cardinalities provide more favorable results in terms of the cost ratio, but in our following experiments we limit the database to moderate sizes in order to present a more balanced view.

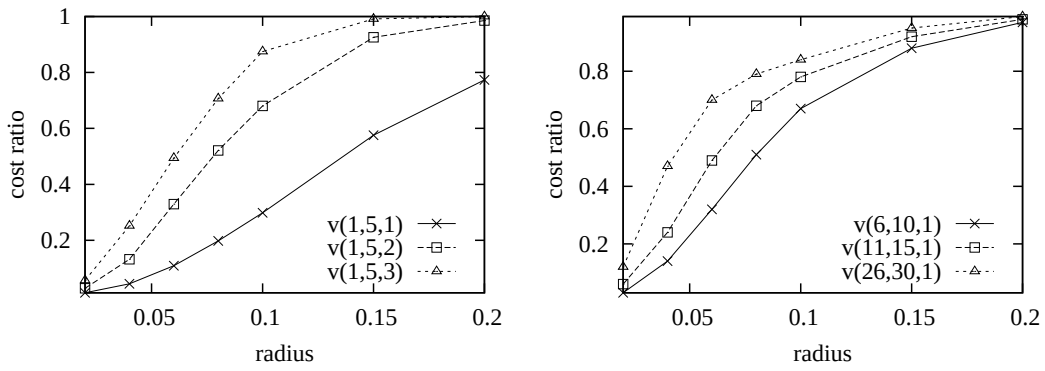


Figure 7.1: Varying the number of components (left) and complexity of the components (right) on the c-tree.

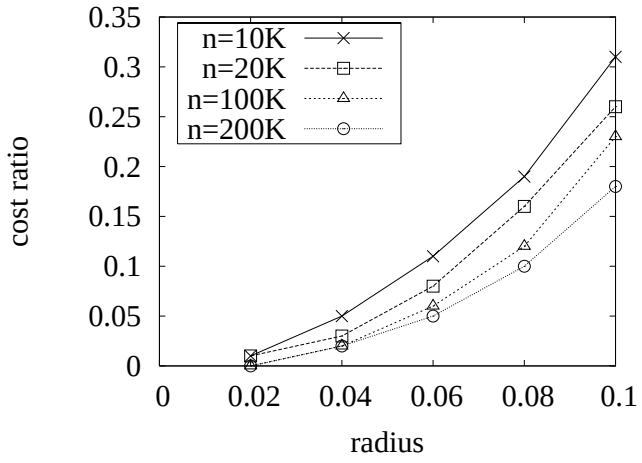


Figure 7.2: Size of the database and query performance in $v(1,5,1)$ with a c-tree having a branching factor of 2

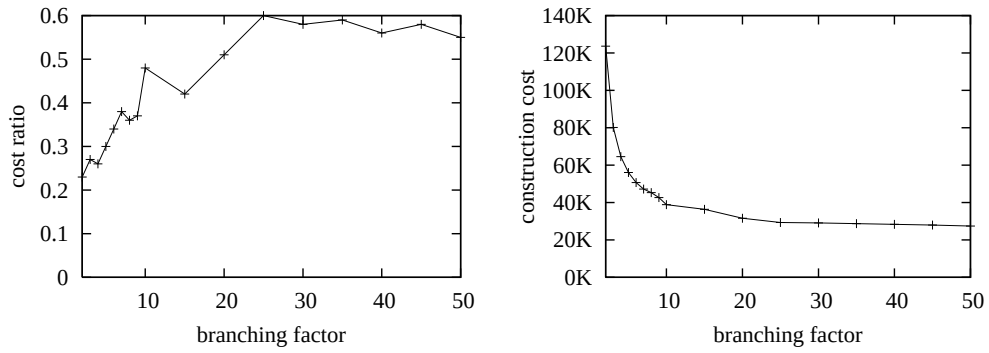


Figure 7.3: Effects of varying fanout on query performance and setup cost for $r = 0.04$ in $v(1, 5, 3)$ having 10000 objects

The branching factor of the c-tree is its most important parameter. Different branching factors produce different possibilities for construction cost and query performance. This is demonstrated in Figure 7.3. Higher branching factors produce shallower trees, and therefore the total number of pivots decreases and so does the construction cost.

The c-forest structure offers even more flexibility for trade-offs between construction times and query performance. Here, the important parameters are the number of c-tree structures used, as well as the fanout of these internal trees. Figure 7.4 summarizes our experiments for random vectors. We see that the c-forest can provide better performance at the expense of greater construction costs. It also appears that for our settings, a branching factor of 4 gave the best results.

Figure 7.5 shows the performance results of c-tree for the image data. Once again we see that the c-tree provides improvements over sequential scan.

In our experiments, we observed that GNAT consumes too much construction time as compared to its query performance. It eventually produces competitive

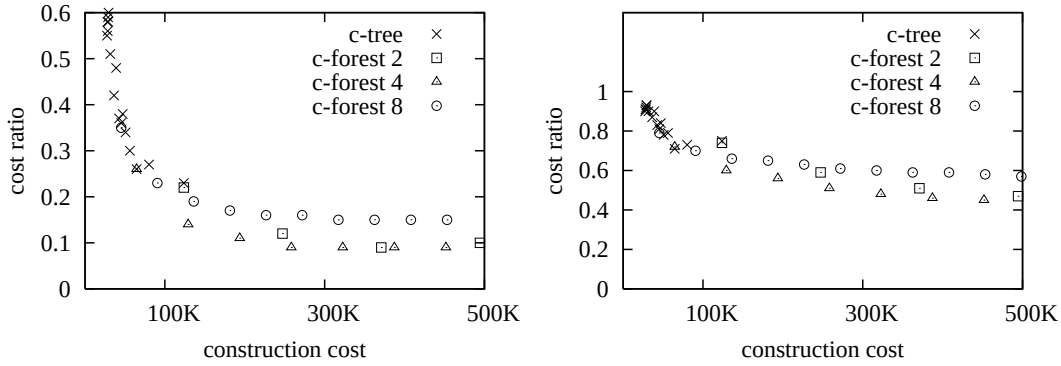


Figure 7.4: Query performance versus setup cost for various c-tree and c-forest configurations for $r = 0.04$ (left) and $r = 0.08$ (right) for $v(1, 5, 3)$.

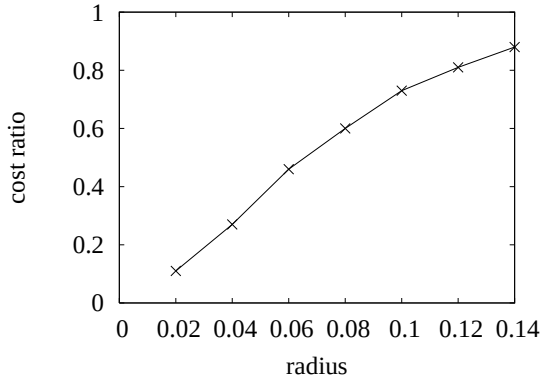


Figure 7.5: c-tree performance on image data.

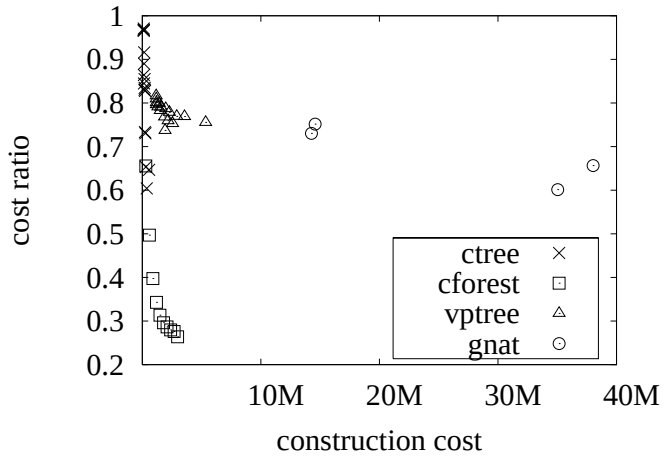


Figure 7.6: Performance by construction cost in $m(uniform, 10, 5)$ having 40K objects for query radius 0.08.

results, but at too great a cost. Figure 7.6 provides a typical comparison between GNAT and the other structures.

Finally, we compared our structures with their natural non-composite counterpart, the vp-tree. Figure 7.7 illustrates the relative cost ratios of these structures using various distributions. The uniform distribution proves to be the most difficult one as Figure 3.2 has already suggested. In this distribution only the c-forest provides adequate query performance. In the other distributions, we see that we can go well beyond the cost ratio value of 0.1. As we move to less difficult distributions, the extra power provided by the c-forest is less evident. In some cases we actually observe poorer performance as we increase the number of inner c-trees. We also see that, for our settings, the c-tree matches the query performance of the vp-tree with lower construction costs and sometimes outperforms it. Given that the c-tree and vp-tree are closely related in terms of their internal structures, this shows that com-

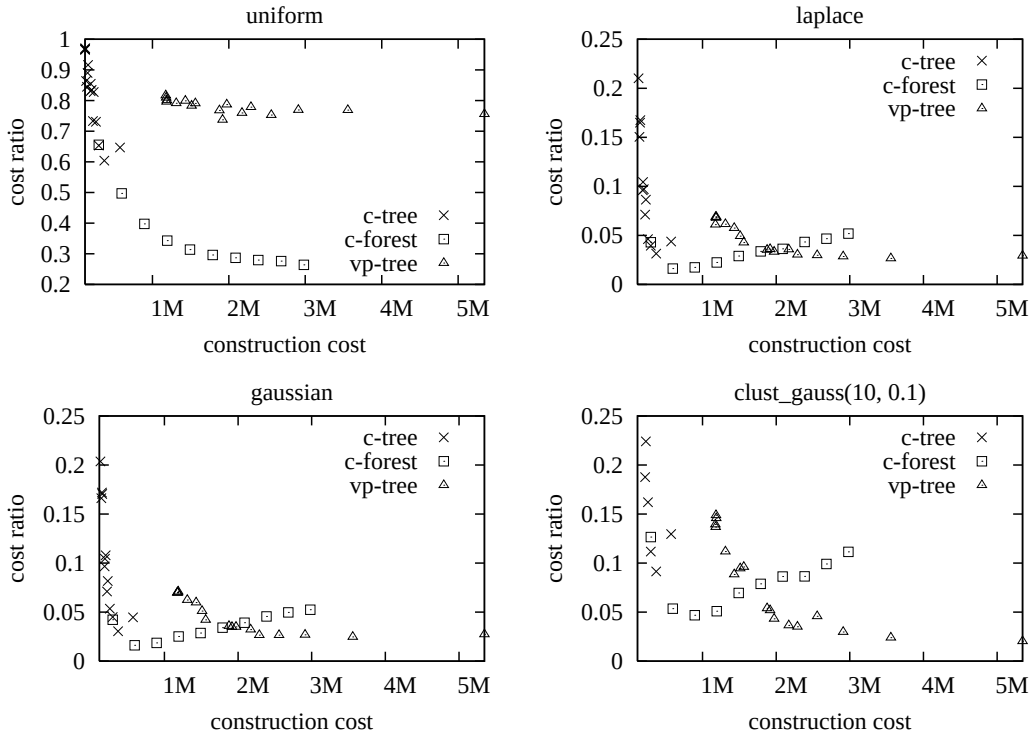


Figure 7.7: Comparison of the structures in various distributions as defined in Chapter 3 for $m(*, 5, 10)$. The database size is 40K, query range 0.08.

posite metrics point of view can indeed improve performance by avoiding computing the full distance computation when possible.

We also see that the c-tree and c-forest are relatively less successful for the given clustered distribution while vp-tree seems unaffected. Figure 7.8 presents the comparison of the best settings of the structures for various query radius values. For c-tree, we used branching factors between 2 and 50. As pointed out before, low fanout values produced the best results. For the c-forest, the internal branching factors were set to 4, and the number of internal c-tree structures varied between 1 and 10. For the vp-tree, we used branching factors up to 50 as we have done with the c-tree structure.

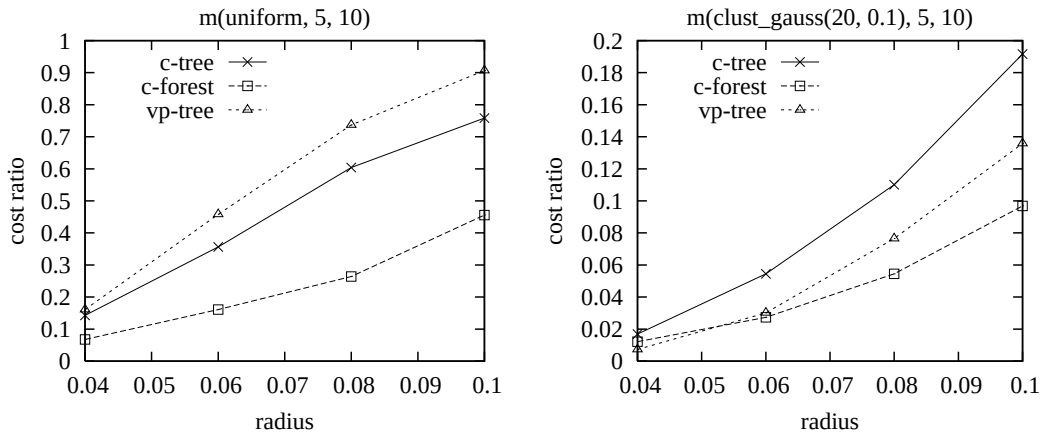


Figure 7.8: Comparison of the best settings of the structures for various query radius values. The database size is 40K.

7.5 Summary and Conclusions

In this chapter we introduced a concept called composite metrics in which a distance function is defined as a linear combination of several metrics. We also defined a new type of range query where the weights of the metrics are given as parameters to the query.

We presented examples from machine learning and multi-classifier systems where composite metrics can be of significant value. In multi-classifier systems classification is based on a number of factors that can be alternatives of each other as in stacked combining of classifiers, or unrelated components of the object as in parallel combining.

We introduced two structures, the c-tree and c-forest, both of which can perform queries in our framework. C-tree is a hierarchical organization of the objects based on distances to local pivots similar to the vp-tree. It can be constructed with-

out dealing with component weights since it handles the components one at a time. C-forest is a collection of c-tree structures which are designed only for elimination of the objects, and not for computing the actual query. The query is finalized by a sweep of objects that are not eliminated by any of the c-tree structures.

We show that c-tree and c-forest can provide significant improvements over sequential scan, and even over the well known vp-tree and GNAT structures.

Chapter 8

Conclusions

In this thesis, we have presented a number of practical improvements to data structures and algorithms for similarity searching in metric spaces, both in the standard context and in a new context called composite metrics. We have evaluated the efficiency of these data structures through a number of empirical analyses. A recurring theme in our methods has been the idea of selecting the most effective pivots, that is, the pivots that are either very close or distant to the query object.

We showed that there is a high correlation between the construction cost of a structure and its query performance. The pivoting operation is the essential means used in these structures for eliminating objects without explicitly computing the distance between these objects and the query object. The greater the construction cost, the more pivots a structure can store, and hence the fewer distance computations are needed at query time. Tree structures like vp-tree and M-tree variants, where the number of pivots is dependent on the tree height are very restricted from this point of view, that is, it is very difficult to invest more in the construction phase to improve query times. In the GNAT structure, the representatives stored in a node do not only store information about their own subtrees. As a result, it is possible to increase the construction cost by increasing the branching factor of the tree.

For structures having the same construction cost, ignoring the computational overhead, the success at query time depends on the efficient use of the pivots. Global pivot-based structures use all the information available, they store the exact distances between the pivot and all of the objects. Other structures typically attempt to group similar objects together and they store distance ranges to these objects. Although this approach may reduce the computational overhead, since the elimination of a single subtree means the elimination of all the objects in it, it reduces the power of the pivots. The likelihood that two distance ranges intersect is higher than the likelihood that one single distance value is within a distance range. As the complexity of data distribution increases, it becomes more difficult to cluster relevant objects together, making the pivots even weaker. This observation is supported by experimental evidence, which indicates that tree structures perform poorer when using the same amount of construction cost.

Although vantage points-based methods make full use of the pivots, this also means they use more space, more computational overhead, and greater construction cost. The main flaw here is that a pivot governs the whole population. We have shown that a pivot has varying degrees of effectiveness, and it is particularly effective for objects that are either close to it or far from it. It may happen to help eliminate other objects especially when the query object is close to the database object, but these contributions of the pivot are negligible compared to more closely related cases.

One of our main contributions, the Kvp structure, uses this fact to eliminate unpromising distance values from the structure. This means there is less information to store, and less information to process at the query time. This also eliminates

the need to use more complicated, global data structures in order to reduce computational overhead. That is why we proposed a flat, unordered structure where each object only stores its distances to relevant pivots. This organization makes it straightforward to implement insertion and deletion operations. Unlike most of the other methods, it requires just a sequential scan of the indexing data, which renders it an excellent candidate to be stored on secondary memory.

We also introduced the EcKvp structure, which offers a solution to the construction cost problem. Even though Kvp eventually discards less useful distance information, it has to compute the distance between the object and the pivot before it can decide that the distance value is unpromising. Many of these costly distance computations are wasted. EcKvp avoids computing all the distances between a pivot and the objects by organizing the pivots themselves in the HKvp structure. At construction time, each object executes a range query on the pivots to retrieve the distance values to a subset of them. Our experiments show that these queries are successful in returning promising pivot distances.

We introduced a variant of the Kvp structure, called the HKvp structure. It helps when there are more pivots than necessary to process a query. It offers the flexibility of eliminating pivots as well as database objects. While eliminating pivots, it uses a measure of importance of pivots. We have shown that the HKvp can identify the promising pivots, and in addition it possesses the same space and computational savings of the Kvp structure.

Finally, we defined the concept of composite metrics to provide a flexible setting for similarity queries, where the distance is defined to be a weighted linear com-

bination of a given set of metrics. The user can pose queries in which the weights are given at query time. We introduced two index structures, the c-tree and c-forest, for handling composite metric queries. We have demonstrated experimentally that they can answer such queries efficiently.

8.1 Future Research

Overall we feel that these structures and concepts introduced in this thesis provide a significant improvement to our understanding of efficient index structures for similarity search in metric spaces. There are, however, a number of areas for future research. The EcKvp structure, for example, has a number of shortcomings. The drop rate parameter introduced with HKvp needs to be adjusted according to the difficulty of the object distribution at hand. We mentioned that this parameter is only used at query time, and therefore it is possible to install a simple learning system that can use the optimal drop rate value based on the query radius. In Chapter 3 we suggested possible approaches to assess the difficulty of a query and calculate the optimal number of pivots to be used. This matter merits further investigation to make our solutions more practical.

We showed that in general using more pivots is better for EcKvp inner queries, but more pivots mean that the construction cost of the inner index grows quadratically, which increases preprocessing time considerably. This suggests that there is an optimal value for the number of pivots where the benefit of using more pivots is balanced against the construction cost. For high-dimensional synthetic data for

which we have a well-defined distance distribution, this optimal value was very high, and was not an issue. This may not be the case for other distributions, however. Determining the best value is a challenging problem. One approach might be to include an optimizer module that tries different variations of inner index size and inner query radius in order to keep the total construction cost at a desired level. Statistics like the optimal inner query radius to achieve the minimum cost per efficiency can be used to improve the optimizer. This can be performed independently without causing any structural changes in the index.

EcKvp uses a variation of the range query to discover relevant pivots, but different approaches can be used to improve the inner query. Note that we have a lot of flexibility here since we have no obligation of returning all the objects that qualify for the query.

In this thesis, we only focused on the range queries. There are other types of queries, most notably the k-nearest neighbor query, which we have not investigated. One reason is these queries can be solved by using a series of range queries. However, it is also possible to develop special algorithms to solve these problems directly.

The methods we analyzed and introduced all produce exact results, in other words, they return all the objects that lie within the query range, and nothing else. Another approach for improving efficiency is through approximation. In approximate range searching it is possible to incorrectly classify objects that lie sufficiently close to the boundary of the query range. These algorithms are expected to run faster in return for their imprecise results. Although some of these methods can be applied to all of the structures in general, there may be ways of improving the query

performance or success rates.

BIBLIOGRAPHY

[1] http://www.corel.com.

[2] http://www.google.com/programming-contest/.

[3] David W. Aha, Dennis Kibler, and Marc K. Albert. Instance-based learning algorithms. *Mach. Learn.*, 6(1):37–66, 1991.

[4] Sunil Arya and David M. Mount. Algorithms for fast vector quantization. In J. A. Storer and M. Cohn, editors, *Proceedings DCC93 (IEEE Data Compression Conference)*, pages 381–390, Snowbird, UT, USA, 1993.

[5] Christopher G. Atkeson, Andrew W. Moore, and Stefan Schaal. Locally weighted learning for control. *Artificial Intelligence Review*, 11(1-5):75–113, 1997.

[6] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[7] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? *Lecture Notes in Computer Science*, 1540:217–235, 1999.

[8] Tolga Bozkaya and Meral Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 357–368, New York, NY, USA, 1997. ACM Press.

[9] Sergey Brin. Near neighbor search in large metric spaces. In *The VLDB Journal*, pages 574–584, 1995.

[10] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, 1973.

[11] Cengiz Celik. Priority vantage points structures for similarity queries in metric spaces. In *Proceedings of EurAsia-ICT*, volume 2510 of *Lecture Notes in Computer Science*, pages 256–263. Springer, 2002.

[12] Edgar Chávez, José L. Marroquín, and Ricardo A. Baeza-Yates. Spaghettis: An array based algorithm for similarity queries in metric spaces. In *SPIRE/CRIWG*, pages 38–46, 1999.

[13] Edgar Chávez, José L. Marroquín, and Gonzalo Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools Appl.*, 14(2):113–135, 2001.

[14] Edgar Chávez and Gonzalo Navarro. Towards measuring the searching complexity of general metric spaces. Technical report, 2001.

[15] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José Luis Marroquín. Searching in metric spaces. *ACM Comput. Surv.*, 33(3):273–321, 2001.

[16] Paolo Ciaccia, A. Nanni, and Marco Patella. A query-sensitive cost model for similarity queries with m-tree. In *Australasian Database Conference*, pages 65–76, 1999.

[17] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *The VLDB Journal*, pages 426–435, 1997.

[18] Thomas G. Dietterich. Ensemble methods in machine learning. *Lecture Notes in Computer Science*, 1857:1–15, 2000.

[19] Robert P. W. Duin and David M. J. Tax. Experiments with classifier combining rules. In *MCS '00: Proceedings of the First International Workshop on Multiple Classifier Systems*, pages 16–29, London, UK, 2000. Springer-Verlag.

[20] Christos Faloutsos, Ron Barber, Myron Flickner, Jim Hafner, Wayne Niblack, Dragutin Petkovic, and William Equitz. Efficient and effective querying by image content. *J. Intell. Inf. Syst.*, 3(3/4):231–262, 1994.

[21] Gísli R. Hjaltason and Hanan Samet. Index-driven similarity search in metric spaces. *ACM Trans. Database Syst.*, 28(4):517–580, 2003.

[22] Caetano Traina Jr., Agma J. M. Traina, and Christos Faloutsos. Distance exponent: A new concept for selectivity estimation in metric trees. In *ICDE*, page 195, 2000.

[23] Caetano Traina Jr., Agma J. M. Traina, Bernhard Seeger, and Christos Faloutsos. Slim-trees: High performance metric trees minimizing overlap between nodes. In *Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology, Konstanz, Germany, March*

27-31, 2000, Proceedings, volume 1777 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 2000.

[24] J.L. Kelly. *General Topology.* D. Van Nostrand Company, New Jersey, 1955.

[25] Josef Kittler, Mohamad Hatef, Robert P. W. Duin, and Jiri Matas. On combining classifiers. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(3):226–239, 1998.

[26] Donald E. Knuth. *Fundamental Algorithms.* The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts, 1973.

[27] Lillian Lee. Measures of distributional similarity. In *37th Annual Meeting of the Association for Computational Linguistics*, pages 25–32, 1999.

[28] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Daklady*, 10:707–710, 1966.

[29] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. The tv-tree: An index structure for high-dimensional data. *VLDB J.*, 3(4):517–542, 1994.

[30] Xiaoguang Lu, Yunhong Wang, and Anil K. Jain. Combining classifiers for face recognition. volume 3, pages 13–16. ICME, 2003.

[31] D. Maio and D. Maltoni. A structural approach to fingerprint classification. In *ICPR '96: Proceedings of the International Conference on Pattern Recognition (ICPR '96) Volume III-Volume 7276*, page 578, Washington, DC, USA, 1996. IEEE Computer Society.

[32] B. S. Manjunath and W. Y. Ma. Texture features for browsing and retrieval of image data. *IEEE Trans. Pattern Anal. Mach. Intell.*, 18(8):837–842, 1996.

[33] J.; Carrasco-Jimnez R.C. Mic-Andrs, M.L.; Oncina. A fast branch and bound nearest neighbour classifier in metric spaces. *0167-8655 - Pattern Recognition Letters*, 17(7):731–739, 1996.

[34] L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.

[35] Luisa Micó, José Oncina, and Enrique Vidal. A new version of the nearest-neighbour approximating and eliminating search algorithm (aesa) with linear preprocessing time and memory requirements. *Pattern Recognition Letters*, 15(1):9–17, 1994.

[36] Nene and Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEETPAMI: IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19, 1997.

[37] Arun Ross, Anil K. Jain, and James Reisman. A hybrid fingerprint matcher, August 2002.

[38] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[39] Steven Salzberg. A nearest hyperrectangle learning method. *Mach. Learn.*, 6(3):251–276, 1991.

[40] Dennis Shasha and Tsong-Li Wang. New techniques for best-match retrieval. *ACM Trans. Inf. Syst.*, 8(2):140–158, 1990.

[41] Ching Y. Suen and Louisa Lam. Multiple classifier combination methodologies for different output levels. In *MCS '00: Proceedings of the First International Workshop on Multiple Classifier Systems*, pages 52–66, London, UK, 2000. Springer-Verlag.

[42] Jeffrey K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Inf. Process. Lett.*, 40(4):175–179, 1991.

[43] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145, 1986.

[44] Dietrich Wettschereck and David W. Aha. Weighting features. In *ICCBR '95: Proceedings of the First International Conference on Case-Based Reasoning Research and Development*, pages 347–358, London, UK, 1995. Springer-Verlag.

[45] Dietrich Wettschereck and Thomas G. Dietterich. An experimental comparison of the nearest-neighbor and nearest-hyperrectangle algorithms. *Machine Learning*, 19(1):5–27, 1995.

[46] David A. White and Ramesh Jain. Similarity indexing with the ss-tree. In *ICDE '96: Proceedings of the Twelfth International Conference on Data Engineering*, pages 516–523, Washington, DC, USA, 1996. IEEE Computer Society.

[47] D. Randall Wilson and Tony R. Martinez. Improved heterogeneous distance functions. *J. Artif. Intell. Res. (JAIR)*, 6:1–34, 1997.