Index Structures for Similarity Search in Multimedia Databases

Dissertation
zur Erlangung des akademischen Grades
des Doktors der Naturwissenschaften (Dr. rer. nat.)
an der Universität Konstanz, Fachbereich Informatik

vorgelegt von
Benjamin Eugenio Bustos Cárdenas

Universität Konstanz

Juni 2006

Tag der mündlichen Prüfung: 16.10.2006

1. Referent: Prof. Dr. Daniel A. Keim, Universität Konstanz
2. Referent: Prof. Dr. Gonzalo Navarro, Universität Chile

Acknowledgments

I would like to express my gratitude to all the people who supported me for the last four years, while I have been working on my Ph.D. thesis.

I would like to thank Prof. Dr. Daniel Keim, my thesis advisor, for his excellent guidance during my Ph.D. I want to thank him first of all for letting me come to Germany to work in his group, for being there when I needed advice, and for letting (and encouraging) me to work on the topics that interested me most. I am also thankful for his friendship and the unconditional support that he gave me during all these years.

I would also like to thank Prof. Dr. Gonzalo Navarro for his valuable suggestions and constant support. He encouraged me to publish my first paper when I was still an Engineering student at the University of Chile, and later to follow an academic career. He put me in contact with Prof. Dr. Keim and made it possible for me to get my Ph.D. position in Konstanz. He was always willing to discuss interesting topics that helped me tremendously during my ongoing research. Also, he generously accepted to serve as a second referee of my thesis and made useful comments that improved its quality. For all these reasons, I am very grateful to him.

I must also offer my thanks to my colleagues and friends at the University of Konstanz: Tobias Schreck, Florian Mansmann, Jörn Schneidewind, Mike Sips, Daniela Oelke, Hartmut Ziegler, Christian Panse, Markus Wawryniuk, Svetlana Mansmann, Evghenia Stegantova, Stefan Hiller, and Maria Carolina Otero. My special thanks go to Tobias Schreck, for his friendship and for all the exciting joint work that we did together during my first years in Konstanz. I would also like to thank Prof. Dr. Dietmar Saupe and Prof. Dr. Oliver Deussen for their help and suggestions with, respectively, the 3D model retrieval project and the algorithms based on graphics processor units.

A special acknowledgement goes to my dear friend Nelson Morales. He provided insightful comments and suggestions for many parts of my thesis, and was always willing to answer my technical questions.

I would also like to thank my family, who always supported me during these four years. Special thanks go to my father, Roberto, and my aunt, Sonia, who handled all the formalities that had to be taken care of in Chile while I was in Germany. Thanks to my friends: Javier Gonzalez, Javier Bustos, Ricardo Lemus, Claudia Farina, and many others, for their constant moral support.

The Faculty of Physics and Mathematics of the University of Chile and the German Research Foundation (DFG) Project No. KE 740/6-1, within the strategic research initiative “Distributed Processing and Delivery of Digital Documents” (V3D2) SPP 1041, which partially supported this work, are gratefully acknowledged. I also thank the anonymous conference and journal reviewers for their helpful comments.

Konstanz, June 2006

Benjamin Bustos

Abstract

An important research issue in the field of multimedia databases is the *retrieval of similar objects*. For most applications in multimedia databases, an exact search is not meaningful. Thus, much effort has been devoted to develop efficient and effective similarity search techniques. A widely used approach for implementing similarity search engines is the *feature-based approach*. In this approach, all multimedia objects stored in a database are transformed into high-dimensional *feature vectors*, which are then inserted into an index structure to efficiently perform similarity queries.

The contribution of this thesis is to explore and propose novel solutions to improve the efficiency of similarity queries in multimedia databases. The thesis begins with a study on how to improve the effectiveness (i.e., the quality of the answer) of a similarity retrieval engine. We first show that by using *combinations of feature vectors* the effectiveness of the similarity search may be significantly enhanced. Then, we describe methods for computing query-dependent weights to perform linear combinations of feature vectors, which can further improve the effectiveness of the similarity search. As almost all index structures for similarity search developed so far can only deal with single feature vectors, the design and analysis of new index structures is necessary to efficiently perform similarity queries that use combinations of feature vectors. This gives an extra motivation for the techniques studied in the rest of the thesis.

In the next part of the thesis, we propose several algorithms and index structures to improve the efficiency of similarity queries. Firstly, we study *pivot selection techniques* for pivot-based indices. We provide an efficiency criterion based on distance histograms for selecting good set of pivots, and present empirical evidence showing that the technique is effective. Secondly, we describe an *improved k nearest neighbor (k-NN) algorithm*, which is based on the best-first traversal algorithm proposed by Hjaltason and Samet. Although the original algorithm is already optimal in the number of distance computations, its space requirements are significant. The improved algorithm aims to lower the space requirements by using distance estimators. Thirdly, we present a *metric access method for dynamic combinations of feature vectors*. The index is pivot-based, and it can take advantage of the previously studied pivot selection techniques. Finally, we introduce an approach that aims to minimize the expected search cost of a similarity query. The idea is to index only the *most frequently used combinations of feature vectors*. If there are restrictions on the available space for constructing indices, then the resulting optimization problem can be modeled as a binary linear program. As binary linear programs are NP-hard in the general case, we also propose algorithms that quickly find good sets of indices.

The last part of the thesis explores the *use of graphics processor units (GPUs) for accelerating database operations*. We present GPU implementations of a high-dimensional nearest neighbor search and a clustering algorithm. An experimental evaluation shows that the proposed GPU algorithms are an order of magnitude faster than their CPU versions.

Zusammenfassung

Im Arbeitsbereich der Multimedia-Datenbanksysteme ist die *Suche ähnlicher Objekte* ein wichtiges Forschungsgebiet. Für die meisten Anwendungen in Multimedia-Datenbanken ist eine exakte Suche nicht sinnvoll. Deshalb wurde viel Mühe darauf verwendet, effiziente und effektive Methoden der Ähnlichkeitssuche zu entwickeln. Eine weit verbreitete Methode für die Implementierung der Ähnlichkeitssuche ist die auf Objekteigenschaften basierende Methode. Diese Methode transformiert alle multimediale Objekte, die in einer Datenbank gespeichert sind, in hochdimensionale Feature-Vektoren. Nach der Transformation werden diese Feature-Vektoren in einen Indexstruktur eingefügt, um so effiziente Ähnlichkeitssuchen durchzuführen.

Diese Dissertation trägt dazu bei neuartige Lösungen vorzuschlagen, indem sie erforscht wie die Effizienz der Ähnlichkeitssuche verbessert werden kann. Die Dissertation beginnt mit einer Untersuchung, wie man die Effektivität (d.h. die Qualität der Ergebnisse) der Ähnlichkeitssuche verbessern kann. Es wird aufgezeigt, dass mit *Kombinationen von Feature-Vektoren* die Effektivität der Ähnlichkeitssuche bedeutsam erweitert werden kann. Es werden Methoden vorgestellt, um Suchobjektsabhängig Gewichte für lineare Kombinationen von Feture-Vektoren zu berechnen, was die Effektivität der Ähnlichkeitssuche weiter verbessern kann. Das Design und die Analyse von neuen Indexstrukturen sind notwendig, um Ähnlichkeitsanfragen mit Kombinationen von Feature-Vektoren effizient durchzuführen, denn bislang können fast alle Indexstrukturen für Ähnlichkeitssuche nur individuelle Feature-Vektoren indizieren. Dies begründete die besondere Motivation, der in der restlichen Dissertation erforschten Methoden.

Im zweiten Teil der Dissertation werden verschiedene Algorithmen und Indexstrukturen vorgeschlagen, um die Effizienz der Ähnlichkeitssuche zu verbessern. Zuerst werden *Techniken zur Auswahl von Pivots* für Pivot-basierte Indexstrukturen untersucht. Es wird ein auf Abstandhistogramme basiertes Effizienzkriterium angeführt, um gute Pivotmenge auszuwählen und der empirische Beweis dargelegt, dass die dargebotene Methode effektiv ist. Desweiteren wird ein *verbesserter k-Nächste-Nachbarn-Algorithmus* dargestellt, welcher auf dem best-first Algorithmus von Hjaltason und Hamet basiert. Obwohl der ursprüngliche Algorithmus in der Abstandsberchnung optimal ist, sind seine Raumanforderungen bedeutend. Der verbesserte Algorithmus zielt auf die Absenkung der Raumanforderungen mit Hilfe von Distanzschätzfunktionen ab. Drittens wird eine *metrische Indexstruktur für dynamische Kombinationen von Feature-Vektoren* dargestellt. Die Indexstruktur ist Pivot-basiert und kann einen Vorteil aus den zuvor entwickelten Pivotwahlmethoden ziehen. Zum Abschluss wird eine Annäherung ausgeführt, welche auf die Minimierung der zu erwartenden Suchzeit einer Ähnlichkeitsuche zielt. Die prinzipielle Idee ist, die *am häufigsten benutzteten Kombinationen von Feature-Vektoren* zu indizieren. Das sich ergebende Optimierungsproblem kann zu einem Binary-Linear-Program ummodelliert werden, wenn es bei der Speicherung von Indexstrukturen Beschränkungen des Speicherplatzes gibt. Da Binary-Linear-Programs im Allgemeinen NP-hart sind, werden Algorithmen vorgeschlagen, die auf schnelle Weise gute Indexmengen finden.

Der Schlußteil der Dissertation beschreibt die Erforschung der Verwendung von Grafikkartenprozessoren (GPU) für die Beschleunigung von Datenbankoperationen. Es werden GPU-Implementierungen für hoch-dimensionale Nächste-Nachbarn-Suche und ein Clusteringalgorithmus vorgestellt. Die experimentelle Evaluierung zeigt, dass die vorgestellte GPU-basierten Algorithmen eine Größenordnung schneller sind als die gleichen aus CPU-basierenden Algorithmen.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The development of multimedia database systems and retrieval components is becoming increasingly important, due to a rapidly growing amount of available multimedia data like images, audio files, video clips, 3D objects, and text documents. As we see progress in the fields of acquisition, storage, and dissemination of various multimedia formats, the application of effective and efficient database management systems becomes indispensable in order to handle these formats.

The application domains for multimedia databases include molecular biology, medicine, geographical information systems, Computer Aided Design/Computer Aided Manufacturing (CAD/CAM), virtual reality, and many others:

- In medicine, the detection of similar organ deformations can be used for diagnostic purposes. For example, the current medical theory of child epilepsy assumes that an irregular development of a specific portion of the brain, called the hippocampus, is the reason for epilepsy. Several studies show that the size and shape of the deformation of the hippocampus may indicate the defect. This may be used to decide whether or not to remove the hippocampus by brain surgery. Searching for similar deformations in a database of hippocampi models can support the decision process and help to avoid unnecessary surgeries [Keim, 1999].

- Biometric devices (e.g., fingerprint scanners) read a physical characteristic from an individual and then search in a database to verify if the individual is registered or not. The search cannot be exact, as the probability that two fingerprint scans, even from the same person, are exactly equal (bit-to-bit) is very low.

- A 3D object database can be used to support CAD tools, because a 3D object can model exactly the geometry of an object and any information needed about it can be derived from the 3D model (e.g., any possible 2D view of the object). These CAD tools have many applications in industrial design. For example, standard parts in a manufacturing company can be modeled as 3D objects. When a new product is designed, it can be composed of many small parts that fit together to form the product. If some of these parts are similar to one of the standard parts already designed, then the possible replacement of the original part with the standard part can lead to a reduction of production costs.

- In text databases, a typical query consists of a set of keywords or a whole document. The documents are modeled as vectors [Baeza-Yates and Ribeiro-Neto, 1999], which are used to assess the semantic similarity between the query (keywords or document) and the stored documents. The search system looks in the database for the most similar (i.e., relevant) documents to the query. The search system may also allow a certain tolerance on the search in case, for example, that some of the given keywords were mistyped or an optical character recognition (OCR) system was used to scan the documents (thus they may contain some misspelled words).

- Movie and video game producers make frequent use of audio tracks, video sequences, and 3D models to enhance realism in entertainment applications. Reuse and adaptation of multimedia material by searching in existing databases is a promising approach to reduce costs in the creation of new material.

A common characteristic of all applications in multimedia databases is that a query searches for *similar objects* instead of performing an exact search, as in traditional relational databases. Therefore, one of the most important tasks in a multimedia retrieval system is to implement effective and efficient *similarity search algorithms* [Keim and Bustos, 2004]. Multimedia objects cannot be meaningfully queried in the classical sense (exact search), because the probability that two multimedia objects are identical is negligible, unless they are digital copies from the same source. Instead, a query in a multimedia database system usually requests a number of objects most similar to a given query object or to a manually entered query specification.

1.1 Similarity search in multimedia databases

One approach to implement similarity search in multimedia databases is by using *annotation information* that describes the content of the multimedia object. Unfortunately, this approach is not very practical in large multimedia repositories, because in most cases textual descriptions have to be generated manually and are difficult to extract automatically. Also, they are subject to the standards adopted by the person who created them, and cannot encode all the information available in the multimedia object. Another restriction that imposes this approach is that any index processing must "guess" (and thereafter fix) which kind of queries can be posed on the data.

A more promising approach for implementing a similarity search system is a *content-based search*, which uses the multimedia data itself. In this case, the multimedia data is used to perform a similarity query. Figure 1.1 illustrates the concept of content-based similarity search using 3D objects. The figure shows a query object (q) and a set of possible relevant retrieval answers (a): The search system should return, from the database, similar objects to the query object.

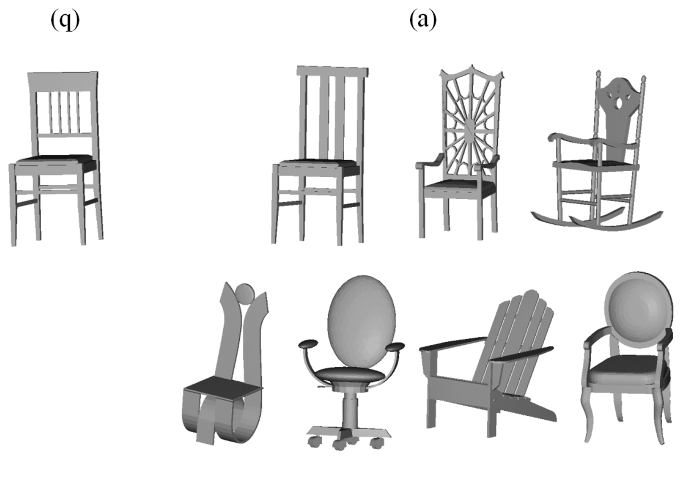

Figure 1.1: Example of a similarity search on a database of 3D objects

To compute the similarity between two multimedia objects, a *distance function* must be defined. This function measures the similarity (or dissimilarity) between two objects. If the distance function holds the properties of a metric (strict positiveness, symmetry, and triangle inequality), then the multimedia objects form a *metric space*. The space of strings together with the *edit distance* (the minimum number of character insertions, deletions, and replacements to make two strings equal) is a classical example of a metric space.

Another approach to model the similarity between multimedia objects is to transform them into points in a *vector space*, which is a particular type of a metric space. Having defined certain object aspects, numerical values are extracted from the multimedia object by using a *transformation function*. These values describe the multimedia object and form a *feature vector* of usually high dimensionality. There are many distance functions that can be defined in vector spaces (e.g., the Euclidean distance).

Both approaches to model similarity have their advantages and disadvantages. In the case of metric spaces, the similarity function usually measures the minimum effort (cost) necessary to transform one object into another (as in the case of the edit distance), thus it needs to solve an optimization problem. This is a formal way of defining similarity between objects but, depending also on the actual multimedia data type, the similarity function may be very complex and may not hold all the properties of a metric (which is important for indexing purposes).

In the case of vector spaces, there are many available metric functions which are easy to compute, and the feature vectors have geometric properties that can be used to improve the indexing of the space. However, it is not always clear which features must be extracted to obtain a good representation of the original data. A practical solution to this problem (that comes from signal processing) resorts to the use of a numerical transformation, such as the Fourier or Wavelet transform. If the data can be represented as a signal, the transform gives the coefficients of a basis function (e.g., sinusoidal basis functions in the case of Fourier transform) whose linear combination produces the original signal. Some of the obtained coefficients are then used to form the feature vectors.

It will depend on the final application which of both models should be used. As explained before, the most practical approach is to model the data as a vector space. However, the metric space approach can be also useful if the similarity measure holds the properties of a metric, because a distance function defined by experts for a specific data type may model similarity better than the feature vector approach.

Once the multimedia data have been modeled either as a metric or a vector space, the similarity search is reduced to a search for close objects or points in the space. There are two typical similarity queries in multimedia databases:

- *Range query*: It returns all objects from the database that are within a given tolerance radius to the query object.

- *k nearest neighbors query*: It returns the k closest objects from the database to the query object.

1.2 Effectiveness and efficiency of similarity search

Two major aspects of similarity search in multimedia databases are effectiveness and efficiency.

Effectiveness is related with the quality of the answer returned by the similarity query. Given a transformation function, a similarity query in multimedia databases is reduced to a search for close points in, for example, a d-dimensional vector space. An effective transformation function should map two similar multimedia objects to two close points in the vector space.

Feature vectors describe particular characteristics of an object based on the nature of the extraction method. It is important to note that different extraction algorithms usually capture different characteristics of the multimedia objects. It is a difficult problem to select some particular feature methods to be integrated into a similarity search system, as we find that not all methods are equally suited for all retrieval tasks. Ideally, a system would implement a set of fundamentally different methods (i.e., a *set of feature vectors*), so that the appropriate feature could be chosen based on the application domain and/or the user preferences.

The specific feature vector type and its given parameterization determine the extraction procedure and the resulting vector dimensionality. In general, different levels of resolution for the feature vector are allowed for each transformation function: More refined descriptors are obtained using higher resolutions. However, higher dimensional feature vectors do not necessarily imply that a better effectiveness will be achieved.

Efficiency is related to the cost of the search (in CPU and I/O time). A naïve method to answer range queries and k nearest neighbors queries is to perform a sequential scan of the database. However, this method may be too slow for real-world applications. Usually, an *index structure* is used to filter out irrelevant objects (or complete zones of the space), thus avoiding the sequential scan.

Several index structures have been proposed for metric and vector spaces. *Metric access methods* [Chávez et al., 2001b] are index structures that use the metric properties of the distance function (especially the triangle inequality) to filter out zones of the space. There are two main classes of metric access methods: *Pivot-based indices* and *indices based on compact partitions*. Both classes of indices aim to divide the space into equivalence classes, and then the index is used to filter out some of the classes at query time. The equivalence classes that could not be discarded must be exhaustively checked for relevant objects.

Spatial access methods [Böhm et al., 2001] are index structures especially designed for vector spaces. Together with the metric properties, spatial access methods use geometric information to discard points from the space. Usually, these indices are hierarchical data structures that use a balanced structure to index the database.

Almost all of these index structures (metric and spatial access methods) have been designed to index single feature vectors, and do not support the use of sets of feature vectors.

1.3 Overview of the thesis

The work presented in this thesis concerns the development and analysis of novel algorithms and data structures that improve the efficiency of similarity search in multimedia databases. As the effectiveness aspects of similarity search are as important as its efficiency aspects, this thesis also propose methods for improving the effectiveness of similarity search. The proposed methods pose new questions and challenges to the efficiency aspects, which are discussed in this thesis.

The thesis starts with an introductory chapter (Chapter 2) that describes in detail the basic notions of similarity search in multimedia databases. It also presents some related work in metric access methods, spatial access methods, and probabilistic algorithms for similarity search in metric and vector spaces. The rest of the thesis is divided in three chapters: Effectiveness of similarity search (Chapter 3), efficiency of similarity search (Chapter 4), and hardware accelerated database algorithms (Chapter 5).

Chapter 3 concentrates on how to improve the effectiveness of similarity search in multimedia databases:

- We present an experimental evaluation on a large set of descriptors for 3D model retrieval (see Sections 3.1 and 3.2), and conclude that on average there are effective 3D descriptors. However, we also conclude that there are no single feature that can dominate over all model classes, that the best descriptor to use depends on the query object, and that the effectiveness of the similarity search may be improved by using *combinations of feature vectors.*

- We propose methods for dynamically selecting and combining feature vectors (see Section 3.3), which may significantly enhance the effectiveness of the similarity search. We describe the *purity* and *entropy impurity* methods, which assess a priori the suitability of a feature vector depending on the query object, and show how to use these methods to select and combine feature vectors.

Chapter 4 describes advanced techniques for searching in metric spaces and indexing methods for combinations of feature vectors:

- We study *pivot selection techniques* for pivot-based indices (see Section 4.1). We provide an efficiency criterion based on distance histograms for selecting good set of pivots. Based on the proposed efficiency criterion, we describe several optimization techniques that allows us to find good sets of pivots. We present empirical evidence showing that the proposed pivot selection techniques are effective.

- We describe an *improved k nearest neighbor algorithm* (see Section 4.2), which is based on the best-first traversal algorithm proposed by Hjaltason and Samet [1995]. The improved algorithm aims to lower the space requirements by using distance estimators. The proposed technique is general and can be used with databases modeled as metric or vector spaces.

- We present a *metric access method for dynamic combinations of feature vectors* (see Section 4.3). The index is pivot-based, and it can take advantage of the previously studied pivot selection techniques. The proposed index structure can support similarity queries that use dynamically weighted combinations of feature vectors.

- We introduce an approach that aims to minimize the expected search cost of a similarity query that uses combinations of feature vectors (see Section 4.4). The idea is to construct a *set of indices* that indexes only the *most frequently used combinations.* If there are restrictions on the available space for constructing indices, the resulting optimization problem can be modeled as a binary linear program. As binary linear programs are NP-hard in the general case, we also propose algorithms that quickly find good sets of indices. The proposed approach is flexible in the sense that is not restricted to a particular dimensionality of the space, to a particular index structure, or to specific cost functions.

Finally, Chapter 5 presents *hardware accelerated algorithms using the graphics processor unit (GPU)* for multimedia databases. GPU technology has improved much faster than CPU technology in the last few years, due to the rapid development of computer games and other multimedia applications, which require extensive graphic processing capabilities to render realistic scenes in real time. It is therefore a good idea to exploit the power of current GPUs to speed up general computations. We depict GPU implementations of a high-dimensional nearest neighbor search and a clustering algorithm. We experimentally evaluate the proposed GPU algorithms, showing that they are an order of magnitude faster than their CPU versions.

The ideas presented in this thesis have been published in Bustos et al. [2003, 2004a,b,c, 2005a,b, 2006a,c], or have been submitted for publication [Bustos et al., 2006b; Bustos and Navarro, 2006].

Chapter 2

Basic concepts and related work

All applications that perform similarity search in multimedia databases need to find objects that are close to each other. The closeness between two multimedia objects is generally defined in terms of a *dissimilarity function*. In this chapter, we describe how multimedia objects can be modeled, either as a metric or a vector space, to implement similarity queries. We present several access methods for multimedia databases and probabilistic algorithms for similarity queries.

2.1 Modeling multimedia data

Given the heterogeneous nature of multimedia databases, the use of mathematical models and tools is necessary to abstract the multimedia data types from the implementation of a similarity search. This section describes two widely used models for multimedia data: *metric spaces* and *vector spaces*.

2.1.1 Metric spaces

Let $\mathbb{X}$ be the universe of valid multimedia objects for a given application (e.g., images). Let $\delta : \mathbb{X} \times \mathbb{X} \to \mathbb{R}^+$ be a function between pairs of objects that returns a positive real value. The pair $(\mathbb{X}, \delta)$ represents a *metric space* if and only if δ holds the following properties:

- *Symmetry*: $\forall x, y \in \mathbb{X},\ \delta(x, y) = \delta(y, x)$.
- *Reflexivity*: $\forall x \in \mathbb{X},\ \delta(x, x) = 0$.

- *Strict positiveness*: $\forall x, y \in \mathbb{X},\ x \neq y \Rightarrow \delta(x, y) > 0$.
- *Triangle inequality*: $\forall x, y, z \in \mathbb{X},\ \delta(x, z) \leq \delta(x, y) + \delta(y, z)$.

The function δ is called the *distance* or *metric* of $\mathbb{X}$. In metric spaces, multimedia objects are directly compared using δ, which can be seen as a "black box" function that indicates the degree of dissimilarity between two objects. The smaller the value given by δ, the more similar are the two given objects.

An example of a metric space is the space of *strings* with the *edit distance*. A string can be defined as a finite sequence of characters, and the edit distance $edit(x, y)$ (also known as *Levenshtein distance*) is the minimum number of insertions, deletions, or substitutions required to transform x into y. Another important example of metric spaces are *vector spaces*.

2.1.2 Vector spaces

A *vector space* $\mathbb{R}^d$ is a particular type of metric space. The space $\mathbb{R}^d$ is composed by d-tuples of real numbers, which are called *vectors*. That is, if $x \in \mathbb{R}^d$ then $x = (x_1, \ldots,\ x_d),\ x_i \in \mathbb{R}, 1 \leq i \leq d$.

There are many metrics for vector spaces. A widely used family of distances is the *Minkowski distance* (L_p), which is defined as

$$L_p(x, y) = \left(\sum_{i=1}^{d} |x_i - y_i|^p \right)^{1/p},\ p \geq 1.$$

Some examples of Minkowski metrics are:

- *Manhattan distance* ($p = 1$): $L_1(x, y) = \sum_{i=1}^{d} |x_i - y_i|$.
- *Euclidean distance* ($p = 2$): $L_2(x, y) = \sqrt{\sum_{i=1}^{d} |x_i - y_i|^2}$.
- *Maximum distance* ($\lim_{p \to \infty} L_p$): $L_\infty(x, y) = \max_{i=1}^{d} |x_i - y_i|$.

Another well-known metric for vector spaces is the *Mahalanobis distance*, defined as

$$\delta(x, y) = \sqrt{(x - y)^T C^{-1} (x - y)},$$

where C is the covariance matrix of the distribution of the vectors and T denotes the transpose operator. If C is the identity matrix, then the Mahalanobis distance is equivalent to the Euclidean distance.

Figure 2.1 illustrates the shapes induced by different metrics on vector spaces. All the points located on the perimeter of each shape are at the same distance from its respective center.

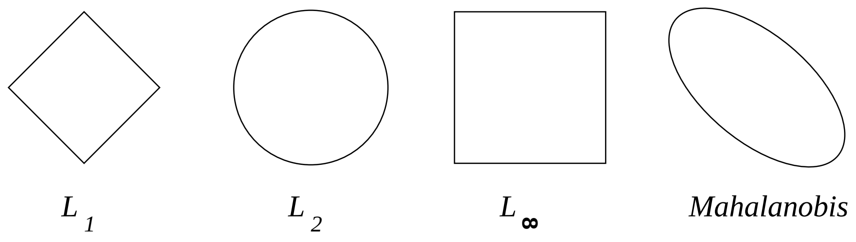

Figure 2.1: Shape of the unitary cube for different metrics on vector spaces

To model multimedia data as a vector space, a *transformation function* must be used, which is highly dependent on the multimedia data type. This function extracts important features from the multimedia objects and maps these features into d-dimensional *feature vectors* (also referred to in the literature as *descriptors*). Thus, a similarity query in the original space is reduced to a search for close points in a d-dimensional vector space.

The *feature-based* approach for similarity search is illustrated in Figure 2.2. Through this thesis, we will refer to a "feature vector" (FV) as any well-defined feature transformation method, and the actual vectors obtained with this function will be called "points" or "objects".

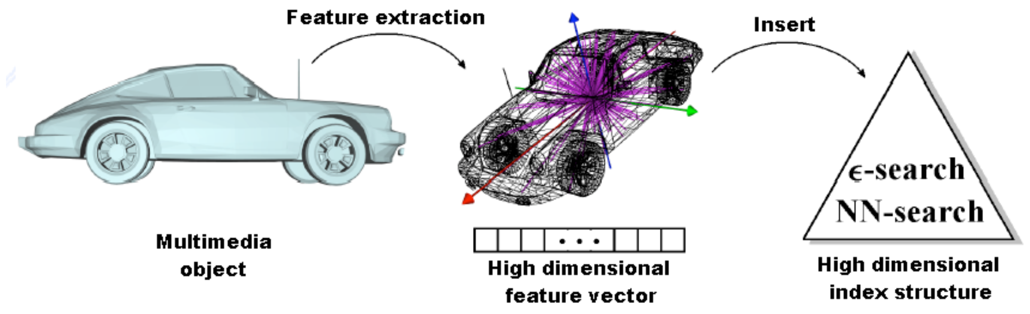

Figure 2.2: Feature-based similarity search

Usually, the dimensionality d of the FV is a parameter of the transformation function: By using higher values of d one can obtain a better (finer) representation of the multimedia object. However, in practical applications there is usually a *saturation point*, where adding more dimensions only adds noise to the description (see Chapter 3). For most applications, the transformation is *irreversible*, i.e., it is not possible to reconstruct the original multimedia object from the description.

2.2 Similarity queries

Let $\mathbb{U} \subset \mathbb{X}$ be a set of multimedia objects (i.e., an instance of a multimedia database). There are two typical similarity queries in multimedia databases:

- *Range query.* A range query (q, r), $q \in \mathbb{X}$, $r \in \mathbb{R}^+$, reports all database objects that are within a distance r to q, that is, $(q, r) = \{u \in \mathbb{U}, \delta(u, q) \leq r\}$. The subspace $\mathbb{V} \subset \mathbb{X}$ defined by q and r (i.e., $\forall v \in \mathbb{V}\ \delta(v, q) \leq r$ and $\forall x \in \mathbb{X} - \mathbb{V}\ \delta(x, q) > r$) is called the *query ball.*

- *k nearest neighbors query (k-NN).* It reports the k objects from $\mathbb{U}$ closer to q. That is, it returns the set $\mathbb{C} \subseteq \mathbb{U}$ such that $|\mathbb{C}| = k$ and $\forall x \in \mathbb{C}, y \in \mathbb{U} - \mathbb{C}, \delta(x, q) \leq \delta(y, q)$.

Note that it is possible that many sets of k objects are a valid answer for the k-NN search. For example, if there are two or more objects at exactly the same distance to the k^{th} NN, any of them can be selected as the k^{th} NN. While this is unusual when using continuous distance functions, it is frequent when using discrete distance functions, such as the edit distance.

Figure 2.3 illustrates a range query (left) and a k-NN query (right) with $k = 3$ in a 2-D vector space using the Euclidean distance. The answers are respectively: $(q, r) = \{u_2, u_4, u_6, u_7\}$; $\mathbb{C} = \{u_2, u_6, u_7\}$. Note that if one knew a priori the distance from q to the k^{th} nearest neighbor (denoted by u^k), then the range query $(q, \delta(q, u^k))$ would return the same set of objects as the k-NN query (assuming that the answer set for the k-NN query is unique).

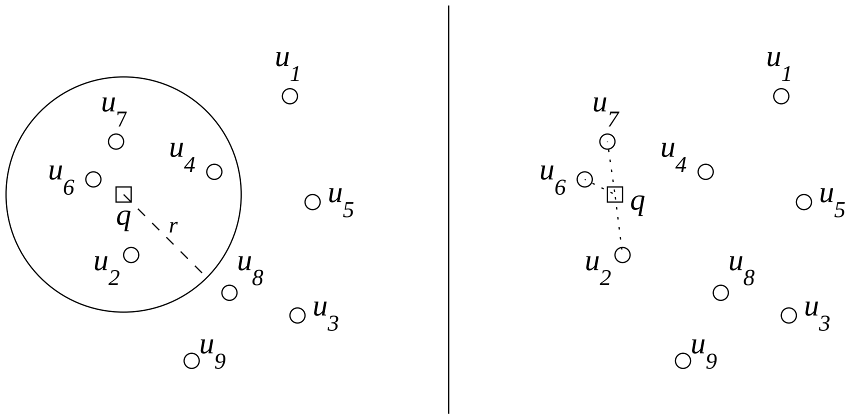

Figure 2.3: Range query and k-NN query examples

Both similarity queries can be answered by performing a linear scan on the database, which takes $O(n)$ time with $n = |\mathbb{U}|$. However, this naïve algorithm may be very slow if the database is too large and/or the distance function is computationally too expensive. This is the main motivation for the research and development of index structures for multimedia databases. The goal of similarity search algorithms is to build an *index* of the database in advance and later perform similarity queries using this index, avoiding a full scan of the database.

2.3 Metric access methods

Metric access methods (MAMs) are index structures and algorithms designed to perform efficient similarity queries in metric spaces. They only use the metric properties of δ, especially the triangle inequality, to filter out objects or entire regions of the space during the search, thus avoiding the linear scan. In general, the function δ is considered expensive to compute, and in many practical applications δ is so costly that the extra CPU time or even I/O time costs can be neglected. For this reason, the complexity of MAMs is generally measured as the *number of distance computations performed.* See Chávez et al. [2001b] for a survey on metric access methods.

Figure 2.4 shows a model for indexing and querying metric spaces. All index structures partition the data into subsets, each of them representing an *equivalence class*, i.e., a subset of objects related somehow. When performing a similarity query, the search algorithm filters out some of the classes. Those classes not discarded by the index must be checked for relevant objects. The cost associated with the index traversal is the *internal complexity* of the search, and the cost of examining the non-discarded equivalence classes is the *external complexity* of the search.

Almost all metric access methods proposed up to now store the index structure in main memory, making them not suitable for very large metric databases. Some exceptions are the *M-tree* (see Section 2.3.2) and the *D-Index* [Dohnal et al., 2003].

MAMs can be classified into two main groups: Indices based on pivots (see Section 2.3.1) and indices based on compact partitions (see Section 2.3.2).

2.3.1 Pivot-based indexing

Pivot-based indices select a number of *pivot* objects from the database, and classify all the other objects according to their distance from the pivots. The canonical pivot-based range query algorithm works as follows: Given a range query (q, r) and a set of t pivots $\{p_1, \ldots, p_t\}, p_i \in \mathbb{U}$, by the triangle inequality it follows for any $x \in \mathbb{X}$ and $1 \leq i \leq t$ that $\delta(q, x) \geq |\delta(p_i, x) - \delta(p_i, q)|$. The

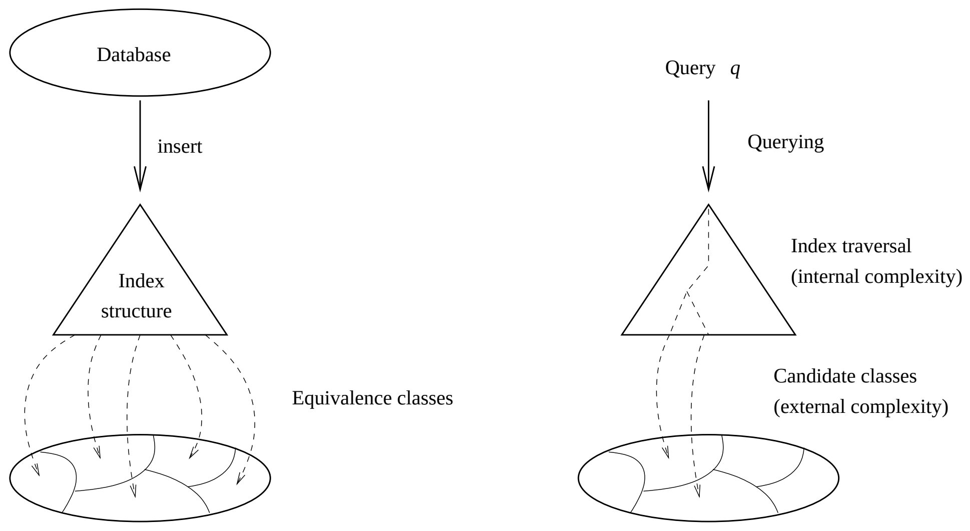

Figure 2.4: Model for indexing and querying metric spaces

objects $u \in \mathbb{U}$ of interest are those that satisfy $\delta(q, u) \leq r$, so one can exclude all the objects that satisfy $|\delta(p_i, u) - \delta(p_i, q)| > r$ for some pivot p_i, without actually evaluating $\delta(q, u)$. This discarding criterion is known as the *pivot exclusion condition*.

The index consists of the tn precomputed distances $\delta(p_i, u)$ between every pivot and every object of the database. At query time it is necessary to compute the k distances between the pivots and q, $\delta(p_i, q)$, to apply the exclusion condition. Those distance calculations are known as the *internal complexity* of the algorithm, and this complexity is fixed if there is a fixed number of pivots. The list of objects $\{u_1, \ldots, u_m\} \subseteq \mathbb{U}$ that cannot be discarded with the exclusion condition, known as the *object candidate list*, must be checked directly against the query. Those distance calculations $\delta(u_i, q)$ are known as the *external complexity* of the algorithm.

Figure 2.5 shows an example of the pivot exclusion condition in a 2-D Euclidean space. The left figure describes the zone of non-discarded objects using pivot p: All elements inside the ring cannot be discarded. The right figure shows the effect of using more pivots: Only those elements belonging to the intersection of the rings cannot be discarded.

The total complexity of the search algorithm is the sum of the internal and external complexity, $t + m$. Since one increases and the other decreases (or at least does not increase) with t, it follows that there is an optimum t^* that depends on the tolerance range of the query, r. In practice, however, t^* is so large that one cannot store the t^*n distances, and the index simply uses as many pivots as space permits.

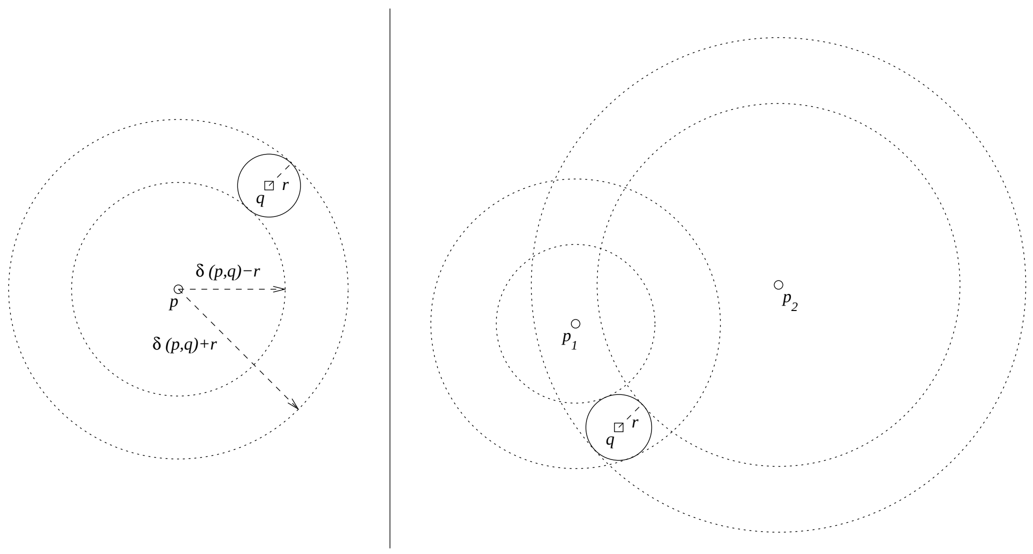

Figure 2.5: Pivot exclusion condition

Several indices resort to a tree structure to avoid considering the exclusion condition for each $u \in \mathbb{U}$ individually. Each tree node p is a pivot and its subtrees correspond to subsets of objects that are inside some given range of distances from p. At search time, $\delta(q,p)$ is computed and the search algorithm only needs to enter tree branches whose ranges of distances intersect $[\delta(q,p) - r, \delta(q,p) + r]$.

Pivot-based indices for discrete distance functions

The *Burkhard-Keller tree (BKT)* [Burkhard and Keller, 1973] is an index structure designed for discrete distance functions. A node p is randomly selected from $\mathbb{U}$. For each distance $i > 0$, the subset s_i of objects which are at distance i from p is computed. A BKT is recursively constructed on each nonempty subset s_i, and the roots of these subtrees conform the children of p. Other tree structures for discrete distance functions are the *Fixed-Queries tree (FQT)* and the *Fixed-Height FQT* [Baeza-Yates et al., 1994]. The FQT is a variant of the BKT, where only one pivot p is the root of all subtrees from the same level. The Fixed-Height FQT fixes the FQT height at a given level h, independently of the bucket sizes at the leaves. The *Fixed-Queries Array (FQA)* [Chávez et al., 2001a] compactly represents a Fixed-Height FQT in an array. This permits to save space, thus one can use more pivots, at the cost of increasing the CPU time by a factor of $O(\log(n))$.

Pivot-based indices for continuous distance functions

The *Vantage Point Tree (VPT)* [Yianilos, 1993] is designed for metric spaces with continuous distance functions. The VPT is a binary tree, where its root is an object p randomly selected from $\mathbb{U}$. Then, the median M of the set of all distances between p and objects from $\mathbb{U}$ is computed. All objects $u \in \mathbb{U}$ with $\delta(p, U) \leq M$ are inserted in the left subtree, and the rest are inserted in the right subtree. For each subtree, a new pivot is selected and a VPT is recursively constructed. Given a range query (q, r), the search algorithm computes $\delta(p, q)$ and adds p to the result if it is within the query ball. If $\delta(p, q) - r \leq M$ the search recursively continues on the left subtree, and if $\delta(p, q) + r > M$ the search recursively continues on the right subtree. Note that it is possible that both subtrees must be traversed. Extensions to the VPT are the *Multi-Vantage Point Tree* [Bozkaya and Ozsoyoglu, 1997] and the *Vantage Point Forest* [Yianilos, 1999].

Other pivot-based indices

The *Approximating and Eliminating Search Algorithm (AESA)* [Vidal, 1986] builds an $O(n^2)$ matrix with all precomputed distances between objects from $\mathbb{U}$, i.e., it uses all database objects as pivots. This index structure has in practice a very good performance, but its $O(n^2)$ space cost makes it only suitable for very small databases. *Linear AESA (LAESA)* [Micó et al., 1994] only uses t fixed pivots, thus its space cost is $O(n)$, and it is basically a direct implementation of the canonical pivot-based search algorithm. A data structure called *Spaghettis* [Chávez et al., 1999] aims to reduce the CPU time used to compute the set of non-discarded objects. The main data structure is also a matrix with the distances between pivots and objects, but now the distances for each pivot p are sorted. Given a range query, the set of non-discarded objects by p can be efficiently computed using binary search. This data structure uses extra pointers to link the same object on the different lists of sorted distances.

Other index structures that use the pivot exclusion condition are the *OMNI family of access methods* [Santos-Filho et al., 2001], which presents a general approach for implementing the pivot exclusion condition on top of existing access methods, and the *D-Index* [Dohnal et al., 2003], which combines clustering techniques with the pivot-based approach.

2.3.2 Indices based on compact partitions

These algorithms are based on dividing the space into *partitions* or *zones* as compact as possible. Each zone stores a representative point, called the *center*, and data that can be used to discard the entire zone at query time, without measuring the actual distance from the zone objects to the query object. Each zone can be partitioned recursively into more zones, inducing a *search hierarchy.* There are two general criteria for partitioning the space: Voronoi partition and covering radius.

The *Voronoi diagram* of a collection of objects is a partition of the space into cells. Each cell contains the objects closer to one particular center than to any other. A set of m centers is selected and the rest of the objects are assigned to the zone of their closest center. Given a range query (q, r), the distances between q and the m centers are computed. Let c be the closest center to q. Every zone of center $c_i \neq c$ which satisfies $\delta(q, c_i) > \delta(q, c) + 2r$ can be discarded, because its Voronoi area cannot intersect with the query ball. Figure 2.6 shows an example of the Voronoi partition criterion. For q_1 the zone of c_4 can be discarded, and for q_2 only the zone of c_4 must be examined.

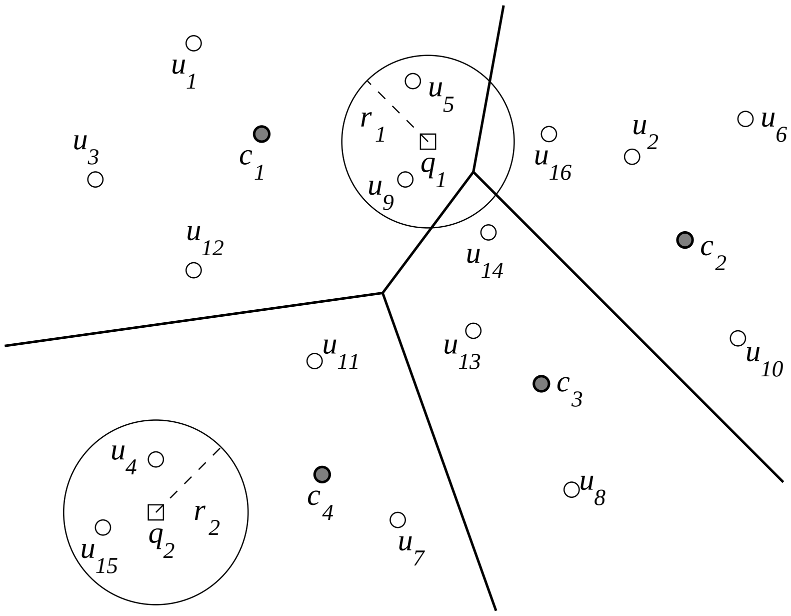

Figure 2.6: Voronoi partition criterion

The *covering radius* $cr(c)$ is the maximum distance between a center c and an object that belongs to its zone. Given a range query (q, r), if $\delta(q, c_i) - r > cr(c_i)$ then zone i cannot intersect with the query ball and all its objects can be discarded. In Figure 2.7, the query ball of q_3 does not intersect with the zone of center c, thus it can be discarded. For the query balls of q_1 and q_2, the zone cannot be discarded, because it intersects these balls.

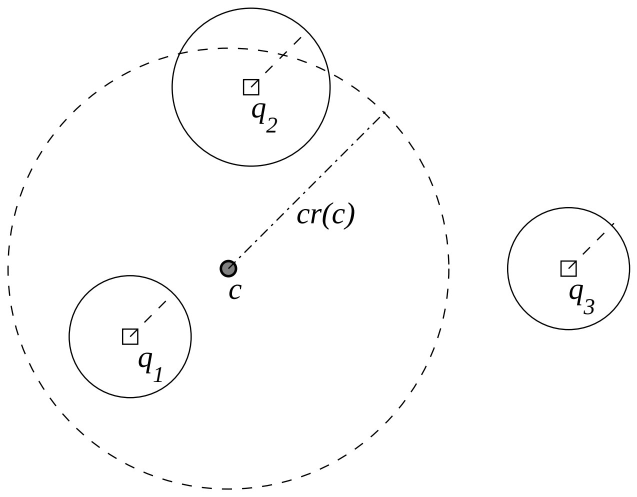

Figure 2.7: Covering radius criterion

The *Generalized-hyperplane tree* [Uhlmann, 1991b] is an index structure that uses the Voronoi partition criterion. Indices that use the covering radius criterion are the *Bisector tree* (*BST*) [Kalantari and McDonald, 1983], the *Voronoi tree* [Dehne and Noltemeier, 1987], the *Monotonous bisector tree* [Noltemeier et al., 1992], the *M-tree* [Ciaccia et al., 1997], the *PM-tree* [Skopal et al., 2005], and the *List of clusters* [Chávez and Navarro, 2005]. There exist indices that use both criteria, for example the *Geometric near-neighbor access tree* (*GNAT*) [Brin, 1995] and the *Spatial approximation tree* [Navarro, 2002]. From these methods, descriptions for the M-tree and the List of clusters are given below.

M-tree

The *M-tree* [Ciaccia et al., 1997] is a dynamic index structure that provides good performance in secondary memory. This index is similar to the GNAT in the sense that it is a tree where some points are selected as centers and the rest of the objects are (generally) assigned to the zone of its closest center. Each zone (branch of the tree) is then recursively indexed with an M-tree. Unlike the GNAT, the M-tree uses the covering radius criterion to filter out branches while performing a similarity query. The data is stored in the leaves of the M-tree, and the internal nodes store the so-called *routing objects*.

A new object u is inserted in the best subtree, which is defined as the one where the covering radius must increase the least in order to cover the new object. In case of ties, the subtree whose center is closest to u is selected. The insertion algorithm proceeds recursively until a leaf is reached and u is inserted in that leaf, storing the distance to the center of its parent node. Node overflows are managed in a similar way as in the B-tree. If an insertion produces an overflow, two objects from the node are selected as new centers, the node is split, and the two new centers are promoted to the parent node (different splitting techniques are studied in Ciaccia et al. [1997]). If the parent node overflows, the same split procedure is applied. If the root overflows, it is split and a new root is created. Thus, the M-tree is a balanced tree. Figure 2.8 illustrates an example of an M-tree.

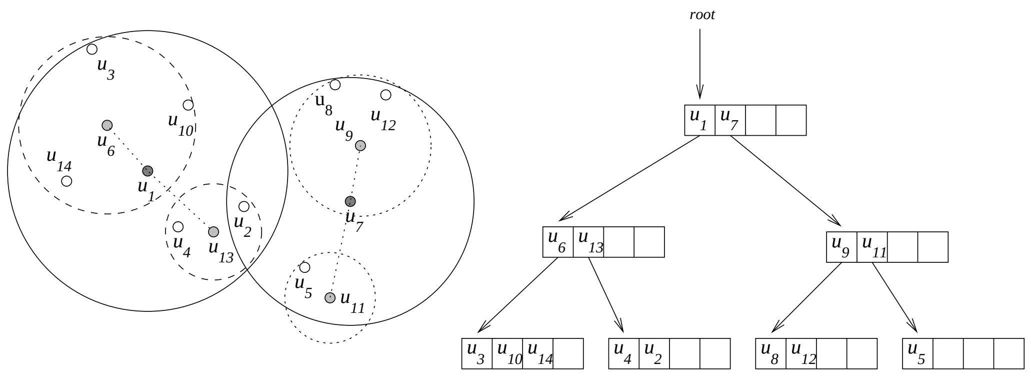
root
u_1 u_7
u_6 u_{13}
u_9 u_{11}
u_3 u_{10} u_{14}
u_4 u_2
u_8 u_{12}
u_5
u_3 u_{10} u_6 u_{14} u_1 u_2 u_4 u_{13} u_8 u_9 u_{12} u_7 u_5 u_{11}

Figure 2.8: Example of an M-tree

Range queries are implemented by traversing the tree, starting from the root. As each node o in the tree contains the distances from the routing objects to the center of its parent node p (except for the root), a similar criterion as the pivot exclusion condition can be used to further filter out branches. In this case, the parent node p is used as a pivot. A subtree of o rooted by o_{ro} (one of the routing objects) can be safely filtered out if $|\delta(p, q) - \delta(o_{ro}, p)| > r + cr(o_{ro})$. If the branch cannot be filtered out, the actual distance between o_{ro} and q is computed. Then, if $\delta(o_{ro}, q) - cr(o_{ro}) > r$ (covering radius criterion) the branch can be filtered out, otherwise the search continues recursively on this branch.

List of Clusters

The *List of Clusters* [Chávez and Navarro, 2005] is a list of "zones". Each zone has a center and stores its covering radius. A center $c \in \mathbb{U}$ is chosen as well as a radius rp, whose value depends on whether the number of objects per compact partition is fixed or not. The *center ball* of (c, rp) is defined as $(c, rp) = \{x \in \mathbb{X}, \delta(c, x) \leq rp\}$. The set $I = \mathbb{U} \cap (c, rp)$ is defined as the bucket of "internal" objects lying inside (c, rp), and the set $E = \mathbb{U} - I$ is defined as the rest of the objects (the "external" ones). The process is repeated recursively inside E. The construction process returns a list of triples (c_i, rp_i, I_i) (center, radius, internal bucket). For the selection of c, it is suggested in Chávez and Navarro [2005] to select as next center the object that maximizes the sum to the previously selected centers.

This data structure is asymmetric, because the first center chosen has preference over the next centers in case of overlapping balls. Figure 2.9 illustrates the asymmetry of the data structure. With respect to the value of the radius rp of each compact partition and the selection of the next center in the list, there are many alternatives. Chávez and Navarro [2005] experimentally showed that the best performance is achieved when the compact partition has a fixed number of objects, so rp simply becomes $cr(c)$, and the next center is selected as the object which maximizes the distance sum to the centers previously chosen. The brute force algorithm for constructing the list takes $O(n^2/m)$, where m is the size of the compact partition, but it can be improved using auxiliary data structures to build the partitions. For metric spaces with high intrinsic dimensionality (see Section 2.5.1), the optimal m is very small.

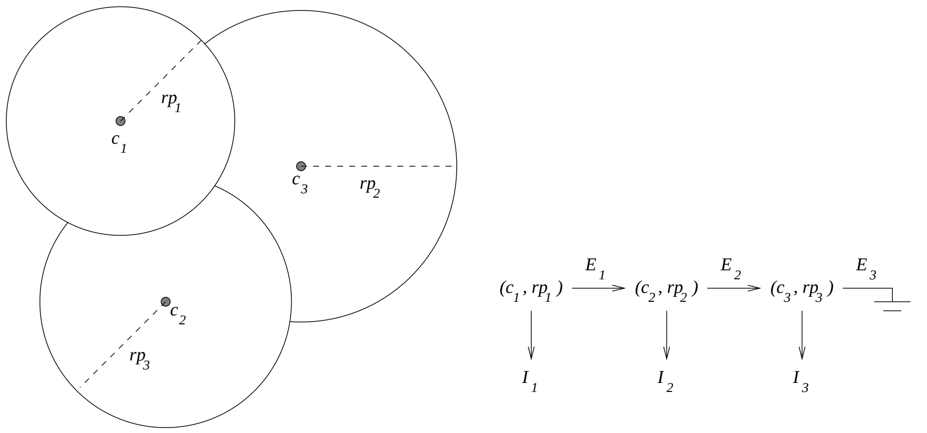

Figure 2.9: Example of a List of Clusters

Given a range query (q, r), the search algorithm computes $\delta(q, c)$, reporting c if it is within the query ball. Then, the algorithm searches exhaustively inside I only if $\delta(q, c) - cr(c) \leq r$. The rest of the list, E, is processed only if $cr(c) - \delta(q, c) < r$, because of the asymmetry of the data structure. The search cost has a form close to $O(n^\alpha)$ for some $0.5 < \alpha < 1.0$ [Chávez and Navarro, 2005].

2.4 Spatial access methods

Spatial access methods (SAMs) are index structures especially designed to index data modeled as vector spaces. These index structures uses more information as MAMs (e.g., geometric properties of the data points), thus providing better structures for efficiently handling similarity queries in these spaces. See Böhm et al. [2001] and Gaede and Günther [1998] for surveys on spatial access methods.

SAMs hierarchically cluster data pages, usually by means of a balanced directory. Figure 2.10 illustrates the basic structure of a SAM. The *root node* is a single directory node and corresponds to the first hierarchy level. All but the last level of the search hierarchy are conformed by *directory pages*. Each node of the directory pages spatially encloses all data points under its subtree. The last level of the hierarchy corresponds to the *data pages*, which contain the actual data points.

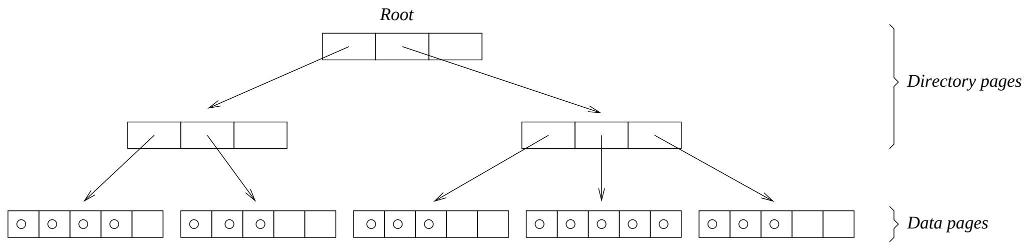

Figure 2.10: Hierarchical index structure of spatial access methods

2.4.1 Examples of spatial access methods

R-tree and its variants

The *R-tree* [Guttman, 1984] is a balanced tree that uses *minimum bounding rectangles* (MBRs) as page regions. Given a vector space $\mathbb{R}^d$, an MBR $mbr \subset \mathbb{R}^d$ is the minimal orthogonal hyperrectangle that encloses a set of given points. Thus, all $(d-1)$-dimensional surfaces of mbr contains at least one data point. The space partitioning is neither complete nor disjoint. The MBRs are allowed to overlap, although this can reduce the overall efficiency of the index structure. Each node of the tree is allowed to contain between m and M objects, with $m \leq M/2$, except for the root node.

Figure 2.11 shows an example of an R-tree in a 2-dimensional vector space. For this example, $M = 5$ and $m = 2$. Each MBR is denoted by R_i, $1 \leq i \leq 7$, and the datapoints are denoted by u_j, $1 \leq j \leq 14$. Note that all surfaces of the MBRs contain at least one data point.

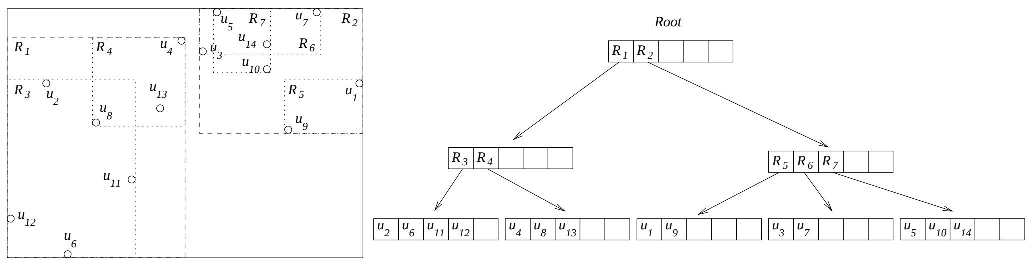

Figure 2.11: Example of an R-tree

New points are inserted in the "best suited" data page: If the point is contained in exactly one data page, then it is inserted there; if the point is contained in several page regions, the one with the smallest volume is selected; if no region contains the point, the one with smallest volume enlargement is selected. In case of ties, the algorithm selects the region with the minimum volume. When a node overflows after an insertion, it must be split into two, thus new MBRs must be computed and the parent node must also be updated. If the parent node overflows, the procedure is recursively applied. If the root overflows, it is split and a new root is created. There are different splitting policies in case of overflow. Guttman [1984] depicts a quadratic and a linear algorithm, and recommends to use the latter.

The *R*-tree* [Beckmann et al., 1990] is an extension of the R-tree. The basic data structure is the same, but the algorithms for splitting regions and inserting new points are improved, and the concept of *forced reinsertion* is presented.

The new insertion algorithm differentiates between data and directory pages in case no region contains the new point. In the former case, the criteria to select the best suited data page are the smallest enlargement of the overlap between regions, the volume enlargement, and the volume of the region (in that order). For the directory pages, the region with smallest enlargement is the chosen one, and in case of ties the algorithm selects the region with smaller volume.

The new split algorithm aims to minimize the overlap between page regions and the coverage of empty space. Also, splits are avoided by the concept of forced reinsertion. The idea is to reinsert a percentage of the objects from the overflowed data region. The selected objects for reinsertion are those furthest from the center of the region. Reinsertion has two beneficial results: The storage utilization grows and the quality improves, because bad decisions taken at the beginning, when the tree had only a few points, can be corrected in this way.

X-tree

The *X-tree* [Berchtold et al., 1996] is an extension of the R*-tree, and it is specifically designed to perform well in high dimensional spaces. It extends the R*-tree by providing *overlap-free split* and *supernodes*. It has been observed experimentally that in high-dimensional spaces (more than 10 dimensions according to Berchtold et al. [1996]), the amount of volume of spaces covered by more than one MBR in an R*-tree approaches quickly the whole data space. This is due to the criteria used by the R*-tree to split nodes, which also aim to minimize the volume of the resulting MBRs. The high amount of overlap between MBRs means that, for any similarity query, at least two subtrees must be accessed in almost every directory node, thus reducing the efficiency of the index structure.

To avoid this problem, the X-tree maintains the history of data page splits of a node in a binary tree. The root of this “split history tree” contains the dimension where a overlap-free split is guaranteed (which is a dimension according to which all MBRs in the node have been split previously). Thus, when a directory node overflows, this dimension is used to perform the split. However, the overlap-free split may be unbalanced, i.e., one of the nodes may be almost full and the other one may be underfilled, thus decreasing the storage utilization in the directory. To avoid this problem, the X-tree does not split in this case and creates instead a *supernode*. A supernode is basically an enlarged directory node, which can store more entries than normal nodes. In this way, the unbalanced split is avoided and a good storage utilization is mantained, at the cost of disminishing some of the discriminative power of the index. Figure 2.12 illustrates an X-tree with two supernodes.

VA-file

The *VA-file* [Blott and Weber, 1997; Weber et al., 1998] aims to implement a very fast linear scan of the database. While it is actually not an index as those previously described, it is a very competitive technique, especially in high dimensional spaces.

The main idea of the VA-file is to store bit-compressed versions of the data points. For each dimension i of the vector space, b_i bits are assigned. Thus, the original vectors are quantized to $\sum_{i=1}^{d} b_i$ bits and sequentially stored in a file. The vector quantization is computed using a grid, where dimension i has a resolution equal to 2^{b_i}. The quantiles of the projections of the points to axis i correspond to the grid lines for that dimension. To perform a similarity query, the quantized points are sequentially scanned, discarding the non-relevant points. The original coordinates for non-discarded points

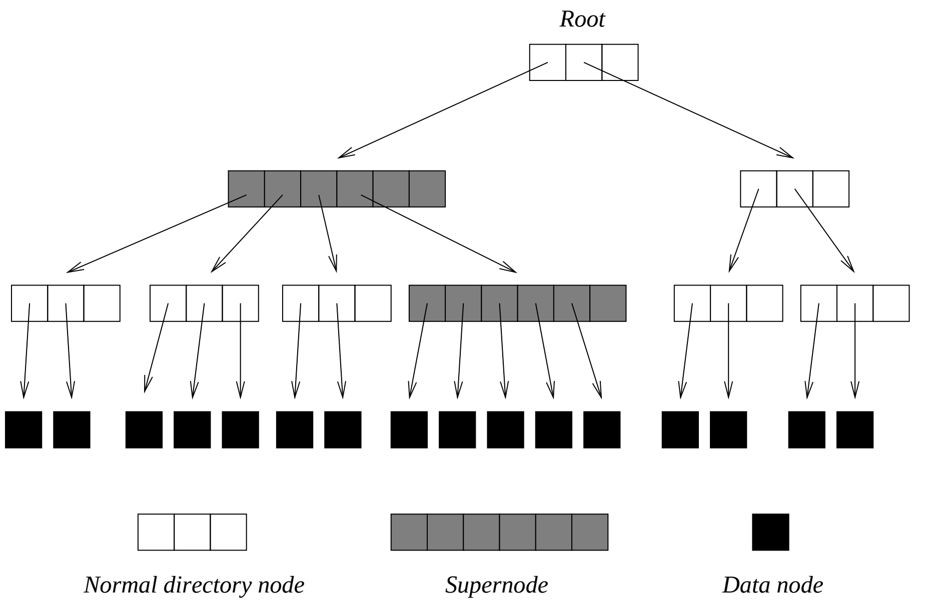

Figure 2.12: Illustration of an X-tree

must be read and compared directly against the query object.

Other spatial access methods

The *kdb-tree* [Robinson, 1981] uses a kd-tree [Bentley, 1975] to create a complete and disjoint partitioning of the space. The page regions are hyperrectangles but they are not MBRs, thus it is possible that a large part of a region is empty. Some advantages of a complete partitioning of the space are that there is always only one path to insert a new object, and that regions can be easily merged maintaining the partitioning of the space. If a page region overflows it is split, the data entries are distributed among the two new nodes, and the split is propagated to the rest of the tree. If lower levels of the tree are also intersected by the split plane, then they must also be split. Thus, an insertion operation costs $O(n)$ in the worst case. The LSD$^{\text{h}}$-tree [Henrich, 1998] is also based on the kd-tree, but it uses a complex region description to reduce the space cost of the index.

The *SS-tree* [White and Jain, 1996b] uses hyperspheres instead of hyperrectangles as page regions, but they are not minimun bounding. The centroid of a set of points is used as the center of the hypersphere, and its radius is always greater than or equal the distance to the farthest point on the set. A new point is inserted in the hypersphere with the closest centroid to it. After an insertion, a new centroid and radius must be computed for the selected hypersphere. When a node overflows, a forced reinsertion is performed, as in the R*-tree. If the children of the selected node have already been reinserted, the node should be split and the axis with the highest variance is selected as the split axis. The chosen split plane is the one that minimizes the sum of variances on each side of the split.

Another index that does not use MBRs as page regions is the *SR-tree* [Katayama and Satoh, 1997]. Instead, it uses the intersection between hyperrectangles and hyperspheres as the page region. The hyperrectangle corresponds to the MBR of the enclosed points, and the hypersphere is the minimum bounding sphere around the centroid that encloses the points. Insertion and split algorithms for the SR-tree are the same as in the SS-tree, only the MBRs must be updated after an insertion or split.

The *Pyramid-tree* [Berchtold et al., 1998] is an index that maps the high-dimensional points into a one-dimensional space, and then uses a B^+-tree to index the data. The space is first partitioned into pyramids, where their top is the centerpoint of the space. Then, each pyramid is cut into slices that are parallel to its basis. A point is mapped to the 1-D space according to the pyramid where it is located and its height (orthogonal distance from the centerpoint to the point). Assuming points uniformly distributed in the space and using the L_{max} distance, it has been shown [Berchtold et al., 1998] that the performance of the Pyramid-tree does not degrade in high-dimensional spaces.

Innovative spatial access methods are proposed on a regular basis. For example: the Δ-tree [Cui et al., 2003] is an index structure optimized for main-memory storage; the ClusterTree [Yu and Zhang, 2003] is a hierarchical representation of data clusters, which can be used to support efficient similarity queries; the iDistance [Yu et al., 2001], the Diagonal Ordering [Hu et al., 2004], and the iMinMax [Yu et al., 2004] propose, like the Pyramid-tree, different approaches to map the data points into a one-dimensional space and use a B^+-tree to index the data.

2.4.2 Similarity queries in spatial access methods

Range queries

The canonical range query algorithm for hierarchical index structures is as follows. Starting from the root, the algorithm performs a depth-first traversal of the index tree. If the actual node is a data page, all points on that data page that are within the query ball are added to the result. If the page is a directory node, the algorithm continues the search on all those children nodes which region pages intersect with the query ball.

Nearest neighbor queries

There are two different approaches for implementing nearest neighbor queries on spatial access methods. The first one is the so-called *RKV* algorithm [Roussopoulos et al., 1995]. It performs a depth-first traversal of the tree, as in the range query algorithm, but in this case there is no fixed criterion to filter out branches from the tree. The optimal criterion would be the actual distance to the NN, but it is unknown at the beginning of the search. Therefore, the algorithm uses conservative estimates of this distance (*closest point distance*) to filter out branches. As the algorithm traverses the tree, more information is gained and the estimates can be improved during the search. One estimate for the distance to the NN is the smallest distance from an already visited point and q, but the RKV algorithm also computes estimates from the page regions visited so far.

First, we need to introduce some concepts to understand the estimation of the NN distance. The *MinDist* value is the minimum distance between a page region and q. For example, if the page regions are minimum bounding spheres, then *MinDist* is equal to the distance from q to the centroid minus the radius of the sphere (the formula in the case of MBRs can be found in Böhm et al. [2001]). The *MaxDist* values is the maximum distance between a page region and q. Finally, the *MinMaxDist* is the maximum distance from the page region to q where a data point is guaranteed to be found. For example, at least one point must be located on each surface of an MBR (cf. Section 2.4.1), thus a better lower bound of the distance from q to a point in the page region than *MaxDist* can be computed. Figure 2.13 illustrates. Note that the upper left corner of the MBR is farther from q than the lower right corner, thus this last one is the geometric point which determines *MinMaxDist* for this example.

The estimation value used in the RKV algorithm is the minimum value between all computed distances to points and all *MinMaxDist* values from page regions processed so far. A branch can be safely filtered out if its *MinDist* is greater than the computed estimation value.

To extend this algorithm to k-NN queries, Roussopoulos et al. [1995] propose to simply discard the use of *MinMaxDist*, and the new estimation value is the k^{th} lowest distance among all computed distances.

The second NN algorithm is the so-called HS algorithm [Uhlmann, 1991a; Hjaltason and Samet, 1995]. It performs a best-first traversal of the index, as page regions are accesed in the order of increasing lower bound distance to q. A priority queue is used to store tree nodes whose parents were already processed. The queue returns always the node with minimum *MinDist* to q.

At the beginning, the priority queue only contains the root node. A node

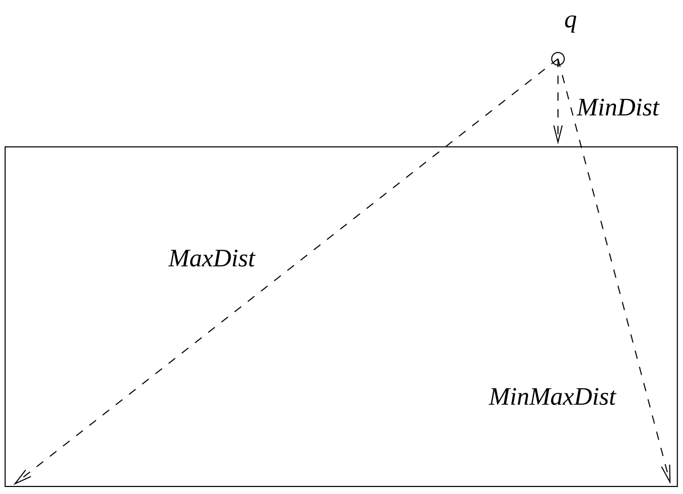

Figure 2.13: MinDist, MaxDist, and MinMaxDist

dp is extracted from the queue and it is processed. If *dp* is a data page, all its points are examined to see if one of them is closer to q than the closest point distance. If this is the case, the candidate NN and the closest point distance are updated. If *dp* is a directory page, then *MinDist* is computed for all its children and they are inserted into the queue. The search stops when when the *MinDist* of the first node in the queue is greater than the current NN distance, which means that no other point can be closer to the query than the current NN candidate (because *MinDist* is a lower bound to the distance from a point to query and the nodes in the queue are sorted by ascending *MinDist*).

For implementing k-NN queries, a second priority queue with the candidate list is needed. As in the case of the RKV algorithm, the new estimation value is the k^{th} lowest distance among all computed distance to points, which results to be the maximum distance from q to one of the points stored in the second queue.

Space complexity of RKV and HS algorithms

As the RKV algorithm performs a depth-first tree traversal and the index tree is balanced, the worst case space complexity for the RKV algorithm is $O(\log(n))$. In contrast, the worst case space complexity of the HS algorithm is $O(n)$, because it is possible that all data pages are loaded into the priority queue before the first data point is found. In spite of this drawback, the HS algorithm has some nice optimality properties, for example in the number of page accesses (see Section 4.2 for more details).

2.5 Efficiency considerations

2.5.1 Intrinsic dimension of a metric space

Estimating the *intrinsic dimensionality* of a metric space (i.e., how “difficult” is to perform a similarity search in that space) is a subject of intense research. One proposal [Chávez et al., 2001b] defines the intrinsic dimension ρ of a metric space $(\mathbb{X}, \delta)$ as

$$\rho = \frac{\mu^2}{2\sigma^2}$$

where μ and σ^2 are respectively the mean and the variance of the distance histogram of $(\mathbb{X}, \delta)$. This definition is coherent with the notion of dimensionality in a vector space, where the coordinate values of its vectors are uniformly distributed. A d-dimensional vector space under the L_p metric has an intrinsic dimensionality of $\Theta(d)$ [Yianilos, 1999; Chávez et al., 2001b].

It follows that the intrinsic dimension of a metric space grows with μ and decreases with σ^2. This means that in spaces with high intrinsic dimensionality the objects tend to be “far away” from each other.

2.5.2 Fractal dimension of a vector space

A vector space $\mathbb{R}^d$ has, by definition, a space dimensionality equal to d. That is, each vector of the space contains d attributes. This value is also known as the *embedding dimensionality* of the space. However, the “real” (intrinsic) dimensionality of the space can be much lower. For example, if all the objects on a 10-dimensional vector space are located in a plane, then the intrinsic dimensionality of the space is only 2. This means that one could map the objects from the 10-dimensional space to a 2-dimensional space while preserving all the distances between objects from the original space. The *intrinsic dimensionality* of a vector space can be estimated by using the concept of *fractal dimension* [Faloutsos and Kamel, 1994].

A *fractal* is a point set which is *self-similar*, i.e., it can be decomposed in parts that are similar to the whole point set. A classic example of a fractal is the *Sierpinski triangle* (see Figure 2.14), which is a fractal consisting of three smaller pieces that are similar to the whole fractal. Each piece is scaled-down by a factor of two.

Given a fractal with r self-similar pieces and with a scale down factor s, its *fractal dimension* $d_{fractal}$ is defined as

$$d_{fractal} = \frac{\log(r)}{\log(s)}.$$

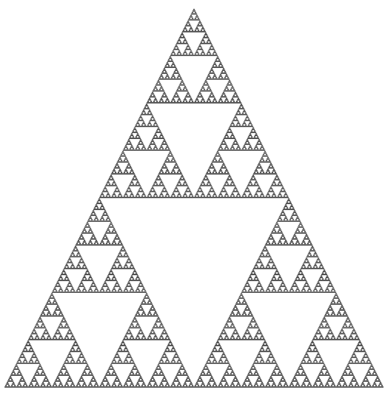

Figure 2.14: Sierpinski triangle

An alternative way to define the fractal dimension [Schroeder, 1991] is as follows: If the space is divided into hypercubic grid cells of side r, and $N(r)$ is the number of cells that contain 1 or more points of the fractal, then the fractal dimension is defined as:

$$d_{fractal} = \lim_{r \to 0} \frac{\log(N(r))}{\log(1/r)}.$$

It has been shown that several distributions of real data exhibit fractal behavior, and the fractal dimension has been used to estimate the performance of index structures that store this data [Faloutsos and Kamel, 1994].

2.5.3 Curse of dimensionality

It is analytically and experimentally shown in Chávez et al. [2001b] that the efficiency of similarity queries in metric spaces systematically degrades in metric spaces with higher intrinsic dimensionalities. This fact is known as the *curse of dimensionality*, and it occurs due to two main effects. Firstly, the distance histograms in metric spaces with high intrinsic dimension have low variance, i.e., all their mass is concentrated near the mean of the distribution. Secondly, to retrieve a fixed fraction of the database, the tolerance radius of a range query must be bigger for metric spaces with high dimensionality, because in these spaces the objects of the database are far away from each other. In general, both effects occur simultaneously.

This means that, in spaces with high intrinsic dimensionalities, no index structure can filter out many objects during a search. For example, Figure 2.15 shows the distance histograms for a low dimensional (left) and a high dimensional (right) metric space. The gray areas correspond to those objects that cannot be discarded using the pivot exclusion condition. One can observe that most of the objects cannot be discarded in the high dimensional metric space.

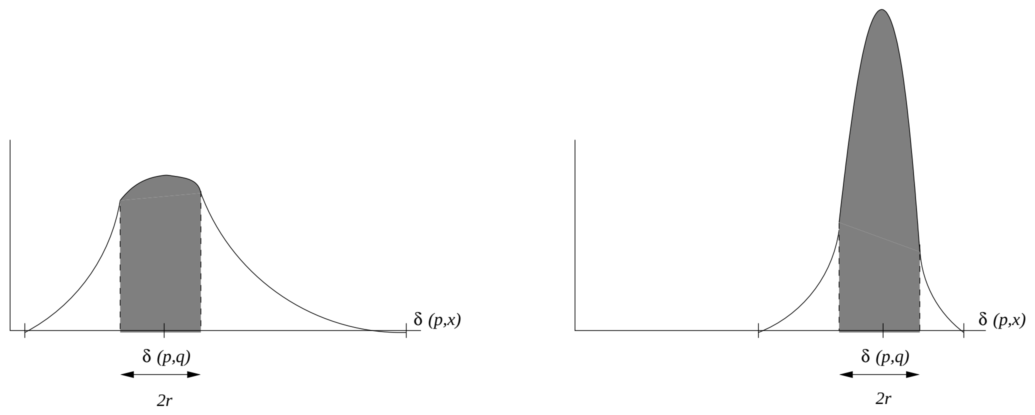

Figure 2.15: Distance histogram of a metric space with low (left) and high (right) intrinsic dimension

The same effects can be observed in vector spaces of high intrinsic dimensionality [Böhm et al., 2001]. Cost models for similarity queries in high dimensional spaces show an exponential dependence on the dimensionality of the vector space [Friedman et al., 1977]. Böhm [2000] presents a more complex cost model based on the Minkowski sum. In addition, the famous results presented in Beyer et al. [1999] (and recently in Shaft and Ramakrishnan [2005]) show theoretically that, for very high dimensionality, the NN problem is inherently linear for a wide range of data distributions.

2.6 Probabilistic and approximate similarity search algorithms

The main bottleneck of the research in similarity search is the curse of dimensionality, which makes the task of searching some metric/vector spaces intrinsically difficult in high dimensional spaces, whatever algorithm is used. A recent trend to remove this bottleneck resorts to probabilistic algorithms, where it has been shown that one can find most of the relevant objects at a fraction of the cost of the exact algorithm. These algorithms are welcome in most applications, because resorting to similarity searching already involves a fuzziness in the retrieval requirements: The process of modeling multimedia data as a metric or vector space involves generally some loss of information.

Thus, in most cases, finding some close objects is as good as finding all of them.

A survey on approximate similarity search algorithms is presented in Ciaccia and Patella [2001]. It proposes a classification scheme for existing approaches, considering as their relevant characteristics: Type of data (metric or vector spaces), error metrics (changing space or reducing comparisons), quality guarantees (none, probabilistic parametric/non-parametric, or deterministic), and user interaction (static or interactive). Approximation algorithms for vector spaces are surveyed in depth in White and Jain [1996a].

2.6.1 Probabilistic algorithms for metric spaces

Zezula et al. [1998] introduce approximate k-NN queries with the M-tree. Three different approximation techniques are proposed, which trade query precision for improved efficiency: Approximation by relative distance errors, approximation by distance distributions, and approximation by the slow-down of distance improvements. The experimental results suggest that the best method is the one based on distance distributions. Given the distance distribution $F_q(x)$ of a query object q (i.e., $F_q(x)$ is the fraction of objects from the database which distance to q is closer or equal than x), the stopping criterion $F_q(d(q, o_A^k)) \leq \rho$ can be defined, where o_A^k is the k^{th} approximated nearest neighbor of q (as found by the search algorithm) and ρ is a given treshold. This criterion is used to stop the search before the exact k-NN are found. No search improvements are obtained when $\rho \leq F_q(\delta(q, o_N^k))$, where o_N^k is the actual k^{th} nearest neighbor of q. If the distribution F_q is unknown, Zezula et al. [1998] propose to use a “representative distance function”, for example the average distribution function defined as $F_{avg}(x) = E[F_o(x)]$.

Clarkson [1999] proposes a data structure called $M(\mathbb{U}, Q)$ to answer nearest neighbor queries. It requires a training dataset Q of m objects, taken to be representative of typical query objects. This data structure may fail to return a correct answer, but the failure probability can be made arbitrarily small at the cost of increasing the query time and space requirements for the index. When the metric space obeys a certain sphere-packing bound [Clarkson, 1999], it is shown that $M(\mathbb{U}, Q)$ answers range queries in $O(K \ln(n) \log(\Upsilon(\mathbb{U} \cup Q)))$ time, with failure probability $O(\log^2(n)/K)$ and requires $O(Kn \log(\Upsilon(\mathbb{U} \cup Q)))$ space, where K is a parameter that allows one to control the failure probability and $\Upsilon(T)$ is the ratio of the distance between the farthest and closest pair of points of T.

Ciaccia and Patella [2000] present an approach to approximate nearest neighbor similarity search called *probabilistic approximately correct NN*

(PAC-NN). The algorithm retrieves an $(1+\varepsilon)$ nearest neighbor with probability greater or equal than $1-\theta$, where ε and θ are parameters that can be tuned at query time. The algorithm can be implemented in an arbitrary index: Ciaccia and Patella describe both sequential and index-based PAC-NN algorithms. Given a query object q, r_θ^q is defined as the maximum distance from q so that the probability of finding an object closer to q than r_θ^q is lower or equal than θ. An estimation of r_θ^q may be obtained from the distance distribution of the query points. Then, the database is scanned until an object u such that $\delta(q,o) \leq (1+\varepsilon)r_\theta^q$ is found, reporting u as the probably approximately correct nearest neighbor of q. On the other hand, an $(1+\varepsilon)$ approximation is guaranteed by pruning from the search every element whose lower bound distance to q (proved by the index structure) exceeds $r^*/(1+\varepsilon)$, where r^* is the current distance to the k^{th} nearest neighbor.

Goldstein and Ramakrishnan [2000] propose an index structure called *P-Sphere tree* for nearest neighbor queries. The tree has a two-level structure, a root level and a leaf level. The root contains a list of "sphere descriptions" and pointers to all leaf levels. Each leaf contains a *center point* and all data points that lie within the sphere described in the corresponding sphere descriptor from the root level. Three parameters must be set before constructing the tree: The fanout of the root, the center points in the sphere descriptors, and the leaf size. The search algorithm consists in determining the leaf whose center point is closest to the query object. Then, a linear scan is performed on that leaf, reporting the closest object to the query. By selecting the appropriate parameters at construction time, which also depend in the desired accuracy level, the index will yield a probably correct answer.

Chávez and Navarro [2003] present a probabilistic algorithm based on "stretching" the triangle inequality. The idea is general, but it is applied to pivot based algorithms. Their analysis shows that the net effect of the technique is to reduce the search radius by a factor β, and that this reduction is larger when the search problem becomes harder, i.e., when the intrinsic dimension of the space becomes high. Even with very little stretching, large improvements in the search time are obtained with low error probability. The factor β can be chosen at search time, so the index can be built beforehand and later one can choose the desired level of accurateness and speed of the algorithm. As the factor is used only to discard elements, no element closer to q than r/β can be missed during the search. In practice, all the elements that satisfy $|\delta(p_i,u)-\delta(p_i,q)| > r/\beta$ for some p_i are discarded. Figure 2.16 illustrates how the idea operates. The exact algorithm guarantees that no relevant object is missed, while the probabilistic one stretches both sides of the ring and may miss some objects.

Bustos and Navarro [2004] propose probabilistic algorithms for indices

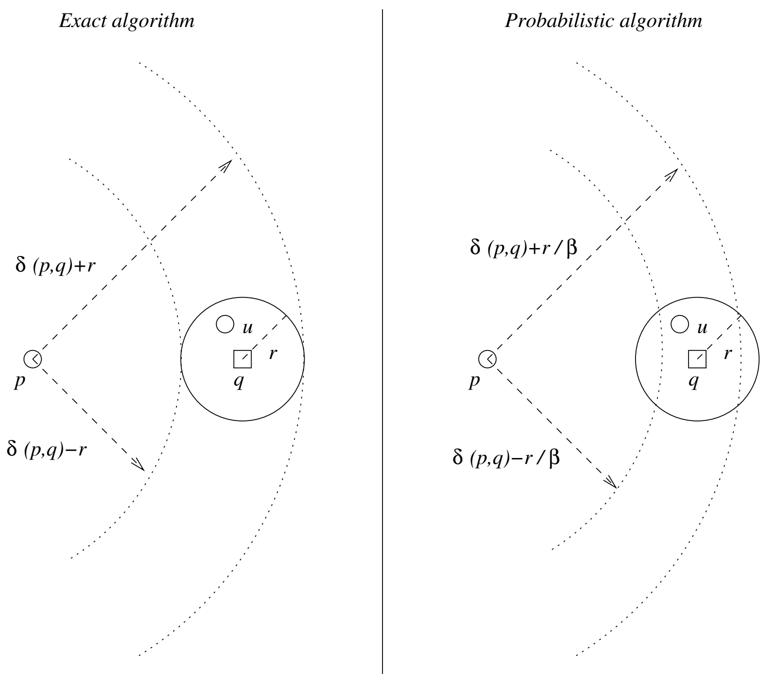

Figure 2.16: Exact and probabilistic algorithm based on pivots

based on compact partitions. The incremental nearest neighbor search [Hjaltason and Samet, 2000] is modified, assigning a fixed amount of work to perform the similarity query. Thus, the resultant algorithm performs time-bounded range search queries in metric spaces. Also, a probabilistic technique based on ranking zones is presented, which is a generalization of the former technique. The idea is to sort the compact partitions induced by the index, to favor the most promising ones. Then, the list is traversed until the assigned amount of work is used up. Several zone ranking criteria were tested, obtaining the best empirical results with an adaptive ranking criterion that weighs the lower bound of the distance from q to a given compact partition with its covering radius.

2.6.2 Probabilistic algorithms for vector spaces

Arya and Mount [1995] propose a general framework to search for an arbitrary region Q in $(\mathbb{R}^d, L_2)$. The idea is to define areas Q^- and Q^+ such that $Q^- \subset Q \subset Q^+$. Points inside Q^- are guaranteed to be reported and points outside Q^+ are guaranteed not to be reported. In between, the algorithm gives no guarantee. The maximum distance between the real and the bounding areas is ε. The vector space is partitioned and indexed using trees, which are used to guide the search by including or excluding whole areas. Every decision about including (excluding) a whole area can be done using Q^+ (Q^-) to increase the probability of filtering out the search in either way. Those areas that cannot be fully included or excluded are analyzed in more detail by going down to the appropriate subtree. The complexity is shown to be $O(2^d \log(n) + (3\sqrt{d}/\varepsilon)^d)$, and a very close lower bound is proven for the problem.

Arya et al. [1994] propose a data structure called *BBD-tree* for searching in a vector space $\mathbb{R}^d$ under any metric L_p. This structure is inspired in the *kd-tree* and it can be used to find the "$(1+\varepsilon)$ nearest neighbor", that is, to find an object u^* such that $\forall u \in \mathbb{U}$, $\delta(u^*, q) \leq (1+\varepsilon)\delta(u, q)$ (see Figure 2.17). The essential idea of the algorithm is to locate the query q in a cell (each leaf in the tree is associated with a cell in the space decomposition). Every point inside the cell is processed to obtain its nearest neighbor p. The search stops when no promising cells are found, i.e., when the radius of any ball centered at q and intersecting a nonempty cell exceeds the radius $\delta(q, p)/(1+\varepsilon)$. The search time for this algorithm is $O(\lceil 1 + 6d/\varepsilon \rceil^d \log(n))$.

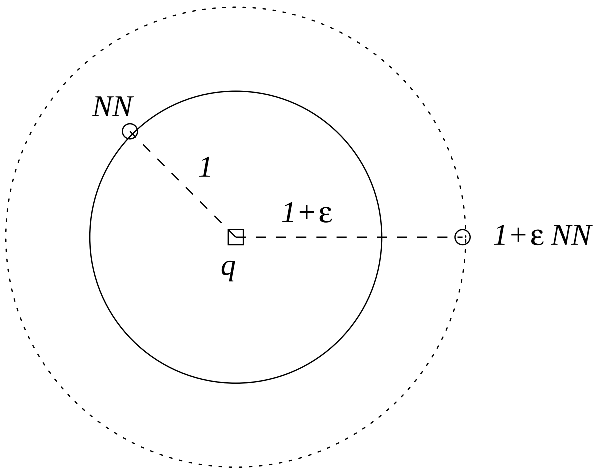

Figure 2.17: Nearest neighbor and $(1+\varepsilon)$ nearest neighbor

Yianilos [2000] presents a technique called "aggressive pruning" for "limited radius nearest neighbors". This query searches for nearest neighbors that are inside a given radius. The idea can be seen as a special case of the framework proposed in Arya and Mount [1995], where the search area is a ball and the data structure is a *kd-tree*. Relevant elements may be lost but irrelevant ones cannot be reported, i.e., $Q^+ = Q$. The ball Q, of radius r and centered at $c = (c_1, \ldots, c_d)$ is filtered out by intersecting it with the area between hyperplanes $c_i - r + \varepsilon$ and $c_i + r - \varepsilon$. The author gives a probabilistic analysis assuming normally distributed distances, which almost holds if the points are uniformly distributed in the space. A parameter $\lambda \in [0, 1]$, independent of the dimensionality of the space, controls the probability that the real NN will be found. The search time is shown to be $O(n^{\lambda})$, where λ decreases as the permitted failure probability ε increases.

Chapter 3

Effectiveness of feature based similarity search

Similarity search systems are designed to retrieve similar objects for a given query from a database. As the concept of “similarity” is inherently vague and depends on the actual application, a retrieval performance evaluation is usually performed to assess how precise the search system is. The *effectiveness* of a similarity search system measures its ability to retrieve relevant objects from the database while at the same time holding back non-relevant ones. Improving the effectiveness of a similarity search system is at least as important as improving its efficiency, because effectiveness is directly related to the quality of the answers that the search system returns.

This chapter presents several techniques that aim to improve the effectiveness of a similarity search engine. For this purpose, we present a case study on 3D model retrieval, although the techniques are general and can be used with any type of multimedia data. We compare a variety of 3D descriptors and develop methods for improving the effectiveness of 3D similarity search. We propose two techniques, called *purity* and *entropy impurity* methods, which assess a priori the quality of a descriptor given a query object. These methods are used for selecting and combining the best descriptors to perform the similarity query, at search time. Our experimental evaluation shows that dynamic combinations of feature vectors may lead to a significant improvement in effectiveness compared with the best single descriptor.

The ideas presented on this chapter have been published in Bustos et al. [2004c,b,a, 2005a, 2006c].

3.1 Evaluation of single feature vectors

3.1.1 Case study: 3D object databases

The problem of searching similar 3D objects arises in a number of fields. Example problem domains include Computer Aided Design/Computer Aided Manufacturing (CAD/CAM), virtual reality (VR), medicine, molecular biology, and entertainment. The improvement in 3D scanner technology and the availability of 3D models widely distributed over the Internet are rapidly contributing to create large databases of this type of multimedia data. Also, the rapid advances in graphics hardware are making the fast processing of this complex data possible and available to a wide range of potential users at a relatively low cost.

As 3D objects are used in diverse application domains, different forms for object representation, manipulation, and presentation have been developed. In the CAD domain, objects are often built by merging patches of parameterized surfaces, which are edited by technical personnel. Also, constructive solid geometry techniques are often employed, where complex objects are modeled by composing primitives. 3D acquisition devices usually produce voxelized object approximations (e.g., computer tomography scanners) or clouds of 3D points (e.g., in the sensing phase of structured light scanners). Probably the most widely used representation to approximate a 3D object is by a mesh of polygons, usually triangles (see Figure 3.1 for an example). For a survey on important representation forms, see Campbell and Flynn [2001]. For 3D retrieval, basically all of these formats may serve as input to a similarity query. Where available, information other than pure geometry data can be exploited, for example structural data that may be included in a VRML (Virtual Reality Modeling Language) representation. Many similarity search methods that are presented in the literature to date rely on triangulations but could easily be extended to other representation forms. Of course, it is always possible to convert or approximate from one representation to another.

3D objects may be very complex, both in terms of the data structures and methods used to represent and to visually render such objects, as well as in terms of the topological and geometric structures of the objects themselves. The primary goal in 3D, as well as in other similarity search domains, is to design algorithms with the ability to effectively and efficiently execute similarity queries. One option is direct geometric matching, where it is measured how easily a given object can be transformed into another one. The cost associated with this transform serves as the metric for similarity [Novotni and Klein, 2001]. However, directly comparing all objects of a database with

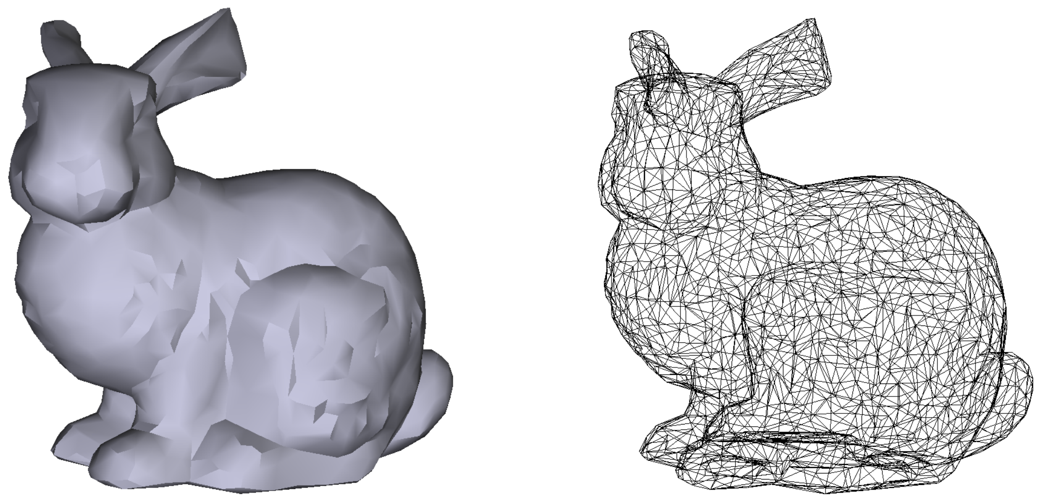

Figure 3.1: A 3D model of a bunny and its mesh of polygons

a query object is time consuming and may be difficult, because 3D objects can be represented in many different formats and may exhibit widely varying complexity. Given that it is also not clear how to use geometry directly for efficient similarity search, in typical methods the 3D data is transformed in some way to obtain numerical descriptors (FVs) for indexing and retrieval. These FVs characterize certain features of 3D objects and can be efficiently compared to each other in order to identify similar shapes and to discard dissimilar ones.

Invariance requirements

Considering the feature-based approach, one can define several requirements that an effective FV for 3D objects should meet. Good FVs should abstract from the potentially very distinct design decisions that different model authors make when modeling the same or similar 3D objects. Specifically, the FVs should be *invariant* to changes in the orientation (translation, rotation and reflection) and scale of 3D models in their reference coordinate frame. That is, the similarity search engine should be able to retrieve geometrically similar 3D objects with different orientations. Figure 3.2 (left) illustrates different orientations of a Porsche car 3D object: The extracted FV should be (almost) the same in all cases. Ideally, an arbitrary combination of translation, rotation and scale applied to one object should not affect its similarity score with respect to another object.

Furthermore, a FV should also be *robust* with respect to small changes of the level-of-detail, geometry and topology of the models. Figure 3.2 (right) shows the Porsche car 3D object at four different levels of resolution. If such transformations are made to the objects, the resulting similarity measures should not change abruptly, but still reflect the overall similarity relations within the database.

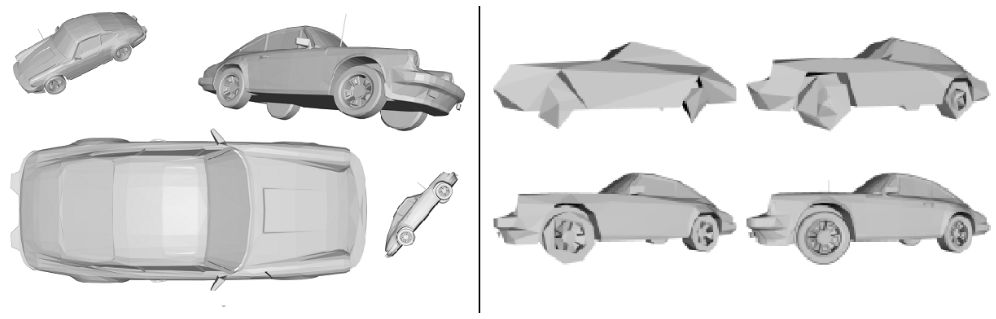

Figure 3.2: A 3D object in different scale and orientation (left), and also represented with increasing level of detail (right)

Invariance and robustness properties can be achieved implicitly by those FVs that consider relative object properties, for example, the distribution of surface curvature of the objects. For other FVs, these properties can be achieved by a *preprocessing normalization step*, which transforms the objects so that they are represented in a canonical reference frame. In such a reference frame, directions and distances are comparable between different models, and this information can be exploited for similarity calculation. The predominant method for finding this reference coordinate frame is pose estimation by *principal component analysis* (PCA), also known as Karhunen-Loève transformation [Paquet et al., 2000a; Vranić et al., 2001]. The basic idea is to align a model by considering its center of mass and principal axes. The object is translated to align its center of mass with the coordinate origin (*translation invariance*), and is then rotated about the origin such that the x, y and z axes coincide with the three principal components of the object (*rotation invariance*). Additionally, *flipping invariance* may be obtained by flipping the object based on a moment test, and *scaling invariance* can be achieved by scaling the model by a canonical factor. Figure 3.3 illustrates PCA-based pose and scaling normalization of 3D objects.

A final FV property that is also desirable to have is the *multi-resolution property*. Here, the FV contains progressive model detail information, which can be used for similarity search on different levels of resolution. This property eliminates the need to extract and store multiple FVs with different levels of resolution if multi-resolution search is required, for example for implementing a filter-and-refinement step. A main class of FVs that implicitly

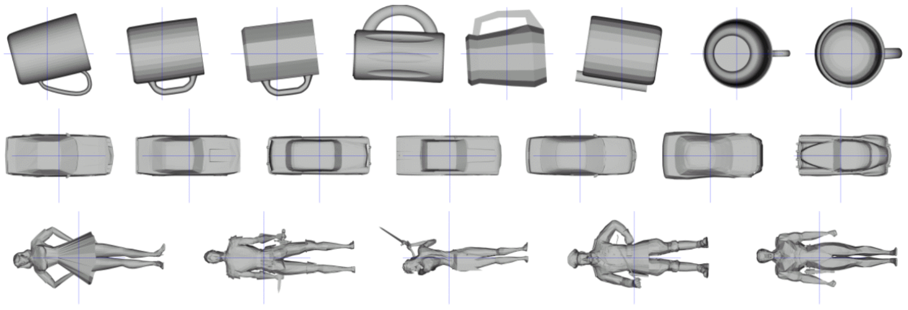

Figure 3.3: Pose estimation using the PCA for three classes of 3D objects

provide the multi-resolution property is one that performs a discrete Fourier or Wavelet transform of sampled object measures.

Feature vector extraction for 3D objects

The extraction of FVs generally can be regarded as a multistage process (see Figure 3.4). In this process, a given 3D object, usually represented by a polygonal mesh, is first preprocessed to achieve the required invariance and robustness properties. The object is then transformed so that its character is either of surface type, or volumetric, or captured by one or several 2D images. Then, a numerical analysis of the shape can take place and, from the result, the FVs are extracted. A brief overview of these basic steps is given below:

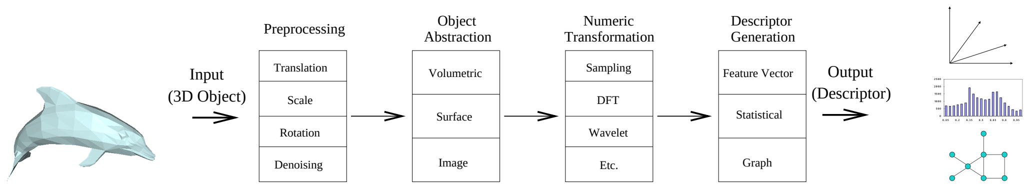

Figure 3.4: Descriptor extraction process model

1. *Preprocessing.* If needed, the preprocessing normalization step (e.g., using PCA as explained in the previous subsection) is applied to the 3D model in order to achieve the rotation, translation, and scale invariance.

2. *Type of object abstraction.* A polygonal mesh can be seen in different ways. We may regard it as an ideal mathematical surface, infinitely thin, with precisely defined properties of differentiability. Alternatively, we may look at it as a thickened surface that occupies some portion

of volume in 3D space, or for watertight models (i.e., models without holes or gaps and where the "inside" and "outside" is clearly defined) as a boundary of a solid volumetric object. The transformation of a mesh into one of these forms is typically called *voxelization.* Statistics of the curvature of the object surface is an example of a descriptor based directly on a surface, while measures for the 3D distribution of object mass, for example, using moment-based descriptors, belong to the volumetric type of object abstraction. A third way to capture the character of a mesh would be to project it onto one or several image planes producing renderings, corresponding depth maps, silhouettes, and so on, from which descriptors can be derived.

3. *Numerical transformation.* The main features of meshes in either of the types of object abstractions outlined before can be captured numerically using one of various methods. Voxel grids and image arrays can be Fourier or Wavelet transformed, and surfaces can be adaptively sampled. This yields a numerical representation of the underlying object. It is not required that the numerical representation allows the complete reconstruction of the 3D object. However, these numerical representations are set up to readily extract the mesh shape descriptors in the final phase of the process.

4. *Descriptor generation.* The descriptors for 3D shape may be grouped in three main categories based on their form.

   (a) *Feature vectors*, or *FVs*, consist of elements in an usually high-dimensional vector space equipped with a suitable metric (e.g., any Minkowski distance). Such feature vectors may describe conceptually different types of shape information, such as spatial extent, visual expression, surface curvature, and so forth. The dimensionality of the vectors is defined by the feature extractor process, and it usually depends on a user-given parameter (in general not all dimensionalities are possible, but one can obtain vectors of different dimensionalities by using different parameter values).

   (b) In *statistical approaches,* 3D objects are inspected for specific features, which are summarized usually in the form of a histogram. For example, in simple cases this amounts to the total surface area in specified volumetric regions. In more complex cases, it may collect statistics about the distances of point pairs randomly selected from the 3D object.

(c) The third descriptor category is more suitable for a *structural 3D object shape description* that can be represented in the form of a graph [Sundar et al., 2003; Hilaga et al., 2001]. A graph can more easily represent the structure of an object that is made up from or can be decomposed into several meaningful parts, such as the body and the limbs of objects modeling animals. However, finding a good dissimilarity measure for graphs is not as straightforward as for feature vectors, and, moreover, small changes in the 3D object may lead to large changes in the corresponding structural graph, which is not ideal for solving the retrieval problem.

The methods in the feature vector class are efficient, robust, easy to implement, and provide some of the best approaches for 3D model retrieval [Tangelder and Veltkamp, 2004; Kazhdan et al., 2003; Vranić, 2004]. Therefore, these are the most popular ones that are explored in the literature. We do not imply, however, that the other methods may be inferior and should therefore be discarded from future research. Most of these methods have their particular strengths and may well be the ideal candidate for a specific application.

Details of the implementations for the different FV methods for 3D model retrieval are outside of the scope of this thesis. For details, see Bustos et al. [2005a], where more than 30 feature extraction methods for 3D model retrieval are surveyed.

3.1.2 Evaluation approach

The database used for our experiments contains 1,838 3D objects that we collected from the Internet[1]. From this set, 472 objects were manually classified by shape similarity into 55 different model classes. Figure 3.5 shows some of the 3D objects from two model classes: Formula 1 (F-1) cars and sea animals. Each classified object of each model class was used as a query object. The objects belonging to the same model class, excluding the query, were taken as the relevant objects.

Table 3.1 gives a complete description of the classified objects of the database. The first column indicates the class identification number. The second column describes the 3D class models. The last column lists the number of objects per model class.

For our 3D model retrieval system, we implemented 16 different FVs, including statistical FVs (3D moments [Paquet et al., 2000b]), geometry-based FVs (ray-based [Vranić and Saupe, 2000], cord-based [Paquet et al.,

[1]Konstanz 3D model search engine. `http://merkur01.inf.uni-konstanz.de/CCCC/`

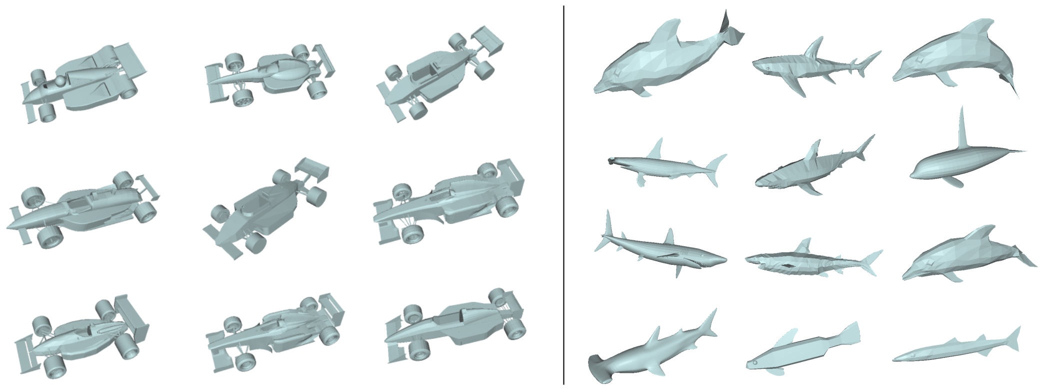
Figure 3.5: Formula 1 cars (left) and sea animals (right) model classes

Class id	Description	# models	Class id	Description	# models
1	ants	6	29	submarines	5
2	rabbits	4	30	warships	5
3	cows	7	31	beds	7
4	dogs	4	32	chairs	24
5	fish-like	13	33	office chairs	6
6	bees	5	34	sofas	4
7	CPUs	4	35	benches	3
8	keyboards	8	36	couches	11
9	cans	4	37	axes	4
10	bottles	14	38	glasses	7
11	bowls	4	39	knives	3
12	pots	4	40	screws	3
13	cups	8	41	spoons	3
14	wine glasses	9	42	tables	6
15	teapots	4	43	skulls	3
16	biplanes	5	44	human heads	8
17	helicopters	9	45	human masks	4
18	missiles	16	46	books	4
19	jet planes	18	47	watches	2
20	fighter jet planes	26	48	sand clocks	4
21	propeller planes	10	49	swords	25
22	other planes	4	50	barrels	3
23	zeppelins	6	51	birches	4
24	motorcycles	5	52	flower pots	9
25	sport cars	6	53	trees	11
26	cars	23	54	weeds	9
27	Formula-1 cars	9	55	human bodies	56
28	galleons	4			

Table 3.1: Description of the classified set of our 3D object database

2000b], shape spectrum [Zaharia and Prêteux, 2001], ray-based with spherical harmonics [Vranić and Saupe, 2002], shape distribution [Osada et al., 2002], ray moments [Vranić and Saupe, 2001], harmonics 3D [Funkhouser et al., 2003], voxel-based [Heczko et al., 2002], 3DDFT [Vranić and Saupe, 2001], shading [Vranić and Saupe, 2002], complex valued shading [Vranić and Saupe, 2002], segment volume occupation [Heczko et al., 2002]), image-based FVs (depth buffer [Heczko et al., 2002], silhouette [Heczko et al., 2002]), and other approaches (rotation invariant point cloud [Kato et al., 2000]).

Table 3.2 presents an overview of the implemented 3D descriptors in our system, according to the processing pipeline from Figure 3.4. The column labeled "Prep." indicates the preprocessing steps that must be applied to the 3D object (R: Rotation, T: Translation, S:Scale). "Obj. abstr." indicates the classification with regard to the underlying object abstraction (volumetric-, surface-, and image-based). "Numerical transf." indicates whether a numerical transformation is applied or not, and which kind (sampling, discrete Fourier transform, spherical harmonics, and curve fitting). Finally, "Type" indicates whether the final descriptor is a FV or a histogram.

Descriptor name	Prep.	Obj. abstr.	Numerical transf.	Type
Rotation invariant point cloud	RTS	Volumetric	Sampling	Hist.
Voxel-based	RTS	Volumetric	None	Hist.
3DDFT	RTS	Volumetric	3D DFT	FV
Segment volume occupation	RTS	Volumetric	None	FV
Harmonics 3D	TS	Volumetric	Spherical harmonics	FV
3D moments	RTS	Surface	Sampling	FV
Ray moments	RTS	Surface	Sampling	FV
Cords-based	RT	Surface	Sampling	Hist.
Shape distribution	None	Surface	Sampling	Hist.
Shape spectrum	None	Surface	Curve fitting	Hist.
Silhouette	RTS	Image	Sampling + DFT	FV
Depth buffer	RTS	Image	2D DFT	FV
Ray-based	RTS	Image	Sampling	FV
Rays with sph. harm.	RTS	Image	Sampl. + sph. harm.	FV
Shading	RTS	Image	Sampl. + sph. harm.	FV
Complex valued shading	RTS	Image	Sampl. + sph. harm.	FV

Table 3.2: Overview of the implemented 3D feature vectors

3.1.3 Effectiveness comparison between descriptors

We use *precision versus recall figures* [Baeza-Yates and Ribeiro-Neto, 1999] for comparing the effectiveness of the search algorithms. *Precision* is the fraction of the retrieved objects which is relevant to a given query, and *recall* is the fraction of the relevant objects which has been retrieved from the database. That is, if R is the set of relevant objects to the query, A is the set of objects retrieved, and R_A is the set of relevant objects in the result set, then

$$precision = \frac{|R_A|}{|A|} \text{ and } recall = \frac{|R_A|}{|R|}.$$

All our precision versus recall figures are based on the eleven standard recall levels (0%, 10%, ..., 100%), and we average the precision figures over all test queries at each recall level.

In addition to the precision at multiple recall points, we also calculate the *R-precision* [Baeza-Yates and Ribeiro-Neto, 1999] for each query, which is defined by the precision when retrieving only the first $|R|$ objects. The R-precision gives a single number to rate the performance of a retrieval algorithm.

We tested all these FVs using different levels of resolution (from a few dimensions up to 512-D), and used the Manhattan (L_1) distance as the similarity function between vectors (we also tested L_2 and L_{max}, but we consistently obtained the best effectiveness scores using L_1). Table 3.3 shows the best R-precision values obtained with all the FVs in descending order. The first column lists the different descriptors. The second column indicates the best dimensionality (in terms of effectiveness) of the FV. The last column lists the average R-precision values obtained for each FV with their best dimensionality.

Descriptor	Best dim.	Avg. R-precision
Depth buffer	366	0.3220
Voxel-based	343	0.3026
Complex valued shading	196	0.2974
Rays with spherical harmonics	105	0.2815
Silhouette	375	0.2736
3DDFT	365	0.2622
Shading	136	0.2386
Ray-based	42	0.2331
Rotation invariant point cloud	406	0.2265
Harmonics 3D	112	0.2219
Shape distribution	188	0.1930
Ray moments	363	0.1922
Cords-based	120	0.1728
3D moments	31	0.1648
Segment volume occupation	486	0.1443
Shape spectrum	432	0.1119

Table 3.3: Average R-precision of the implemented 3D descriptors

The best overall FV among our set of implemented methods was the depth buffer, with an average R-precision of 0.32. The difference in effectiveness between the best and the worst performing FV (depth buffer and shape spectrum, respectively) was significant (up to a factor of 3x). However, the difference in effectiveness between similar performing FVs was small, especially when comparing the most effective descriptors. This implies that, in practice, these best FVs should be suited about equally well for retrieval of general polygonal objects. We observed that descriptors that rely on consistent polygon orientation like shape spectrum or volume exhibited low retrieval rates, as consistent orientation is not guaranteed for many of the models retrieved from the Internet. Also, the geometrical moment-based descriptors seem to offer only limited discrimination capabilities. Figures 3.6 and 3.7 show the precision versus recall figures for all the implemented descriptors (first eight and last eight descriptors according to Table 3.3, respectively).

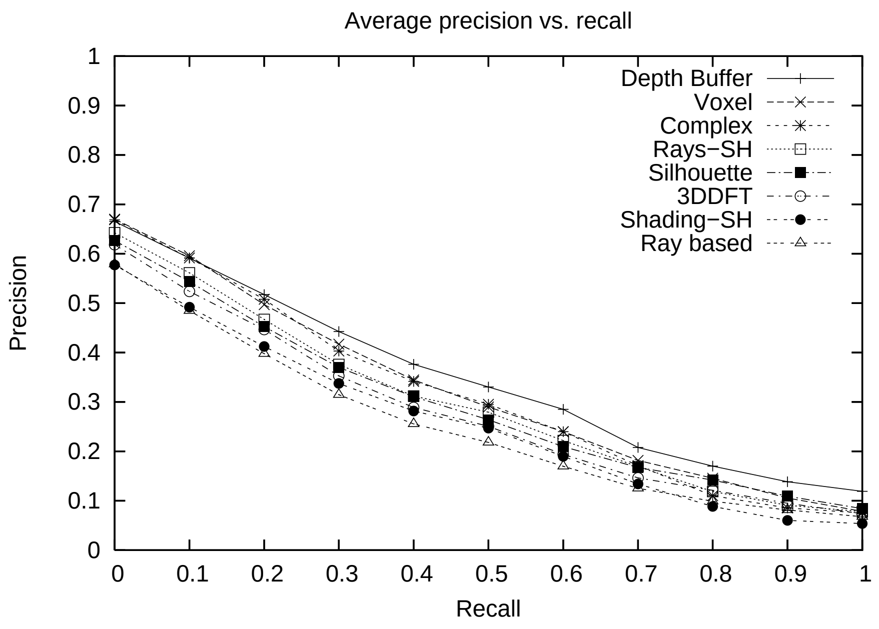

Figure 3.6: Average precision versus recall with best dimensionality, first eight descriptors according to Table 3.3

Figures 3.8 and 3.9 (first eight and last eight descriptors, respectively) show the effect of the descriptor dimensionality on the overall effectiveness. The figures show that the effectiveness of the FVs first increases with dimensionality, but the improvement rate diminishes quickly for roughly more than 64 dimensions for most FVs (except for 3DDFT). It is interesting to note that the saturation effect is reached for most descriptors at roughly the

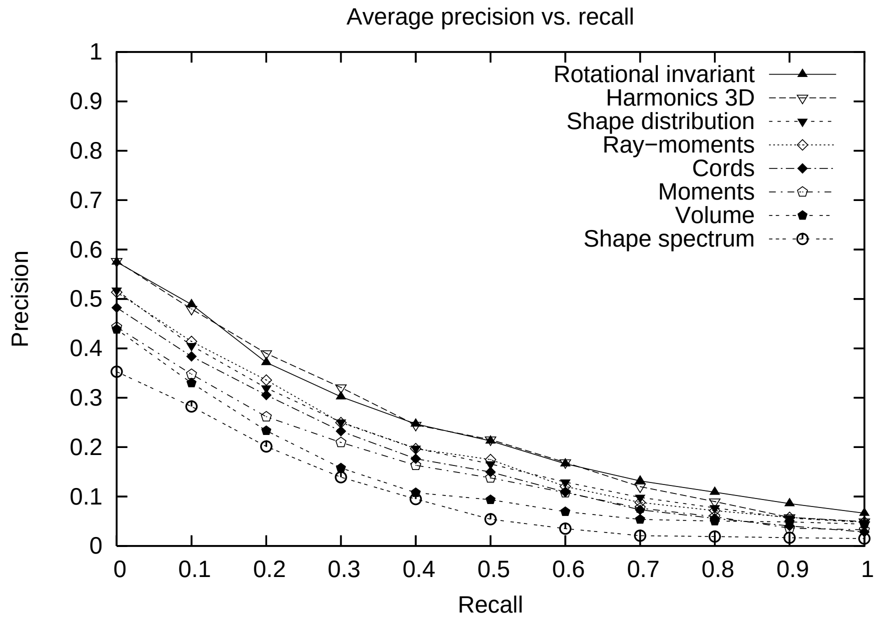

Figure 3.7: Average precision versus recall with best dimensionality, last eight descriptors according to Table 3.3

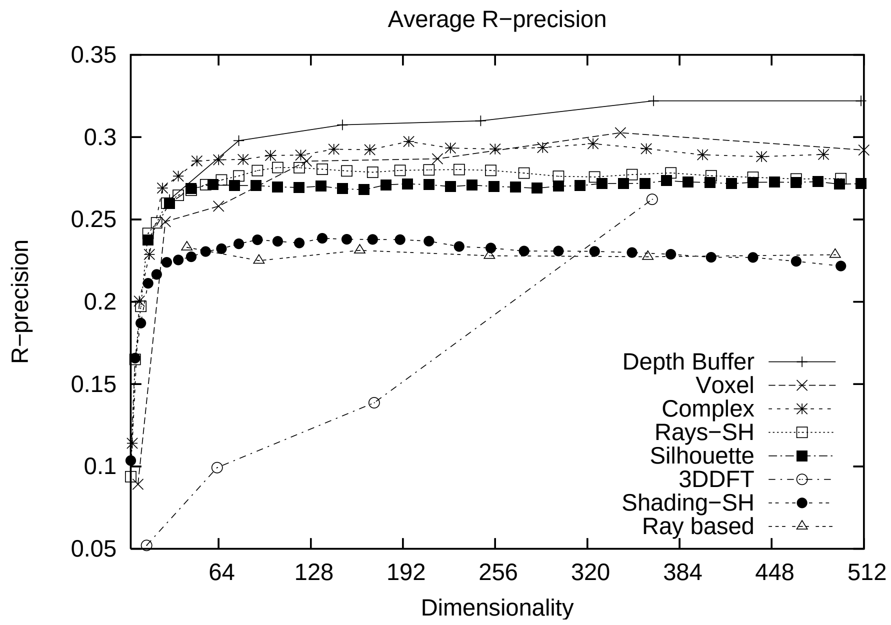

Figure 3.8: Dimensionality versus R-precision, first eight descriptors according to Table 3.3

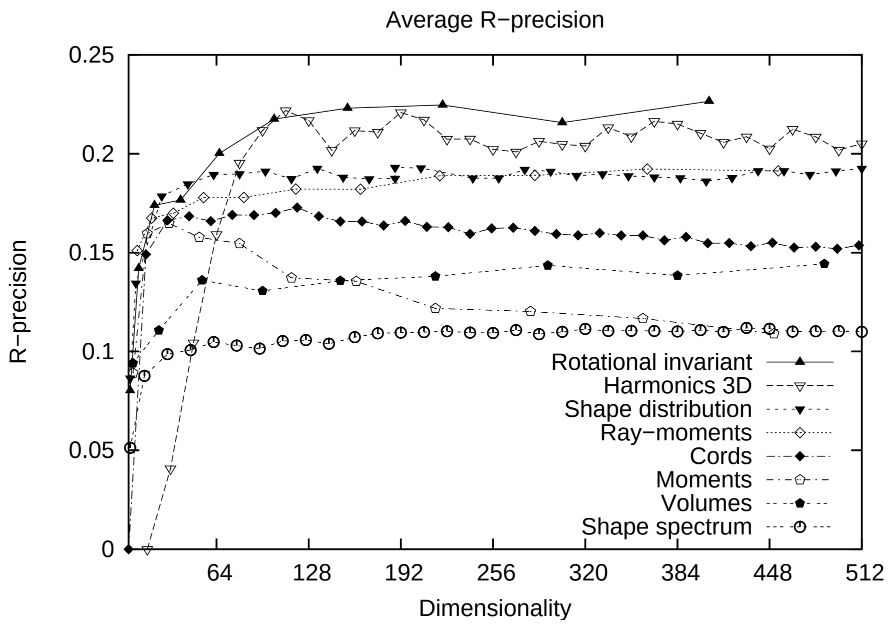

Figure 3.9: Dimensionality versus R-precision, last eight descriptors according to Table 3.3

same dimensionality level. This is an unexpected result, considering that different FVs describe different characteristics of 3D objects.

We also performed some tests using the Princeton shape benchmark [Shilane et al., 2004], to contrast our experimental results with those obtained using a different 3D ground truth. In summary, we obtained the same results as with our database with only minor differences (see Bustos et al. [2006c] for details).

3.1.4 Analysis of the experimental results

From the results obtained in our experiments on the set of implemented descriptors, we conclude that the best descriptors on average are those based on projections (2-D, ray-based) of the original 3D object, for example, depth buffer, silhouette, and so forth. Exceptions to this rule are the voxel and the 3DDFT FVs, which are volumetric descriptors and also obtained a good experimental effectiveness. Surface-based descriptors obtained, in general, low effectiveness values. All the implemented FVs showed good robustness with respect to the level of detail of the 3D objects. The good retrieval quality of image-based descriptors from our experiments are in accordance with [Chen et al., 2003], where an image-based descriptor embedded in an advanced multistage matching framework was shown to provide excellent retrieval results, and outperformed several other descriptors.

However, we also observed significant variance with respect to the effectiveness of retrieval when comparing the results for classes of objects. For different classes of objects, a different FV was usually the most effective one. For example, Figure 3.10 shows the average precision versus recall figures for the Formula 1 cars model class. In this case, the best effectiveness is obtained with the depth buffer and the harmonics 3D FVs. Note that the best FV for this model class is also the best FV on average. The R-precision value is also given for each FV.

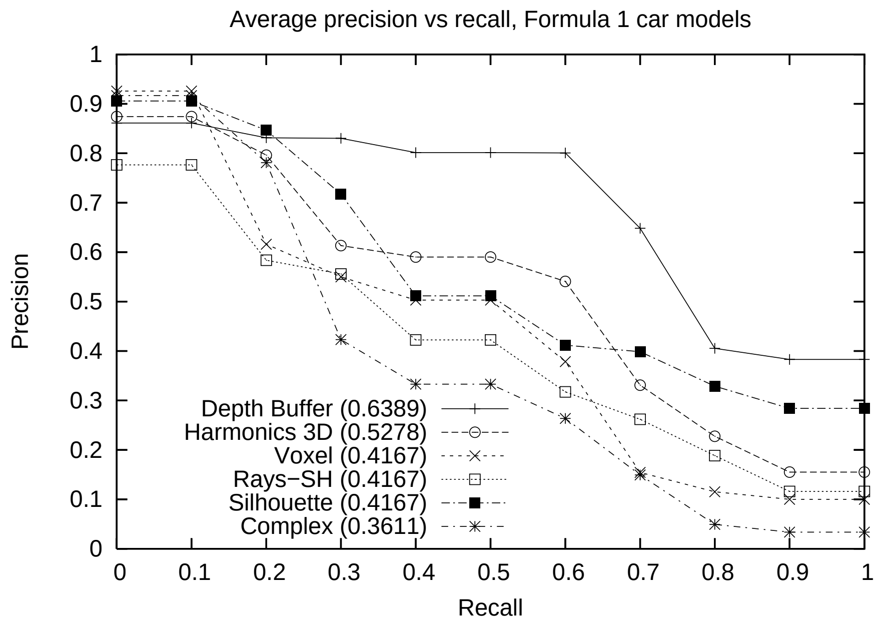

Figure 3.10: Average precision versus recall figures and R-precision values for the F-1 cars model class

Figure 3.11 shows the average precision versus recall figure for the sea animals model class. For this class, the best FVs are the silhouette and the ray-based spherical harmonics FVs. This result shows that for some model classes the best average FV (depth buffer) does not perform well. Moreover, the best three FVs for this class are different from the best three FVs of the F-1 cars model class. It follows that an appropriate selection of the FV used for the similarity search, *which depends on the query object*, may improve the overall retrieval effectiveness as compared with the standard policy of always choosing a certain default FV.

Unfortunately, we could not find a strong correlation between geometric

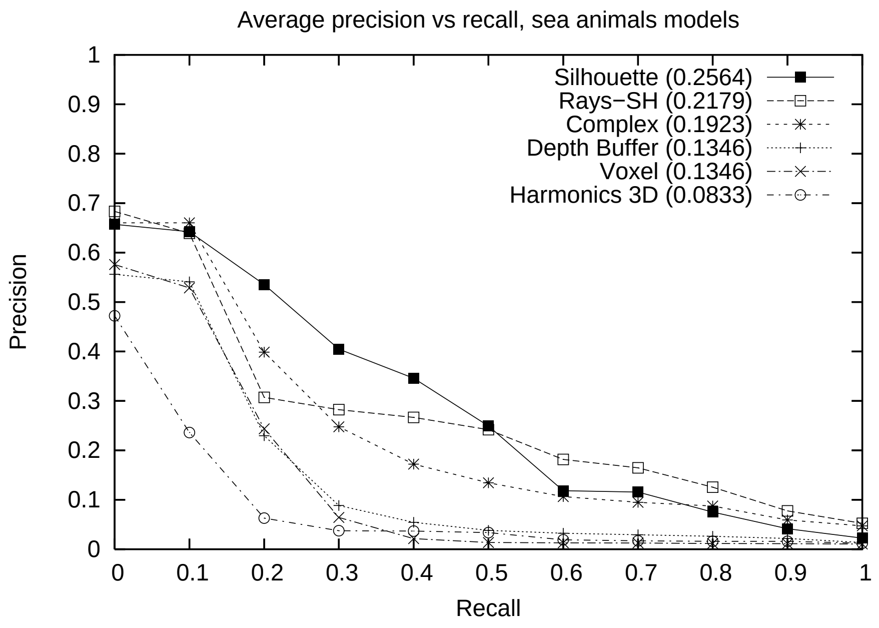

Figure 3.11: Average precision versus recall figures and R-precision values for the sea animals model class

properties of the 3D object class and the best suited FV for that model class. A notable exception for this is the shape spectrum FV (the worst descriptor on average), which obtained the best retrieval effectiveness for the human model class. Shape spectrum was able to recognize different models of human bodies in different poses, something that was not possible for the rest of the implemented FVs. This can be attributed to the fact that the shape spectrum FV considers the distribution of local curvature on the 3D object, which does not vary considerably in similar 3D models with different poses (e.g., human bodies in different positions). Another observation that we made is that model classes which are difficult to correctly orient using PCA (cf. Figure 3.3, first row) are best retrieved by FVs that are inherently rotational invariant, for example, the harmonics 3D FV.

Besides these specific exceptions, it is difficult to assess a priori which FV will have the best retrieval effectiveness for an unknown query object. On average, depth buffer or voxel will do pretty well, but one would like to always select the best FVs given a query object to perform the similarity search. In the rest of this chapter, we propose methods for assessing the suitability of a FV for a given query object, showing that *combinations of selected FVs* may enhance significantly the effectiveness of the similarity search system.

3.2 Combinations of feature vectors

The retrieval performance analysis of the previous section suggests that there are a number of FVs that achieve good average retrieval performance on the majority of query classes, but that there is no clear winner amongst them. Instead, the individual FVs have different strengths and weaknesses, and they represent complementary information regarding the description of 3D objects. Because FVs capture different aspects and characteristics of the models, we propose to use *combinations of FVs* for further improving the retrieval effectiveness of the similarity search, thus avoiding the disadvantages of using a single feature, capturing only a single characteristic of an object.

Figure 3.12 shows an example of three similarity queries (with the same query object) using two single feature vectors (depth buffer and silhouette, respectively) and a combination of both feature vectors (see Section 3.2.1). Note that all objects retrieved by the combination of feature vectors are similar to the query object.

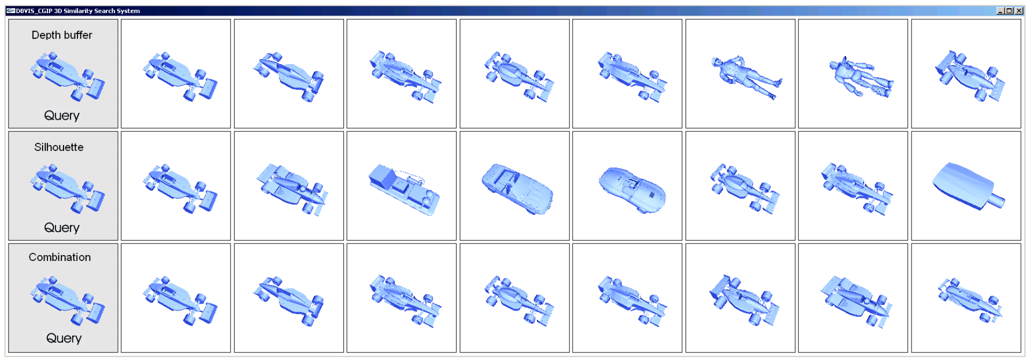

Figure 3.12: Example of similarity search using a combination of feature vectors

A question that arises is how can different FVs be combined in a similarity search system. A simple concatenation of all available FVs is not advisable because the effectiveness of the similarity search would degrade with the inclusion of FVs irrelevant to the query. Therefore, it is an interesting problem to find whether there are combinations of FVs that are better suited for performing similarity search on certain object classes, or even if there are combinations that dominate others for all types of queries.

We propose two methods for combining FVs: An unweighted combination of FVs and a dynamically weighted combination of FVs.

3.2.1 Unweighted combinations of feature vectors

We ran retrieval experiments on all possible combinations of the best 5 FVs according to Table 3.3 and using their best dimensionality. This gives a total of $\sum_{k=2}^{5}\binom{5}{k} = 26$ different combinations of FVs. To construct the combinations, we use the sum of the unweighted normalized distances using each FV.

Definition 1. *The unweighted normalized combined distance d_c is defined as:*

$$\delta_c(q,o) = \sum_{i=1}^{F} b_{c_i} \frac{\delta_i(q,o)}{norm_i}$$

where F is the total number of FVs, b_{c_i} is a binary variable that indicates whether FV f_i is included in combination c, δ_i is the distance function used with FV f_i, and $norm_i$ is a normalization factor for f_i.

The unweighted combination approach treats all FVs of the combination as equally important when determining the similar objects for the query object. The normalization factors are required to make the sum of distances meaningful (e.g., in case the FVs produce points of different dimensionalities). The advantage of using the *sum of distances* as combined distance is that if all δ_i distances are metrics, the sum of distances also holds the properties of a metric (for a proof, see Section 4.3.1).

#	R-precision	Combined feature vectors
1	0.3220	Depth buffer
2	0.3803	Voxel, complex
3	0.4108	Depth buffer, voxel, complex
4	0.4200	Depth buffer, voxel, complex, silhouette
5	0.4220	Depth buffer, voxel, complex, silhouette, rays-SH

Table 3.4: Average R-precision for the best unweighted combinations of feature vectors

Table 3.4 shows the average effectiveness of the best combinations of FVs in terms of R-precision and combination cardinality. The results confirm our assumption that there are FV combinations that significantly improve the retrieval performance over the best single FV (depth buffer) in the average case. The maximum R-precision value reached on average over all query classes by a combination amounts to 42.20%, which is equal to an improvement of more than 31% compared to the performance of the depth buffer.

This best combination is composed of all five FVs. The largest improvement occurs when changing from the single to the 2-combination case (voxel and complex FVs). The improvement increases further with combination cardinality, but the increment becomes smaller as we add more FVs to the combination. For the last combination, the improvement in effectiveness is negligible.

Figure 3.13 shows the precision versus recall curves for the best unweighted combinations.

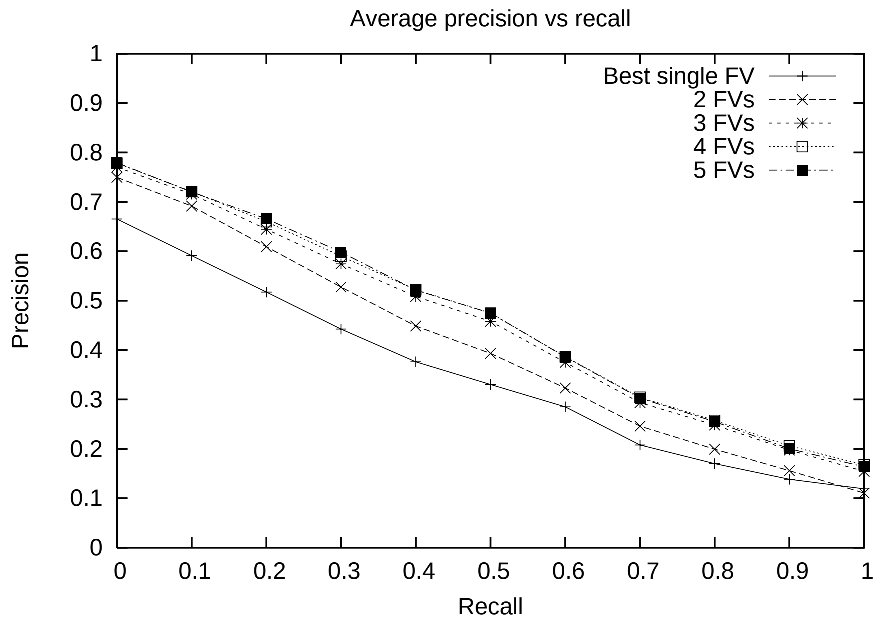

Figure 3.13: Average precision versus recall for the best feature vector and the best unweighted combinations for an increasing numbers of feature vectors

We also performed a much larger series of experiments considering combinations of up to nine FVs (this was the limit in practice that returned results in reasonable time). In these experiments, we found that the retrieval effectiveness even starts to decrease when adding more FVs after a certain saturation point has been reached. We reached this saturation point with combinations of 5 FVs.

3.2.2 Weighted combinations of feature vectors

A further improvement over the unweighted combination of FVs can be achieved by assigning *weights* to each FV in the combined distance, because it is expected that not all FVs are equally relevant to all queries, and using a non-suitable FV can even lower the effectiveness of the search. We tested all possible weightings for the combination of the six FVs using three different weight values (0, 1, 2), resulting in $3^5 - 1 = 242$ different combinations. We call this approach *fix-weighted combination*, because each combination uses the same set of weight values $w = \{w_1, \ldots, w_5\}$ for all queries. The weights are assigned to the first 5 FVs in the order given by Table 3.3 (e.g., w_1 corresponds to depth buffer, w_2 corresponds to voxel, and so on).

Definition 2. *The fix-weighted combined distance is defined as:*

$$\delta_{fix\text{-}weighted}(q, o) = \sum_{i=1}^{F} w_i \frac{\delta_i(q, o)}{norm_i}$$

The experimental results showed that the set of weights $w^* = \{2, 1, 2, 1, 1\}$ provides the best performance, which is slightly better than the case of unweighted combination of FVs. The precision versus recall plot is shown in Figure 3.14. The R-precision for each case is also indicated in the chart.

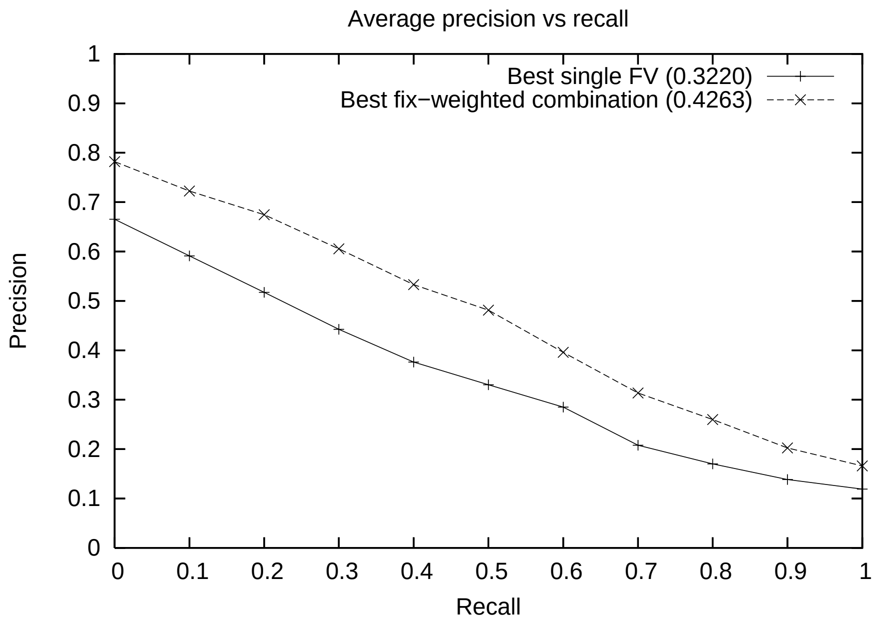

Figure 3.14: Average precision versus recall figures for the best fix-weighted combination and the best single feature vector

Although the weight vector w^* provides excellent retrieval performance, it is expected to be highly correlated to our database. Thus, it will probably not be useful for another 3D object database, because the optimal average weighting may be different. Moreover, in a dynamic database, it is not possible to determine the best weighting factors by experimentally analyzing all combinations of weighting factors for all possible queries. All these negative attributes make this approach impractical for real-world applications.

3.3 Dynamic combinations of feature vectors

The experimental results from the last section showed that resorting to combinations of FVs is a promising approach to improve the effectiveness of similarity queries in multimedia databases. However, a static combination of FVs will not necessarily provide optimal results, because if one of the considered FVs has a very bad effectiveness for the given query object, it will spoil the final result. Then, the problem is to determine which FVs to combine, as the inclusion of FVs irrelevant to q can harm the overall effectiveness of the search system. In general, different FVs provide the best effectiveness for different query objects. Thus, the similarity search system should give a higher priority to the best FVs for q to perform the similarity query.

To solve this problem, we propose to use *dynamic weighting methods* for the combinations of FVs, which aim to give a high weight only to those FVs that are most promising given a query object. In our proposed approach, the suitability of a FV is estimated against a training (or reference) database before performing the weighted combined query against the actual database.

We present two measures for the a priori estimation of individual FV performance, namely the *purity measure* and the *entropy impurity measure.* The values returned by these measures may be used for implementing dynamically weighted combinations of FVs.

Let $\mathbb{X}$ be the universe of valid multimedia objects and $\mathbb{U} \subseteq \mathbb{X}$ the database. Let $\mathbb{T} \subseteq \mathbb{U}$ be a *training set of classified objects*, where $\omega_j \subseteq \mathbb{T},\ 1 \leq j \leq M$, is a *model class* of objects (i.e., all objects in ω_j are considered similar), and $\mathbb{T} = \biguplus \omega_j$ (i.e., $\mathbb{T}$ is the disjoint union of the M model classes).

Let $q \in \mathbb{X}$ be a query object. Given a FV f, a *ranking* R^k_{fq} is a list of the k-NN of q from $\mathbb{T}$ with respect to f, sorted in ascending order by the distances to q. Figure 3.15 shows an example of a ranking for $k = 5$. Three of the k-NN belong to one of the model classes (dark grey) and the other two belong to another model class (light grey). The ranking will be the base for the proposed measures of FV performance.

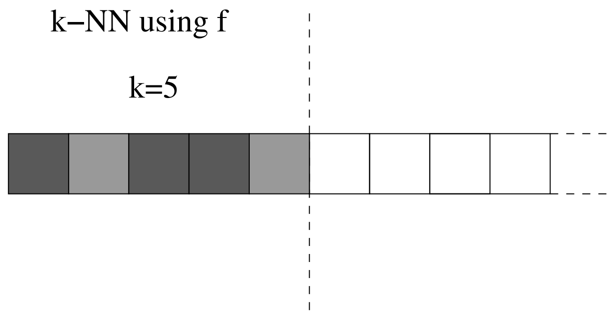

Figure 3.15: Example of a ranking R^k_{fq} using $k = 5$

3.3.1 Purity measure

Let S_i be the subset of objects from the ranking R^k_{fq} that belong to model class i, that is, $S_i = \omega_i \cap R^k_{fq}$.

Definition 3. *The* purity *of FV f for the query object q with respect to k is defined as*

$$purity(f, q, k) = \max_{1 \leq i \leq M} \{|S_i|\}.$$

The purity of a FV indicates the maximum number of objects that belong to a same model class in the first k positions of each ranking. This value aims to measure the coherence of the retrieved objects using FV f. Our hypothesis is that a well suited FV for the given query object will retrieve objects from the training set that belong to the same model class (i.e., its purity measure is high). On the other hand, if a FV retrieves objects from different model classes, then the answer is not coherent (i.e., its purity measure is low) and hence the FV is considered to be not suitable for this query object.

Considering the ranking illustrated in Figure 3.15, it follows that the purity of FV f is

$$purity(f, q, 5) = \max\{3, 2\} = 3.$$

Note that $|S_i| = 0$ for all model classes that are not represented on the ranking, thus they have no influence on the purity value of f.

3.3.2 Entropy impurity measure

The *entropy impurity* [Duda et al., 2001] is a well known measure used in the context of decision tree induction. It measures the impurity of a node N of a tree with respect to the elements assigned to N. If all these elements have the same class label then the impurity is 0, otherwise it is a positive value that increases up to a maximum when all classes are equally represented. (Other impurity measures are the *Gini impurity* and the *misclassification impurity* [Duda et al., 2001], but the best experimental results were obtained using entropy impurity.)

Let $P^k_{\omega_j}(R^k_{fq})$ denote the fraction of objects at the first k positions of R^k_{fq} that belong to class ω_j.

Definition 4. *The entropy impurity of a FV f with respect to q is defined as*

$$impurity(f,q,k) = -\sum_{j=1}^{M} \begin{cases} P^k_{\omega_j}(R^k_{fq})\log_2(P^k_{\omega_j}(R^k_{fq})) & \text{if } P^k_{\omega_j}(R^k_{fq}) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Given a ranking R^k_{fq}, if the k objects belong to the same class then the impurity of FV f is 0; otherwise it is a positive number, with its greatest value occurring when the number of classes covered by the k objects in R^k_{fq} is maximal.

Considering the ranking illustrated in Figure 3.15, it follows that the entropy impurity of FV f is

$$impurity(f,q,5) = -\left(\frac{3}{5}\cdot\log_2\left(\frac{3}{5}\right) + \frac{2}{5}\cdot\log_2\left(\frac{2}{5}\right)\right) \approx 0.29.$$

3.3.3 Dynamic selection and combinations of feature vectors

The previously defined purity and entropy impurity measures may be used to dynamically weight combinations of FVs, where the weight values depend on the query object. Let $\mathbb{F} = \{f\}$ be the set of available FVs ($|\mathbb{F}| = F$). We can use the performance estimators to combine FVs by computing a *dynamically weighted combination of FVs* or by *selecting and combining t FVs.*

Dynamically weighted combination of feature vectors

The obtained estimator value for FV f is used to compute its associated weight. After all weights are computed, the combined distance function is computed as the linear weighted combination of the distances using each FV.

As the FV performance estimation increases with purity, we directly use the purity values as weights for the combination. We subtract 1 from the purity value to make it equal to 0 if the purity of the FV is minimal (i.e., equal to 1).

Definition 5. *The* purity weighted distance *is defined as*

$$\delta_c(q,o) = \sum_{i=1}^{F} (purity(f_i,q,k) - 1) \cdot \frac{\delta_i(q,o)}{norm_i}.$$

As the FV performance estimation decreases with the entropy value and it has a maximum value equal to $\log_2(k)$, we use $\log_2(k) - impurity(\cdot)$ (which returns a value in $[0, \log 2(k)]$) as weight for the combination.

Definition 6. *The* entropy impurity weighted distance *is defined as*

$$\delta_c(q,o) = \sum_{i=1}^{F} (\log_2(k) - impurity(f_i,q,k)) \cdot \frac{\delta_i(q,o)}{norm_i}.$$

The combined distance function $\delta_c(q,o)$ is then used to perform the similarity query on the database.

Selection of t feature vectors

We use the estimator value to select the best t FV, and then we linearly combine them. This is equivalent to using binary weights $w \in \{0,1\}$ for the linear weighted combination of distances, setting $w_i = 1$ only if FV f_i is one of the t selected FVs.

The system selects the t most promising FVs according to the obtained performance estimation values with the purity or the entropy impurity measures. In the case of the *purity-based selection*, the t FVs with highest purity values are selected. In the case of the *entropy-impurity-based selection*, the t FVs with smallest entropy impurity values are selected. Both methods solve ties by selecting the best FV according to a precomputed ranking of FVs using, for example, a table with the average single FV performance (e.g., Table 3.3).

3.3.4 Experimental evaluation

We used the classified subset of our 3D model database as a training dataset for the purity and entropy impurity measures, and used all the implemented FVs for 3D models (16 in total) to perform the combinations. We computed the average R-precision values to compare the effectiveness of the proposed methods.

Figure 3.16 shows the obtained R-precision values as a function of parameter k with the purity-based methods. The figure shows the curve for the dynamically weighted combination of FVs and three different cases of selection of FVs: $t = 2$, $t = 3$, and $t = 8$ (we made experiments for $1 \leq t \leq 15$, and the best results were obtained with $t = 8$). It follows that the most effective method is the dynamically weighted combination of FVs. For all curves, the best effectiveness result was obtained with k between 3 and 5. The results did not vary much on the whole range of k values, but a sustained decrease of effectiveness can be observed for $k > 5$.

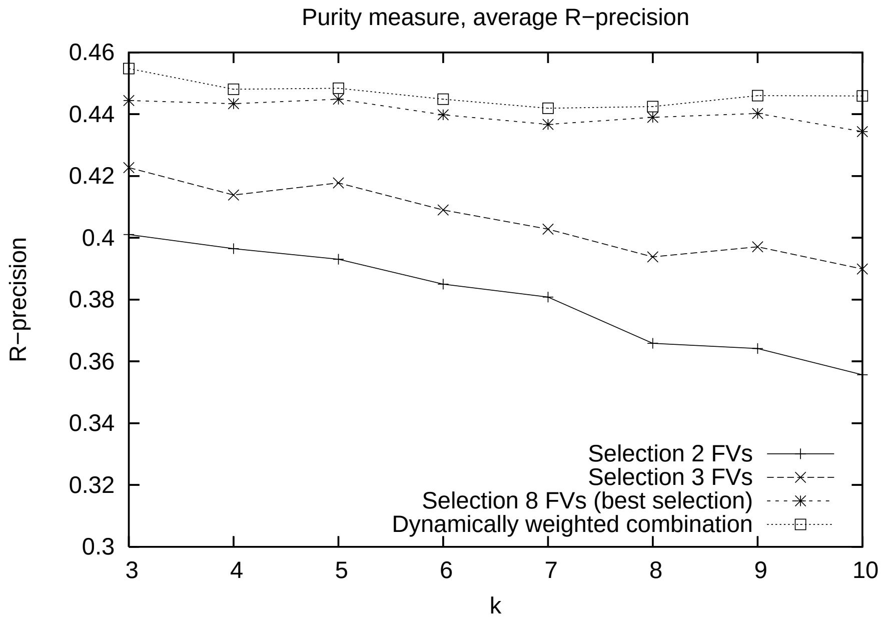

Figure 3.16: Average R-precision as a function of parameter k for the methods based on the purity measure

Figure 3.17 shows the effectiveness values with the entropy-impurity-based methods. The obtained results are very similar to those obtained with the purity-based methods. The best effectiveness values were also obtained with k between 3 and 5.

Table 3.5 presents a summary of the best experimental results with all the methods described in this chapter. By using dynamically weighted combinations of FVs, the overall effectiveness of the similarity search system improved significantly (41% compared with the best single FV). This improvement is much higher than the improvements obtained by using different single FV (cf. Table 3.3). An advantage of the dynamically weighted combination method compared with the selection method is that one does not need to fix in ad-

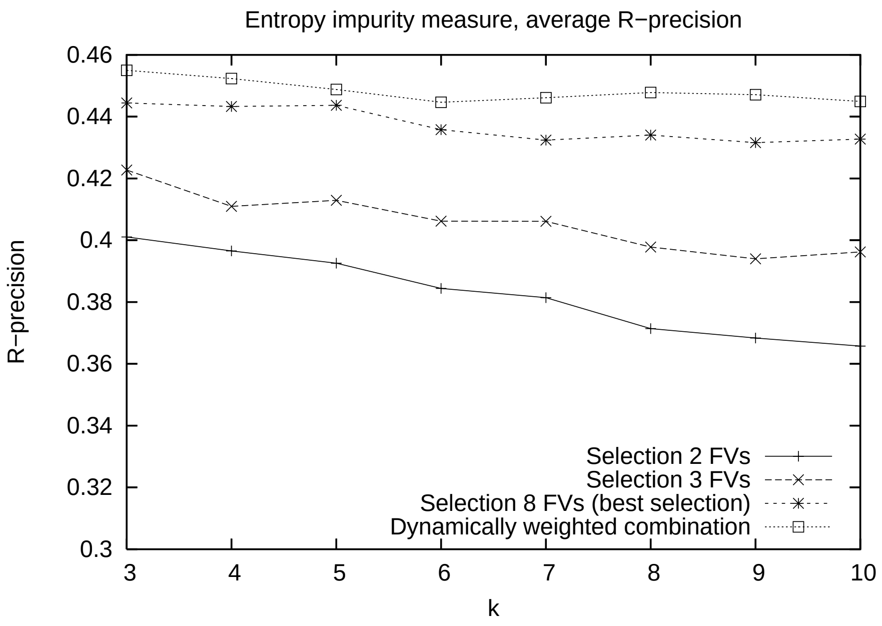

Figure 3.17: Average R-precision as a function of parameter k for the methods based on the entropy impurity measure

vance how many FVs must be selected (we needed to test all possibilities to find the optimal value $t = 8$), but the method automatically assigns a weight to each FV.

Method	Avg. R-precision	Relative improv.
Best single FV	0.3220	0%
Entropy impurity selection 1 FV (k=3)	0.3520	9%
Entropy impurity selection 2 FV (k=3)	0.4011	25%
Unweighted combination (5 FVs)	0.4220	31%
Entropy impurity selection 3 FVs (k=3)	0.4227	31%
Best fix-weighted combination	0.4263	32%
Purity selection 8 FVs (k=5, best selection)	0.4448	38%
Entropy impurity weighted combination (k=3)	0.4550	41%

Table 3.5: Summary of the average R-precision for the proposed methods

To discard that the selection of the model classes could have some influence on the obtained results, we qualitatively validated the obtained results running a cross-validation test [Han and Kamber, 2001]. We divided the set of the classified objects into two equally sized groups: A *query set* and a *training set*. The query set was used as in the described experimental framework (i.e., each object from this was used as a query, and the relevant objects for each query were the objects of the query set belonging to the same model class). The training set was considered to be outside of the database, that is, it was not considered for computing the effectiveness of the search system (it was only used to compute the weights). The partitioning of the classified set of objects was randomly performed, and we computed the average over 100 random partitions. The results of the cross-validation test confirmed the significant improvement in effectiveness of the dynamically weighted combination of FVs approach. An independent experimental evaluation using a different database [Ohbuchi and Hata, 2006] also confirmed the effectiveness of our proposed approach.

It is worth noting that we also tested a combination technique not based on distance aggregation, but on rank aggregation [Fagin et al., 2003]. In this framework, each considered FV is used to rank the objects for a given query. To each position in the rank, an inversely proportional score is assigned. The first position has the maximum score, the last position has the minimum score, and the scores monotonically decrease with the position in the ranking. The scores corresponding to each object in the different rankings are added up, and then they are sorted in descending order according to their aggregated score, obtaining the final ranking. We implemented a linear as well as a superlinear decreasing scoring function, but found that the effectiveness results obtained with this method were lower than the effectiveness score of the best single FV (depth buffer).

3.4 Conclusions

We experimentally evaluated the performance of 3D object descriptors using standard effectiveness measures from Information Retrieval (precision vs. recall diagrams and R-precision values). The experimental effectiveness comparison showed that there are a number of descriptors that have a good average effectiveness and work well in many query classes (e.g., depth buffer, voxel, and complex descriptors). Other descriptors work well with some specific model classes (e.g., shape spectrum with the human model class), and some of them are effective when the normalization step using PCA is not effective (e.g., harmonics 3D). However, we made the observation that no single FV dominates over all others in all situations, and that depending on the query object a different FV is the optimal one to use.

We found that by using combinations of FVs it is possible to significantly enhance the effectiveness of a similarity search system. However, a simple combination of all available FVs does not yield the best results, because a bad FV for a given query object may spoil the final result. Therefore, we proposed methods for dynamically weighting and combining FVs. The impurity and entropy impurity methods assess a priori the suitability of a FV based on a small training dataset. Algorithms for selecting the best FVs and for computing dynamic weights for the combination were described.

Our experimental results showed that both purity and entropy impurity methods allow us to significantly improve the effectiveness of the search system, especially with the dynamically weighted combination technique. We experimentally found that it is possible to improve the effectiveness by 36%, in terms of R-precision, with respect to the best single FV using the dynamic combination technique with a small set of good FVs (5 in our experiments). This improvement in retrieval effectiveness is significant, because the observed improvement gap between descriptors is on average 6%. The obtained results showed that similarity search systems may profit from automatic FV combination techniques.

We would like to remark that a static combination of FVs is an inferior technique compared to dynamically weighted combinations. Firstly, one needs to find the best static combination for a given database: If the database change, it is not clear if the chosen static combination will continue being the best one. Secondly, if the static combination includes a bad FV for some query objects, the effectiveness of the search system for those objects may be reduced. Finally, we made a experiment were an "oracle" told us which was the best combination of FVs for each query object (for this experiment, we computed the results with all possible combinations of FVs for each query object). Using the optimal combination per query, we got an average R-precision of 0.5997, far better than the best static combination (see Table 3.5). With dynamic combination methods we could potentially obtain results close to this optimum.

It is worth noting that the proposed techniques are general and not restricted to 3D objects, and that they can be used with any multimedia data type (images, audio, etc.) if an appropriate distance function is defined.

The proposed methods for combining FVs open the issue of efficiently supporting similarity queries that use dynamically weighted combinations of FVs, because the standard indexing techniques (cf. Chapter 2) have been primarily designed for a single FV. In Sections 4.3 and 4.4 we will study and propose solutions for this problem.

Chapter 4

Efficiency of feature based similarity search

Efficient query processing for similarity search in multimedia databases has been an active research area, and several metric [Chávez et al., 2001b] and spatial access methods [Böhm et al., 2001] have been already proposed. However, there are still many open problems in this area. This chapter presents novel solutions for some of these open problems, which include advanced techniques for searching in metric spaces and algorithms and index structures for multimedia objects described by a combination of feature vectors (FVs). We experimentally evaluate the proposed algorithms and data structures using a wide variety of synthetic and real datasets.

This chapter is organized as follows. Section 4.1 presents several pivot selection techniques for pivot-based indices. Section 4.2 describes an improved k-NN algorithm that lowers the memory requirements of the standard best-first search algorithm. In Section 4.3, we present a pivot-based index structure for dynamically weighted combinations of FVs. Finally, in Section 4.4 we introduce the problem of finding the optimal set of indices for frequently used combinations of FVs.

4.1 Pivot selection techniques

With few exceptions, similarity search algorithms in metric spaces based on the use of pivots (see Section 2.3.1) select them at random from the objects of the metric space. However, it is well known that the way in which the pivots are selected can drastically affect the performance of the algorithm. Between two sets of pivots of the same size, better chosen pivots can largely reduce the search time. Alternatively, a better chosen small set of pivots (requiring much less space) can yield the same efficiency as a larger, randomly chosen, set.

In this section, we propose an efficiency measure to compare two pivot sets, combined with an optimization technique that allows one to select good sets of pivots. The proposed criterion is based on the distance distribution of the metric space. We provide abundant empirical evidence, showing that the proposed pivot selection technique is effective. The proposed technique consistently produces good results in a wide variety of cases and is based on a formal theory. It is also shown that good pivots are outliers, but that selecting outliers does not ensure that good pivots are selected.

This work has been published in Bustos et al. [2003].

4.1.1 Motivation

Almost all proximity search algorithms based on pivots choose them randomly from the objects of the database. However, it is well known that the way pivots are selected affects the search performance [Micó et al., 1994; Faragó et al., 1993; Chávez et al., 2001b]. Some heuristics to choose pivots better than at random have been proposed, but in general all of them try to choose objects that are *far away* from each other. For example, Micó et al. [1994] propose to choose objects that maximize the sum of the distances between previously chosen pivots (see Section 4.1.6 for more details), Yianilos [1993] proposes a heuristic based on the second moment of the distance distribution which selects objects that are far away, and Brin [1995] proposes a greedy heuristic to select objects that are the farthest apart (note that this last structure does not select pivots, but "split points"). However, these heuristics only work in specific metric spaces and have a bad behavior in others. Faragó et al. [1993] show that, in $\mathbb{R}^d$ with the Euclidean metric, it is possible to find an optimal set of $d+1$ pivots, selecting them as the vertices of a sufficiently large regular d-dimensional simplex containing all the objects of the database. Unfortunately, this result does not apply to general metric spaces.

4.1.2 Efficiency criterion

Depending on how pivots are selected, they can filter out less or more objects. Thus, the first task is to define a criterion to tell which from two pivot sets is expected to filter out more, and hence reduce the number of distance computations carried out during a similarity query. Since the internal complexity is fixed, only the external complexity can be reduced. This is achieved by making the object candidate list (the list of objects that could not be discarded) as short as possible.

Let $(\mathbb{X}, \delta)$ be a metric space and let $\mathbb{U} \subseteq \mathbb{X}$ be the set of objects or database. A set of t pivots $\{p_1, p_2, \ldots, p_t\}$, $p_i \in \mathbb{U}$, defines a space $\mathbb{P}$ of distance tuples between pivots and objects from $\mathbb{U}$. The mapping of an object $u \in \mathbb{U}$ to $\mathbb{P}$, which will be denoted $[u]$, is carried out as $[u] = (\delta(u, p_1), \ldots, \delta(u, p_t))$. Defining the metric

$$\Delta_{\{p_1,\ldots,p_t\}}([x],[y]) = \max_{1 \leq i \leq t} |\delta(x, p_i) - \delta(y, p_i)|\,,$$

it follows that $(\mathbb{P}, \Delta)$ is a metric space, which turns out to be $(\mathbb{R}^t, L_\infty)$. Given a range query (q, r), the pivot exclusion condition (cf. Section 2.3.1) in the original space $\mathbb{U}$ becomes

$$\Delta_{\{p_1,\ldots,p_t\}}([q],[u]) > r$$

for the new metric space $(\mathbb{P}, \Delta)$. Figure 4.1 shows the mapping of the objects and the new exclusion condition.

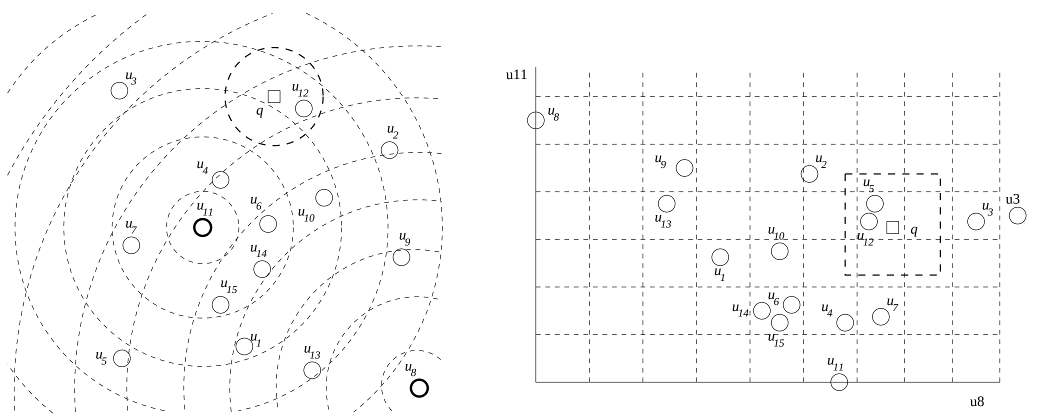

Figure 4.1: Mapping the objects from $(\mathbb{U}, \delta)$ onto $(\mathbb{P}, \Delta)$, using two pivots

To achieve a candidate object list as short as possible, the probability of $\Delta_{\{p_1,\ldots,p_t\}}([q],[u]) > r$ should be as high as possible. One way to achieve this is to maximize the mean of the distance distribution of Δ, which will be denoted μ_Δ (some alternative efficiency estimators were also tested, cf. Section 4.1.5). Hence, the set $\{p_1, \ldots, p_t\}$ is a better set of pivots than $\{p'_1, \ldots, p'_t\}$ when

$$\mu_{\Delta_{\{p_1,\ldots,p_t\}}} > \mu_{\Delta_{\{p'_1,\ldots,p'_t\}}}\,.$$

An estimation of the value of μ_Δ is obtained in the following way: A pairs of objects $\{(a_1, a'_1), (a_2, a'_2), \ldots, (a_A, a'_A)\}$ from $\mathbb{U}$ are chosen at random. All the pairs of objects are mapped to space $\mathbb{P}$, obtaining the set $\{\Delta_1, \Delta_2, \ldots, \Delta_A\}$ of distances Δ between every pair of objects. The value of μ_Δ is estimated as $\mu_\Delta = \frac{1}{A}\sum_{1\leq i\leq A} D_i$. It follows that $2t$ distance computations are needed to compute distance Δ for each pair of objects using t pivots. Therefore, $2tA$ distance computations are needed to estimate μ_Δ.

4.1.3 Pivot selection techniques

We present three pivot selection techniques based on the proposed efficiency criterion. Each technique has a cost measured in number of distance computations at index construction time. As more work in optimizing the pivots is done, better pivots are obtained. When comparing two techniques, the same amount of work to spend is given to them. We describe the optimization cost of each technique.

These selection techniques can be directly adapted to work with algorithms that use a fixed number of pivots, such as Fixed-height FQT, FQA, LAESA and Spaghettis. They can also be easily adapted to other pivot-based algorithms.

i. *Selection of N random groups.*

   N groups of t pivots are chosen at random among the objects of $\mathbb{U}$, and μ_Δ is calculated for each of this groups of pivots. The group that has the maximum μ_Δ value is selected.

   Optimization cost: Since the value of μ_Δ is estimated N times, the total optimization cost is $2tAN$ distance computations.

ii. *Incremental selection.*

   A pivot p_1 is selected from a sample of N objects of $\mathbb{U}$, such that that pivot alone has the maximum μ_Δ value. Then, a second pivot p_2 is chosen from another sample of N objects of $\mathbb{U}$, such that $\{p_1, p_2\}$ has the maximum μ_Δ value, considering p_1 fixed. The third pivot p_3 is chosen from another sample of N objects of $\mathbb{U}$, such that $\{p_1, p_2, p_3\}$ has the maximum μ_Δ value, considering p_1 and p_2 fixed. The process is repeated until t pivots have been chosen.

   Optimization cost: If the distances $\Delta_{\{p_1,\ldots,p_{i-1}\}}([a_r],[a'_r])$, $1 \leq r \leq A$, are kept in an array, it is not necessary to redo all the distance computations to estimate μ_Δ when the i^{th} pivot is added. It is enough to calculate $\Delta_{\{p_i\}}([a_r],[a'_r])$, $1 \leq r \leq A$, because it follows that

$$\Delta_{\{p_1,\ldots,p_i\}}([a_r],[a'_r]) = \max(\Delta_{\{p_1,\ldots,p_{i-1}\}}([a_r],[a'_r]), \Delta_{\{p_i\}}([a_r],[a'_r])).$$

Therefore, only $2NA$ distance computations are needed to estimate μ_Δ when a new pivot is added. Since the process is repeated t times, the total optimization cost is $2tAN$ distance computations.

iii. *Local optimum selection.*

A group of t pivots is chosen at random among the objects of the database. The matrix $M(r,j) = \Delta_{p_j}([a_r],[a'_r])$, $1 \leq r \leq A$, $1 \leq j \leq t$, is calculated using the A pairs of objects. It follows that $\Delta([a_r],[a'_r]) = \max_{1 \leq j \leq k} M(r,j)$ for every r, and this can be used to estimate μ_Δ. Also, for each row of M the index of the pivot with the maximum value, called r_{max}, and the second maximum value, called r_{max2}, must be kept. The *contribution contr* of the pivot p_j is the sum over the A rows of how much does p_j help increase the value of $D([a_r],[a'_r])$, that is, $contr = M(r, r_{max}) - M(r, r_{max2})$ if $j = r_{max}$ for that row, and $contr = 0$ otherwise. The pivot whose contribution to the value of μ_Δ is minimal with respect to the other pivots is marked as the *victim*, and it is replaced, when possible, by a better pivot selected from a sample of X objects of the database. The process is repeated N' times.

Optimization cost: The construction cost of the initial matrix M is $2At$ distance computations. The search cost of the victim is 0, because no extra distance computations are needed, all information is in M. Finding a better pivot from the X objects sample costs $2AX$ distance computations, and the process is repeated N' times, so the total optimization cost is $2A(t + N'X)$ distance computations. Considering $tN = t + N'X$, i.e., $N'X = t(N-1)$, the optimization cost is $2AtN$ distance computations.

Note that it is possible to exchange the values of N' and X while maintaining the optimization cost. In the experiments, two possible value selections were tested: $(N' = t) \wedge (X = N-1)$ (called *local optimum A*) and $(N' = N-1) \wedge (X = t)$ (called *local optimum B*). Another value selection was also tested, $N' = X = \sqrt{t(N-1)}$, but the obtained result did not show any improvement on the performance of the algorithm.

4.1.4 Experimental evaluation with synthetic datasets

The proposed selection techniques were tested on a synthetic set of random points in a k-dimensional vector space treated as a metric space, that is, the fact that the space has coordinates was not used, but the points were treated as abstract objects in an unknown metric space. The advantage of this choice is that it allows us to control the exact dimensionality the index is working with, which is very difficult to do in general metric spaces. The points are uniformly distributed in the unitary cube, the tests use the L_2 (Euclidean) distance, the dimension of the vector space is in the range $2 \leq dim \leq 14$, the database size is $n = 100,000$ (except when otherwise stated) and range queries were performed returning 0.01% of the total database size, taking an average from 10,000 queries. Results with real-world datasets are shown in Section 4.1.7.

About the parameters A and N of the optimization cost: The experimental results showed that, given an amount of work to spend, it is better to have a high value of A and a low value of N. This indicates that it is worth to make a good estimation of μ_D, while small samples of candidate objects suffice to obtain good sets of pivots. For the experiments in this section, these parameters have fixed values as follows: $A = 100,000$ and $N = 50$.

Comparison between the selection techniques

Figures 4.2 and 4.3 show the comparison between all of the selection techniques, when varying the number of pivots and keeping the dimension of the space fixed. These results show that the incremental selection technique and the local optimum A technique obtain the best performance in practice. Local optimum B works well only in spaces with low dimension and with few pivots, obviously influenced by the setting of parameter N. Selection of N random groups shows little improvement over random selection in all cases.

Although the incremental and local optimum A techniques give almost the same efficiency, the incremental selection have some advantages that makes it the favorite pivot selection technique: It is a much simpler technique and it allows us to easily add more pivots to the index. Also, the only way to determine the optimum number of pivots t^*, for a fixed tolerance range, is to calculate an average of the total complexity of the algorithm for different values of t, where t^* is equal to the value of t which minimizes the total complexity. That is, it is worth adding pivots to the index until the total complexity stops improving. The incremental selection allows us to add more pivots to the index at any time without repeating the optimization process, if the distances $\Delta_{\{p_1,\ldots,p_t\}}([a_r],[a'_r])$, $\forall r \in 1 \ldots A$ are kept. On the other hand,

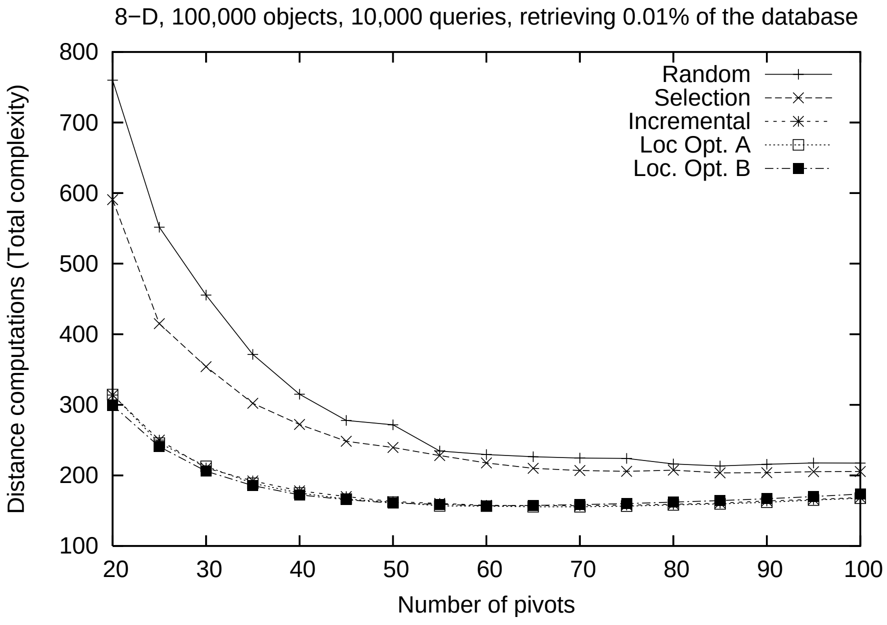

Figure 4.2: Comparison between selection techniques in an 8-dimensional random vector space

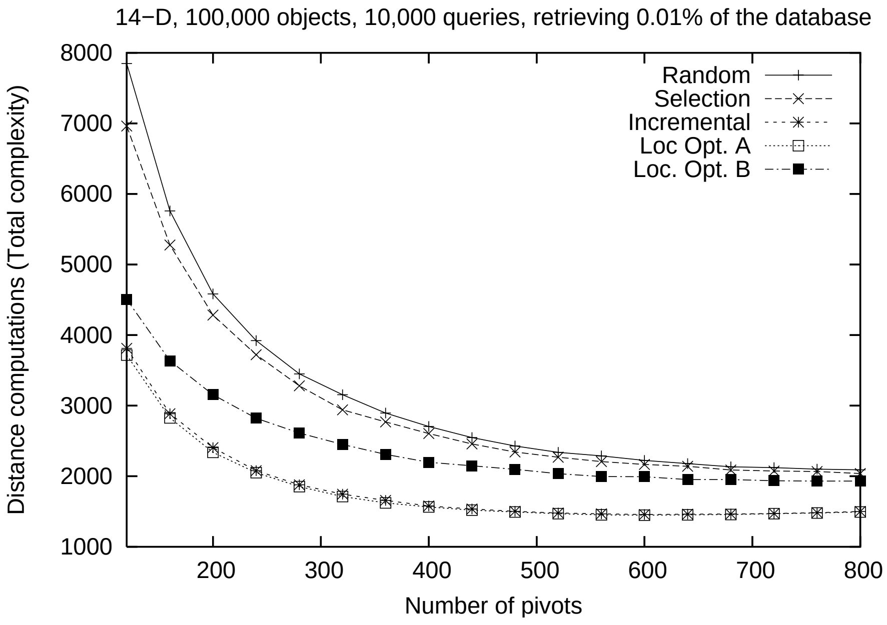

Figure 4.3: Comparison between selection techniques in a 14-dimensional random vector space

selection of N random groups and local optimum selection techniques must reperform the optimization process in order to obtain a new set of pivots, because these techniques cannot take advantage of the work done previously. For this reason, it is much easier to calculate the optimum number of pivots t^* using the incremental selection technique.

Comparison between random and good pivots

Figures 4.4 and 4.5 show a comparison for the internal and total complexity between random and incremental selection when using the optimum number of pivots for each technique. Figure 4.4 shows a comparison when varying the dimension of the space. Since k^* is equal to the internal complexity of the algorithm, it follows that not only the optimum number of pivots is lower when using the incremental selection, but also the total complexity of the algorithm. Figure 4.5 shows a comparison in an 8-dimensional vector space varying the database size. Again, the results show that the optimum number of pivots and the total complexity of the algorithm is lower when using the incremental selection. The obtained results show that the incremental selection technique effectively produces good sets of pivots.

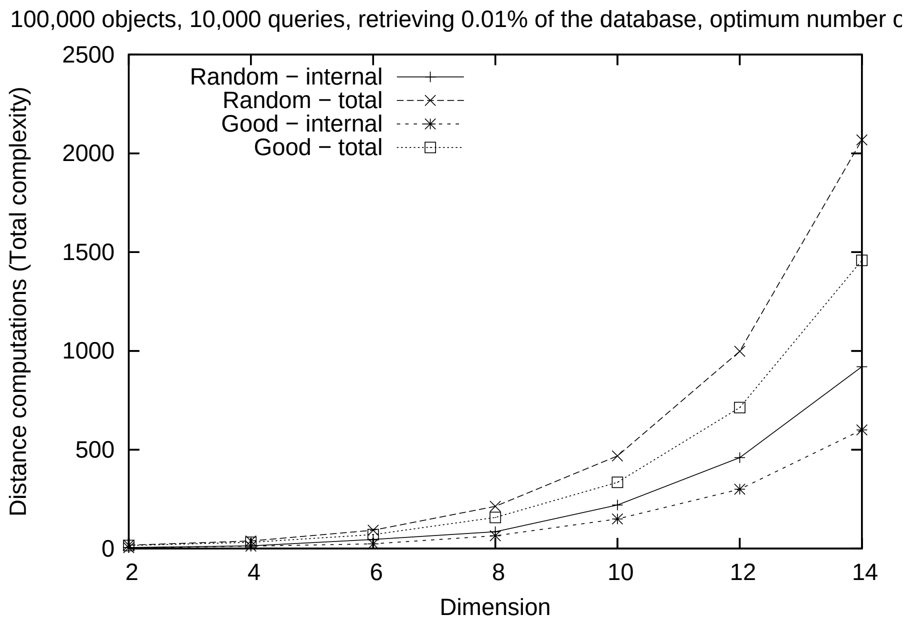

Figure 4.4: Comparison between random and good pivots when varying dimensionality

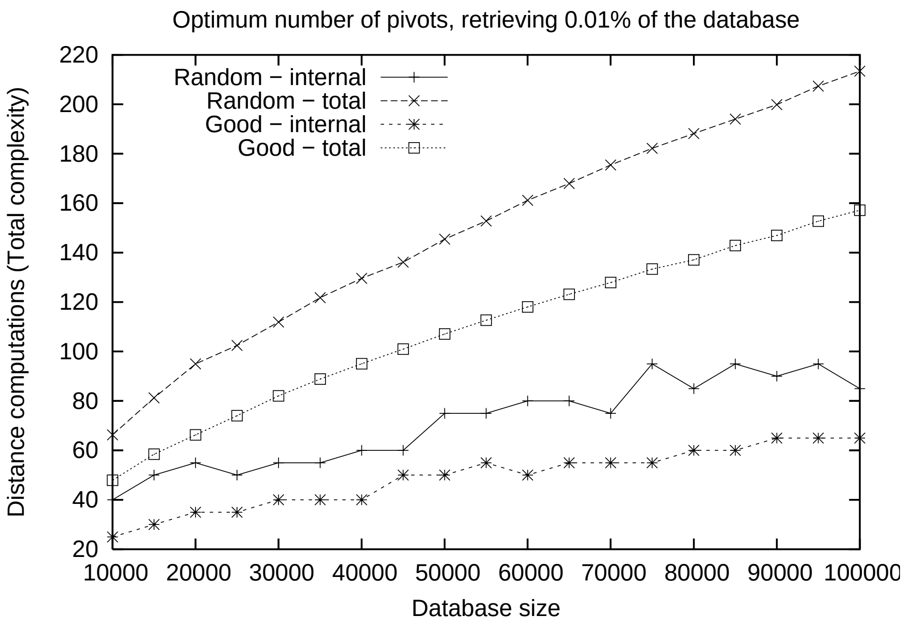

Figure 4.5: Comparison between random and good pivots when varying database size

Figure 4.6 shows the result of an experiment in a 30-dimensional vector space, where the elements have a Gaussian distribution, that is, the elements form clusters. The space is formed by 100 clusters, each of them centered at a random point in the space, and the variance for each coordinate is 0.001. The result shows that good pivots improve the performance of the search algorithm in comparison with random pivots.

The profit when using k^* pivots with incremental selection seems low in high dimensional spaces. However, consider that much fewer pivots (i.e., less memory) are needed to obtain the same result than with random selection. For example, in a 14-dimensional vector space, the optimum number of pivots using random selection is 920, while incremental selection only needs 280 pivots to achieve a better total complexity, hence saving almost 70% of the memory used in the index.

The optimization cost used in these experiments given by parameters A and N, may seem a little bit high. However, it is possible to obtain good results with a fraction of the used optimization cost. Figure 4.7 shows the results of an experiment in a uniform 8-dimensional vector space, using the optimum number of good pivots and varying parameter A from 100 to 100,000. The results shows that for values higher than 10,000 the improvement is negligible. Even when using a value as low as $A = 100$, we observed

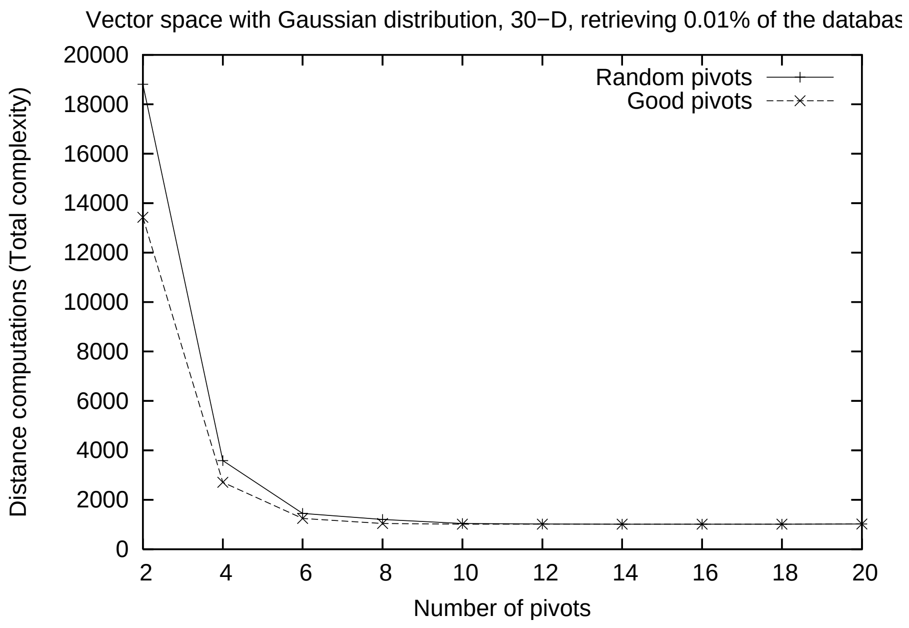

Figure 4.6: Experimental results with a vector space with Gaussian distribution

an improvement of 12% in the total complexity over random pivots.

4.1.5 Alternative efficiency estimators

Another possibility for maximizing the probability of the pivot exclusion condition is trying to reduce the variance of the distribution of Δ, σ^2_Δ, at the same time μ_Δ is maximized. To accomplish this, one can try to maximize the *intrinsic dimension* of the space $\mathbb{P}$, defined as $\mu_\Delta/2\sigma^2_\Delta$ (see Section 2.5.1). Another possible efficiency estimator is to maximize the minimum value of the distribution of Δ. This aims to shift the distance distribution to the right as much as possible.

Figure 4.8 shows the results of an experiment in a synthetic 8-dimensional vector space, comparing the two additional efficiency estimators against the original one. The figure shows that the original estimator selects better sets of pivots compared with the others, which even cannot do better than random selection for more than 40 pivots.

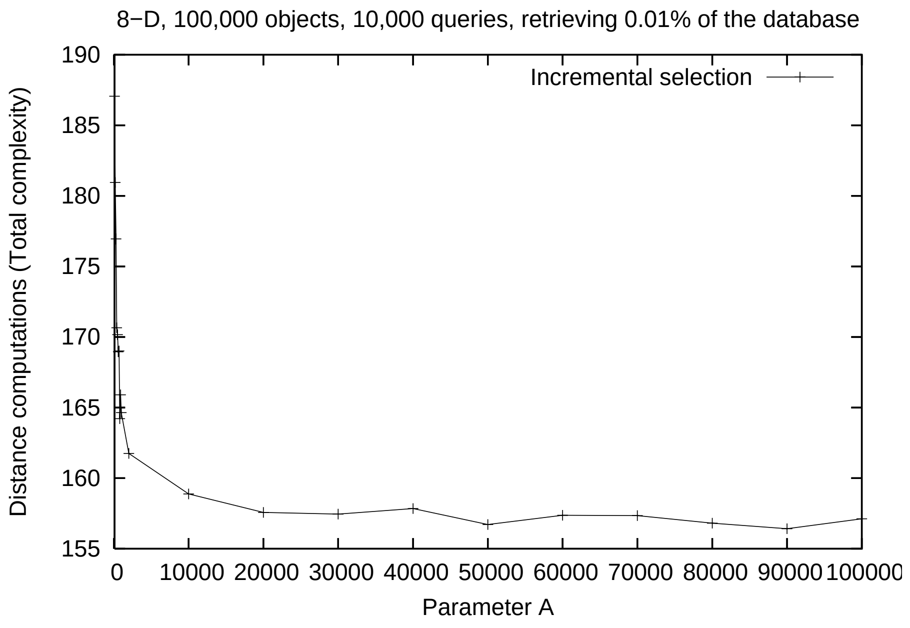

Figure 4.7: Efficiency of the selection technique when varying parameter A

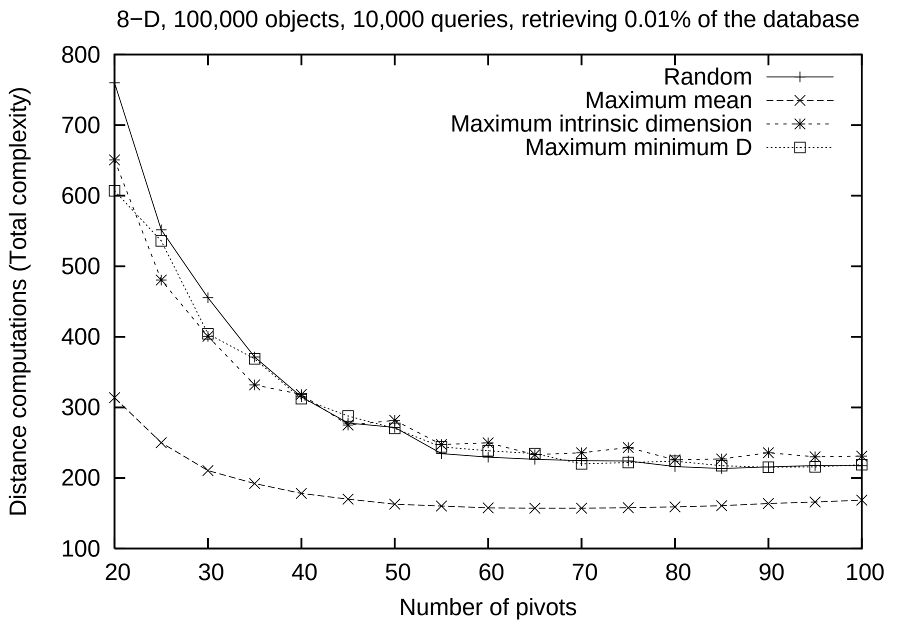

Figure 4.8: Comparison between different efficiency estimators

4.1.6 Properties of a good set of pivots

When studying the characteristics of the good sets of pivots, it was found that good pivots are *far away* from each other, i.e., the mean distance between pivots is higher than the mean distance between random objects of the metric space, and also that good pivots are *far away* from the rest of the objects of the metric space. The objects that satisfy these properties are called *outliers*. It is clear that pivots *must be far away from each other*, because two very close pivots give almost the same information for discarding objects. This is in accordance with previous observations [Faragó et al., 1993; Yianilos, 1993; Brin, 1995].

Then, it can be assumed that good pivots are outliers, so a new selection technique could be as follows: Use the same incremental selection method with the new criterion of selecting objects, which maximize the sum of the distances between the pivots previously chosen, selecting the first pivot at random. This selection technique will be called *outlier selection*, and it was already proposed in Micó et al. [1994]. It carries out $(i-1)N$ distance computations when the i^{th} pivot is added, where N is the size of the sample of objects from where a new pivot is selected. Hence, the optimization cost of this technique is $\frac{t(t-1)}{2}N$.

It is important to note that outlier selection *do not use the efficiency criterion described in Section 4.1.2*, because this alternative selection technique maximizes the mean distance in the original space and the efficiency criterion maximizes the mean of the distance Δ. These criteria do not always go together.

Figures 4.9 and 4.10 show the results obtained when comparing incremental and outliers selection techniques in random 8-dimensional and 14-dimensional vector spaces, respectively. The figures show that the outlier selection has slightly better performance than the incremental selection. This result can lead to the conclusion that outlier selection is the best pivot selection technique, but in the next section it will be shown that this assumption is not true for general metric spaces.

4.1.7 Experiments with real data

This subsection presents three examples of the use of the proposed efficiency criterion and the outlier selection, where the objects of the metric space are not uniformly distributed. The incremental selection technique is used to select good pivots. The local optimum A technique was also tested with these databases, obtaining slightly better results compared with the incremental selection. However, the incremental selection technique is preferred over the

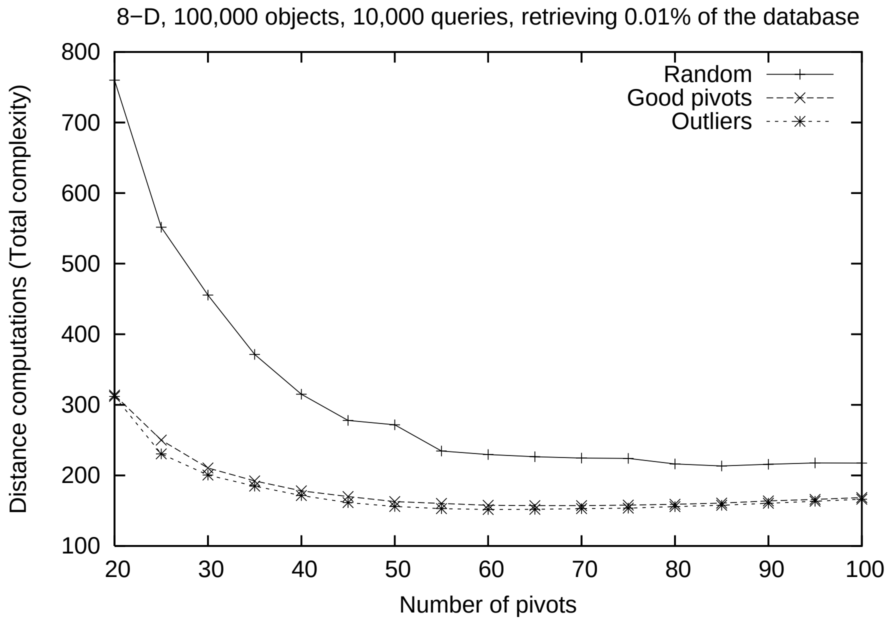

Figure 4.9: Comparison between incremental and outliers selection techniques in 8-dimensional random vector spaces

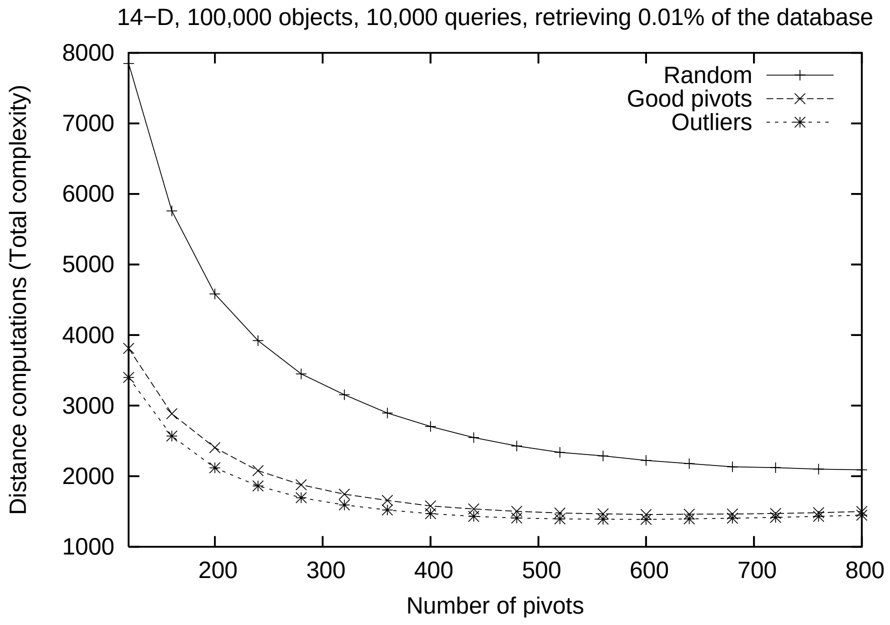

Figure 4.10: Comparison between incremental and outliers selection techniques in 14-dimensional random vector spaces

local optimum technique for the reasons stated in Section 4.1.4.

Figure 4.11 shows the results of the experiment over a string space, that is, the objects of the database were strings taken from an English dictionary of 69,069 terms, and 10% of the database was used as the query set. The distance function used was the *edit distance* (the minimum number of character insertions, deletions and substitutions to make two strings equal), and the tolerance range was set to $r = 2$, which retrieves an average of 0.02% of the database size per query. For this dataset, the incremental selection improved the performance of the algorithm more, with respect to random pivots, compared to the outlier selection.

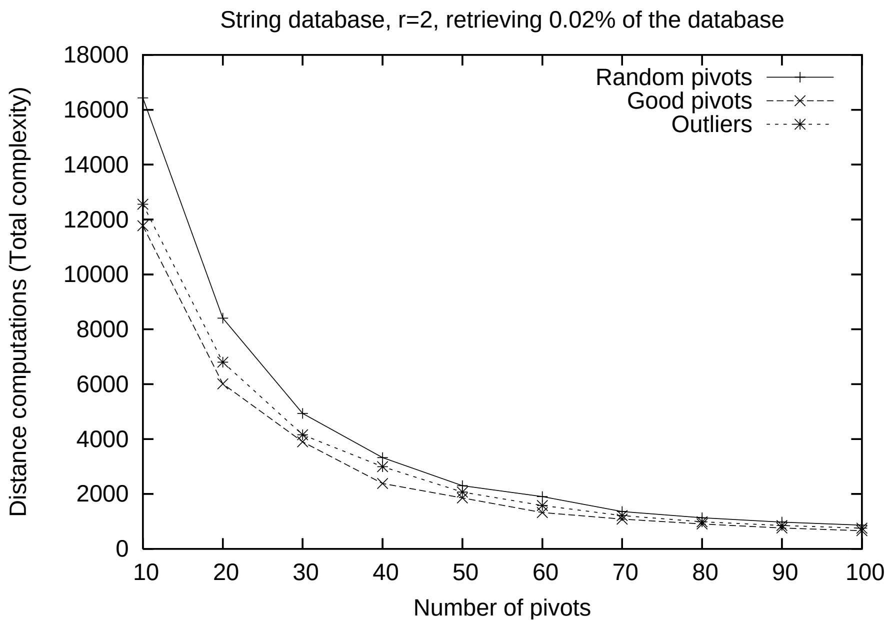

Figure 4.11: Experimental results with a string database

Figure 4.12 shows the results of the experiment when the objects of the database are a set of 40,700 images from NASA archives[1]. Those images were transformed into 20-dimensional vectors, and 10% of the database was defined as the query set. The tolerance range was set to return on average 0.10% of the objects of the database per query. The figure shows that for more than 25 pivots the outliers selection technique had a poorer performance than the random selection, while incremental selection always performed better.

Figure 4.13 shows the result of the experiment with a database of 112,682 color images, where each image is represented by a 112-dimensional feature

[1]Source: 6th DIMACS Implementation Challenge: Available Software. `http://www.dimacs.rutgers.edu/Challenges/Sixth/software.html`.

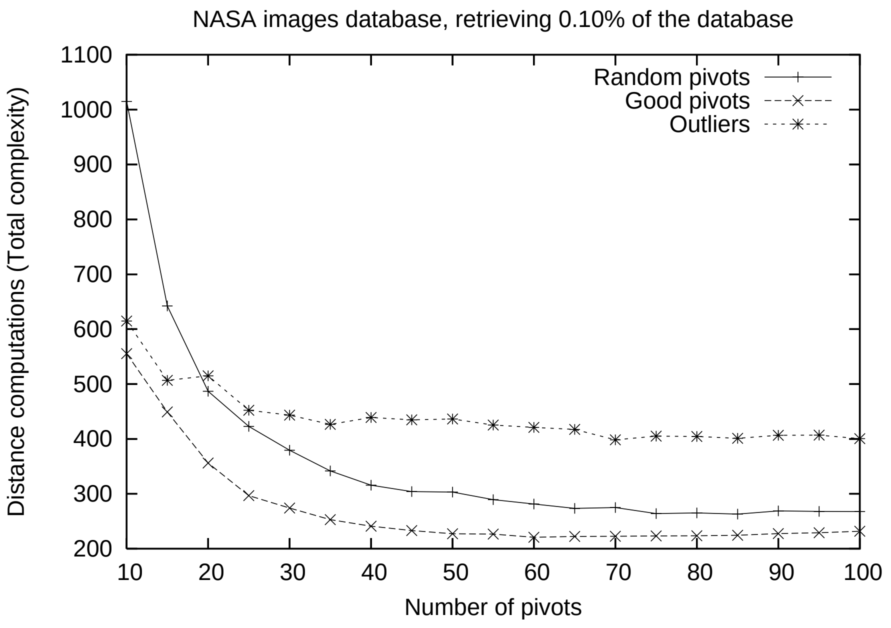

Figure 4.12: Experimental results with the NASA images database

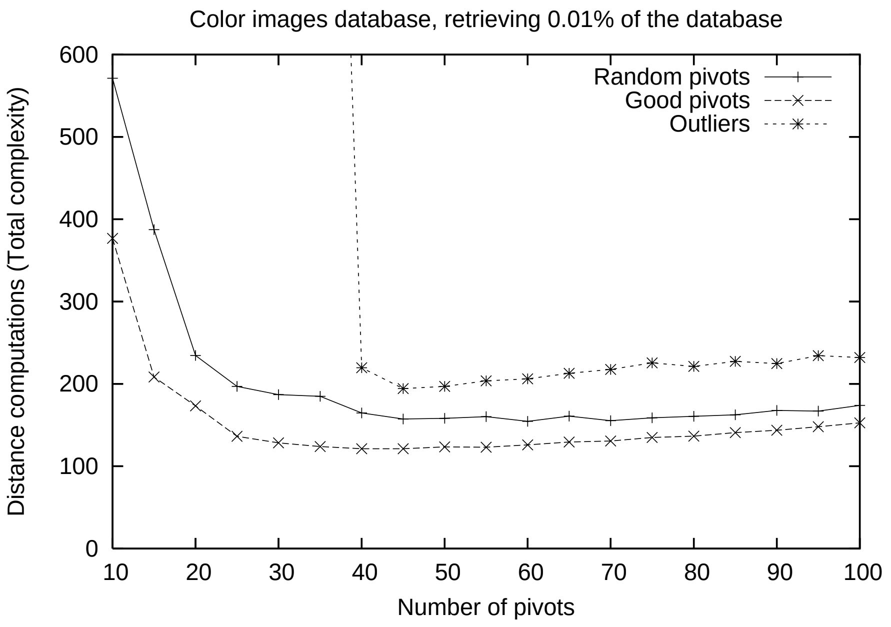

Figure 4.13: Experimental results with a color image database

vector. A 10% of the database was used as the query set. The result shows that good pivots performed better than random pivots, but outliers performed worse than random pivots. In fact, with less than 40 pivots the results using outliers were an order of magnitude worse than with random pivots.

The last two results are in contrast with those obtained on uniformly distributed vector spaces.

4.2 Improved k nearest neighbor algorithm

One of the most important similarity queries in multimedia databases is the k nearest neighbor (k-NN) search. The standard best-first k-NN algorithm uses the lower bound distance to filter out objects during the search. Although optimal in several aspects, the disadvantage of this method is that its space requirements for the priority queue that stores unprocessed clusters can be linear in the database size. Most of the optimizations used in spatial access methods (e.g., filtering out using MinMaxDist) cannot be applied in metric spaces, due to the lack of geometric properties.

We propose a new k-NN algorithm that uses *distance estimators*. The new algorithm aims to reduce the storage requirements of the search algorithm. An experimental evaluation with synthetic and real datasets confirms the savings in storage space of our proposed algorithm.

This work has been submitted for publication [Bustos and Navarro, 2006].

4.2.1 Motivation

Unlike the case of range searching, where the tree traversal order is irrelevant, for k-NN search we wish to find close candidates as soon as possible, as this will determine how much of the tree is traversed. The most common traversal order is *depth-first* traversal. In this case, the tree traversal is recursive, and the criterion to try to find soon good k-NN candidates translates into traversing the children of the current node from most to least promising. Given a criterion to prefer one node over another, depth-first traversal does not achieve the optimal node traversing order because it is forced to be depth-first. On the other hand, the amount of memory it requires does not exceed the height of the tree index.

An optimal, *best-first* traversal ordering uses a global priority queue where unprocessed tree nodes are inserted, giving higher priority to the more promising ones, and the tree is traversed in the order given by the queue [Uhlmann, 1991a; Hjaltason and Samet, 1995]. Each extracted node inserts its children in the queue. If this priority is defined as a *lower bound to the distance* between q and any element in the subtree rooted at the tree node, the search can finish as soon as the most promising node has a lower bound larger than the distance to the current k^{th} NN. This algorithm has been proved to be optimal in the number of required page accesses [Böhm et al., 2001]. It was also proved to be *range-optimal*, that is, the number of distance computations to find the k-NN answer is exactly that of a range search with distance $d(q, o_k)$, where o_k is the k^{th} NN. This range search would give the same answer as the k-NN search, so there is no penalty for not knowing $d(q, o_k)$ beforehand [Hjaltason and Samet, 2000].

The algorithm has, however, an important problem, which may complicate its use in practice. The problem is the amount of memory required for the queue, which can store *as many elements as the database itself.* The non-optimal depth-first-search algorithm may be preferable because of its much lower space consumption, which is proportional to the depth of the hierarchy. Samet [2003] recently proposed a technique to alleviate this problem, based on computing an upper bound to the distance between q and any subtree element, and using the fact that one knows that a subtree must have at least one element within that upper bound distance to q. We refine this idea, which will be described later as a particular case of ours. Our refinement consists in using the information on the number of elements in a region of the space, all of which lie within that upper bound. Unlike Samet's approach, ours translates into a relevant contribution even on vector spaces, where Samet's approach is never better than the MinMaxDist estimator.

4.2.2 Proposed best-first k-NN algorithm

In this section, we describe the standard best-first k-NN search and present in detail our improved k-NN algorithm. Table 4.1 lists the notation used throughout this section.

We assume that the index is a *hierarchical data structure* which groups close objects in clusters. We will refer to these clusters in our metric space as *balls*, which resembles a cluster's shape in an Euclidean vector space. A ball B from the index contains a number of objects from the database, represented by $B.bsize$. The *center* of B is a distinguished object $B.c \in B$, usually selected trying to minimize the *covering radius* of B, $B.cr = \max\{b \in B,\ d(B.c, b)\}$, that is, the maximum distance between $B.c$ and any other object in B. Those elements $b \in B$ can be recursively organized into balls, which descend from B forming a *search hierarchy.* This type of hierarchical clustering index is very popular [Kalantari and McDonald, 1983; Dehne and Noltemeier, 1987; Noltemeier et al., 1992; Brin, 1995; Ciaccia et al., 1997;

Symbol	Definition
$\mathbb{X}$	Universe of valid objects
$\mathbb{U} \subseteq \mathbb{X}$	Database
$\delta(x, y)$	Distance between x and y
$q \in \mathbb{X}$	Query object
$k \in \mathbb{N}$	# of NN to be retrieved
$B \subseteq \mathbb{U}$	Ball (cluster of objects)
$B.c \in \mathbb{U}$	Center of ball B
$B.cr \in \mathbb{R}$	Covering radius of B
$B.bsize \in \mathbb{N}$	# objects inside B
$B.children$	Set of children balls of B
$B.lbound \in \mathbb{R}$	Lower bound distance from B to q
$B.ubound \in \mathbb{R}$	Upper bound distance from B to q
B_b	Bubble of B
$B_b.csize \in \mathbb{N}$	Size of bubble B_b in C
Q	Queue with unprocessed balls
C	Queue with NN candidates
$c.distq \in \mathbb{R}$	Distance from c to q
$C.maxUB \in \mathbb{R}$	Maximum upper bound distance in C
$C.size \in \mathbb{N}$	Sum of all $B_b.csize$ in C

Table 4.1: Notation used on Section 4.2

Navarro, 2002; Chávez and Navarro, 2005]. There are other indexes that, although less obviously, can also be considered as belonging to this scheme [Burkhard and Keller, 1973; Yianilos, 1993; Baeza-Yates et al., 1994; Bozkaya and Ozsoyoglu, 1997; Yianilos, 1999].

Given a query q and a ball B, the *lower bound distance* from q to B, $B.lbound$, is a lower bound to $d(q, b)$ for any $b \in B$. Similarly, the *upper bound distance* from q to B, $B.ubound$, is an upper bound to $d(q, b)$ for any $b \in B$[2]. Figure 4.14 illustrates both bounds in 2D space using the Euclidean distance and the covering radius. On index structures for vector spaces that use minimum bounding rectangles (MBRs), the lower (upper) bound distance can be defined as the minimum (maximum) distance from q to the MBR. These distance bounds can be used to filter out balls while performing a similarity query. For example, if we know that the upper bound distance to the k^{th} NN candidate at some point of the search is $maxUB$, and that $maxUB < B.lbound$ for a ball B, then it is not possible that an object inside the ball is closer to q than any of the current k-NN candidates. Thus, one can safely discard B and all its descendents.

[2]Our proposed algorithms are general and work with any hierarchical index structure with appropriately defined distance estimators. For simplicity, we describe them using the covering radius for computing the distance bounds.

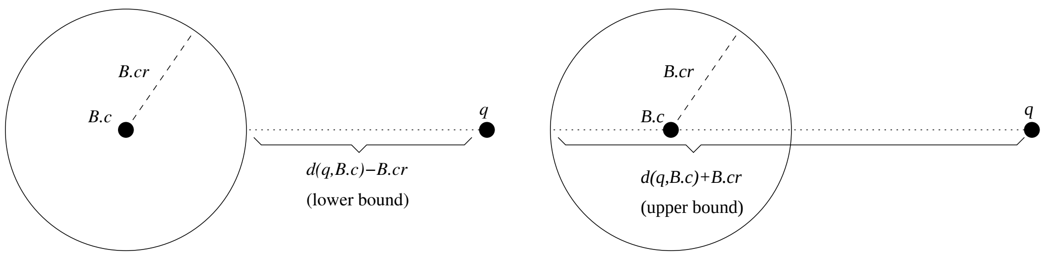

Figure 4.14: Distance estimators: Lower and upper bound distance from q to any object on the ball

Standard best-first k-NN algorithm

The best-first k-NN search algorithm [Hjaltason and Samet, 1995] uses two queues, one (Q) that contains the balls not yet processed (also called *active page list* in the literature), and the other (C) with the k-NN candidate list. A ball is stored in Q if it has not yet been processed but its parent has already been processed. At each step of the search, the algorithm removes the ball B from Q with smallest *B.lbound*. The distance between the center of each child of B and q is computed, inserting in C all centers that are closer than the current k^{th} NN candidate. The child balls are inserted into Q. The algorithm ends when Q becomes empty or when the minimum *lbound* from a ball in Q is greater than the distance to q of the current k^{th} NN candidate, as at this point no other ball can improve the current candidate list.

Algorithm 4.1 depicts the algorithm. We use the normal priority queue operations on max-queue C (initialize empty, *Add*, max, *DequeueMax*) and min-queue Q (initialize empty, *Add*, *DequeueMin*). Note that the size of C never exceeds k, but Q can be as large as the database size. As explained before, this algorithm is optimal in several aspects but it has a serious memory usage problem (for Q), which our proposal seeks to alleviate.

Description of our proposed algorithm

As in the standard k-NN algorithm, we use two priority queues, Q and C. Q contains unprocessed balls whose center have already been processed, and is sorted by *lbound*. C is sorted by *ubound*. This time, however, C will contain a mixture of objects and *bubbles*. A *bubble* B_b in C corresponds to ball B that exists in Q, but the bubble itself does not contain any element. From the bubble B_b we only know the upper bound *B.ubound* and size *B.bsize* of its corresponding ball B (i.e., *B.bsize* indicates the number of objects $u \in \mathbb{U}$ that are inside B). The existence of bubble B_b in C just tells us that there are *B.bsize* elements at a maximum distance of *B.ubound* from q, yet we

Algorithm 4.1: Standard k-NN search

```
Input: $Index$, $q \in \mathbb{X}$, $k \in \mathbb{N}$
Output: $k$-NN
$Q \leftarrow \emptyset$;
$C \leftarrow \emptyset$;
$B \leftarrow$ root of $Index$;
$B.c.distq \leftarrow d(q, B.c)$;
$B.lbound \leftarrow B.c.distq - B.cr$;
$Q.Add(B, B.lbound)$;
$C.Add(B.c, B.c.distq)$;
while $Q \neq \emptyset$ do
    $B \leftarrow Q.DequeueMin()$;
    if $|C| = k \wedge B.lbound \geq C.Max().distq$ then break;
    foreach $B' \in B.children$ do
        $B'.c.distq \leftarrow d(q, B'.c)$;
        $B'.lbound \leftarrow B'.c.distq - B'.cr$;
        if $|C| < k \vee B'.lbound < C.Max().distq$ then
            $Q.Add(B', B'.lbound)$;
            $C.Add(B'.c, B'.c.distq)$;
            if $|C| = k + 1$ then $C.DequeueMax()$;
return $C$
```

still do not know those elements. With this upper bound information, we can filter out irrelevant elements of Q earlier. For example, assume we find B such that $B.bsize > k$. Before knowing the elements of B, we find B' such that $B'.lbound \geq B.ubound$. At this point we can discard B', as we know that we will get enough better k-NN candidates from B, even when we still have not obtained them.

Our algorithm maintains the following invariants. We assume that there are at least k elements in the database, otherwise the query is trivial. For simplicity, we assume that balls do not directly contain objects, just additional balls. The last balls of the tree contain balls that have only one element, so the balls become empty once their centers are removed. This does not restrict the algorithm in any way, it is just a way to present it.

(i) We process the database hierarchy starting at the root, and never process a node without having processed its parent.

(ii) Any hierarchy ball not already processed is in Q or descends from a ball in Q, yet the centers of balls in Q have already been processed.

(iii) Any object c in C is the center of a ball already processed, $c.csize = 1$.

(iv) Any bubble B_b in C corresponds to a ball B currently in Q, $B_b.csize = B.bsize - 1$.

(v) $C.size \geq k$ is the sum of *csize*'s of objects and bubbles in C. $C.maxUB$ is the maximum *ubound* in C, taking $c.ubound = c.distq$ for objects.

(vi) C contains the objects and bubbles with smallest *ubound* processed so far.

(vii) If we remove any element from C with *ubound* equal to $C.maxUB$, then $C.size < k$.

The above invariants ensure the correctness of the following termination conditions:

- Assume $Q = \emptyset$ at some point. Then we have processed all the database objects (i, ii). Moreover, there cannot be bubbles in C (iv), so C contains just objects, of $csize = 1$ (iii). Therefore, C contains exactly k objects (v, vii), and those are the objects with smallest *distq* in the database (vi). Thus C is the correct answer to the query.
- Assume, at some point, that B has the smallest *lbound* in Q and $B.lbound \geq C.maxUB$. Since $ubound \geq lbound$ for any element and *lbound* for a descendant of B can never be smaller than $B.lbound$, condition (vi) holds for all the database, not only for the elements processed so far (ii). Moreover, C cannot contain any bubble B'_b, because $B'_b.lbound < B'_b.ubound \leq C.maxUB \leq B.lbound$ for any ball B in Q, and ball B' must be in Q (iv). Thus the same arguments as before show that C is the correct answer to the query.

From the second point we also see that, if $B.lbound \geq C.maxUB$ for *any* B, then the output of the algorithm does not vary if we remove B and all its descendents from Q. This is the key to reduce the storage requirement of Q.

We explain now how we set and maintain the invariants throughout the algorithm. We initialize Q with the only ball that roots the whole index (for simplicity we assume there is only one such root, it is easy to insert several roots if so is the index structure). Its center and corresponding bubble are inserted in C. This satisfies all the invariants. At each step of the algorithm, we extract the ball B from Q with the smallest $B.lbound$. Recall that $B.c$ has already been processed. Now, to restore invariant (ii), we must insert in Q every child B' of B such that $B'.lbound < C.maxUB$ (otherwise, we know that the descendents of B' can be immediately filtered out from the search). Then, to restore invariants (iii, iv), we must insert in C every center $B'.c$ and bubble B'_b, as well as remove bubble B_b from C, if present. Actually, if B_b is in C, we are replacing it with the centers $B'.c$ and bubbles B'_b, which add up the same $B_b.csize$. These replacements/insertions cannot therefore affect invariants (v, vi) as the new *ubounds* are never larger than that of B_b. Yet, we have to restore invariant (vii). We must remove from C the elements with largest *ubound* as long as $C.size \geq k$. We choose those with largest *ubound* so as to maintain (vi), and update $C.size$ and $C.maxUB$ to maintain (v). The remaining invariant (i) holds because we only access B' from its already processed parent B. Although not necessary for the correctness of the algorithm, we remove balls of Q that become irrelevant each time $C.maxUB$ is reduced. This diminishes the memory requirement for Q.

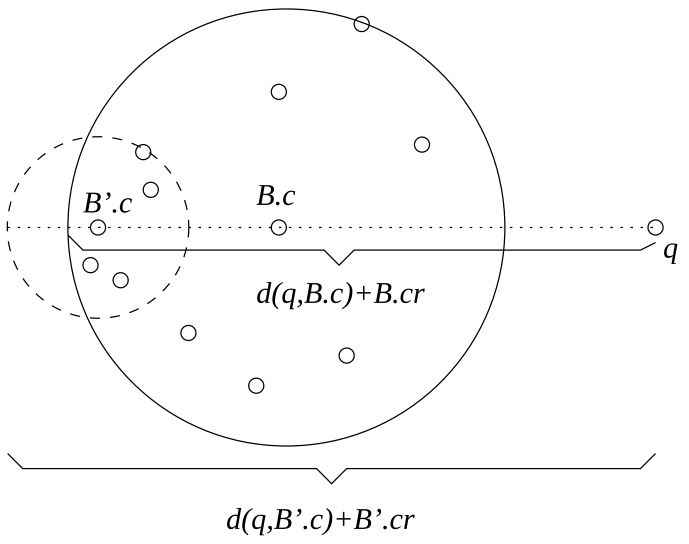

Figure 4.15: The correct *ubound* for B' is $\min\{B.ubound,\ d(q, B'.c) + B'.cr\}$

Algorithm 4.2 shows the pseudocode of the proposed k-NN search algorithm. Note that we enforce that *lbound* is increasing and *ubound* is decreasing as we descend in the hierarchy. Although this is true, it might not occur automatically if we simply use, for example $B'.ubound = d(q, B'.c) + B'.cr$ for B' child of B, because the ball of B' could spatially exceed that of B, although we know that there cannot be objects of B' in the exceeded area (Figure 4.15 illustrates). Thus the correct value is $B'.ubound = \min(B.ubound,\ d(q, B'.c) + B'.cr)$. We make the correction only if B_b is in C, otherwise $B.ubound > C.maxUB$ and the correction is irrelevant. *Add*2 is a special insertion procedure into C, which in addition to the element and its *ubound* gives the *csize* of the element. *Add*2 takes care of updating $C.size$ and $C.maxUB$, and of maintaining invariants (vi, vii). Algorithm 4.3 shows the pseudocode of the *Add*2 function for C.

Algorithm 4.4 shows the pseudocode for *Shrink* on Q, called each time the maximum upper bound distance $C.maxUB$ might change, to reduce the storage requirements of the search algorithm. Note that, thanks to the use of *Shrink*, we can always finish when Q becomes empty, since if the other termination condition holds, then *Shrink* will take care of removing all of the remaining elements from Q. To reduce the CPU cost associated with *Shrink*, it should be called when $C.maxUB$ has changed (not when it might have changed, as shown for simplicity), and implement Q as a min-max heap.

A key element of our k-NN algorithm is $B.bsize$. If this value is not stored in the index, the proposed algorithm cannot run and the best that one can do in that case is to assume $B.bsize = 2$ for internal hierarchy nodes (since $B.bsize \geq 2$, for the center and at least another point). This is precisely what was done by Samet in previous work [Samet, 2003], and we refine that work here assuming $B.bsize$ is known. Slightly better than assuming $B.bsize = 2$ is, if B has cb child balls and co child objects, assume $B.bsize = 2 \cdot cb + co$.

Example

Figure 4.16 shows an example of a k-NN query. (Note visually that all k-NN are on $B1$ and $B1.ubound < B.c.distq$.) The index consists of a ball B which has three child balls: $B1$, $B2$, and $B3$. From the figure, it follows that $B.bsize = 1+k+X+Y$. The figure shows the index hierarchy up to the first level. At the beginning, $B.c$ is inserted into C as well as the bubble B_b with upper bound $B.c.distq+B.cr$. Thus, $C.size = 1+k+X+Y \geq k$ and if we extract the bubble with the larger *ubound* then $C.size = 1 < k$, thus the invariants hold. Ball B is inserted into Q and the algorithm enters the loop. Ball B is extracted from Q, and then B_b is removed from C. Now the algorithm processes the children of B. The object $B1.c$ and the bubble $B1_b$ are inserted into C. Meanwhile, $B.c$ is removed from C because $B1.ubound < B.c.distq$ and $C.size = k \geq k$ with $B1_b$ and $B1.c$ in C (note that the algorithm will also perform the same operations if $B1.bsize > k$). The maximum upper bound $C.maxUB$ is updated to $C.maxUB = B1.ubound$, $B1$ is inserted into Q, and the algorithm invokes $Q.Shrink()$. $B2$ and $B3$ will never be inserted into Q, because $C.maxUB < B2.lbound < B3.lbound$. Therefore, the maximum length of Q until this step was 1. Using the standard algorithm, $B2$ and $B3$ *will be* inserted into Q, even when they will never be removed from the queue (the algorithm will stop searching before removing them), thus

Algorithm 4.2: Our proposed k-NN search algorithm

```
   Input: $Index$, $q \in \mathbb{X}$, $k \in \mathbb{N}$
   Output: $k$-NN
1  $Q \leftarrow \emptyset$;
2  $C \leftarrow \emptyset$;
3  $C.maxUB \leftarrow \infty$;
4  $C.size \leftarrow 0$;
5  $B \leftarrow$ root of $Index$;
6  $B.c.distq \leftarrow d(q, B.c)$;
7  $B.lbound \leftarrow B.c.distq - B.cr$;
8  $Q.Add(B,\ B.lbound)$;
9  $C.Add2(B.c,\ B.c.distq,\ 1)$;
10 $C.Add2(B,\ B.c.distq + B.cr,\ B.bsize - 1)$;
11 while $Q \neq \emptyset$ do
12     $B \leftarrow Q.DequeueMin()$;
13     if $B_b \in C$ then
14         $ubound \leftarrow B_b.ubound$;
15         $C.size \leftarrow C.size - B_b.csize$;
16         $C.Remove(B_b)$;
17     else $ubound \leftarrow \infty$;
18     foreach $B' \in B.children$ do
19         $B'.c.distq \leftarrow d(q, B'.c)$;
20         $C.Add2(B'.c,\ B'.c.distq,\ 1)$;
21         if $B'.bsize > 1$ then
22             $B'.lbound \leftarrow B'.c.distq - B'.cr$;
23             if $B'.lbound < C.maxUB$ then $Q.Add(B',\ B'.lbound)$;
24             $minub \leftarrow \min\{ubound,\ B'.c.distq + B'.cr\}$;
25             $C.Add2(B',\ minub,\ B'.bsize - 1)$;
26         $Q.Shrink()$;
27 return $C$
```

Algorithm 4.3: New *add* algorithm for C

```
   Input: $B$, $ubound \in \mathbb{R}^+$, $csize \in \mathbb{N}$
1  if $ubound < C.maxUB$ then
2      $B_b \leftarrow CreateBubble(B)$;
3      $B_b.ubound \leftarrow ubound$;
4      $B_b.csize \leftarrow csize$;
5      $C.size \leftarrow C.size + B_b.csize$;
6      $C.Add(B_b,\ B_b.ubound)$;
7      while $C.size - C.Max().csize \geq k$ do
8          $C.size \leftarrow C.size - C.Max().csize$;
9          $C.DequeueMax()$;
10     if $C.size \geq k$ then $C.maxUB \leftarrow C.max().ubound$;
```

Algorithm 4.4: Shrink algorithm for Q

```
1  while $Q \neq \emptyset$ and $Q.Max().lbound \geq C.MaxUB$ do
2      $Q.DequeueMax()$;
```

wasting storage space.

Assume now that $B1$, $B2$, and $B3$ are inserted into Q (for example, this could be the case if the index contained several roots). The algorithm removes $B1$ from Q and processes each of its children. Then, it updates $C.maxUB$. Note that the algorithm ensures that $C.maxUB \leq B1.ubound$. Next, procedure $Q.Shrink()$ is invoked, which removes balls $B2$ and $B3$ from Q, thus diminishing the average length of Q during the search.

In both cases, the algorithm was able to filter out balls $B2$ and $B3$ without processing them and at a very early stage of the search. Therefore, the storage requirement for Q was successfully diminished using our proposed algorithm.

4.2.3 Cost analysis of the proposed algorithm

Now we compare the computational complexity of the original and our proposed k-NN search algorithms. Firstly, as both algorithms perform a best-first traversal of the index, it follows that they compute exactly the same number of distance computations for the same query object. Thus, for this concept the CPU cost is the same for both algorithms.

Regarding the insertion/deletions of elements in Q, the CPU cost of the original algorithm is $O(totQ \cdot \log \max(k, maxQ))$, where $totQ$ is the overall number of balls inserted into Q and $maxQ$ the maximum size of Q across

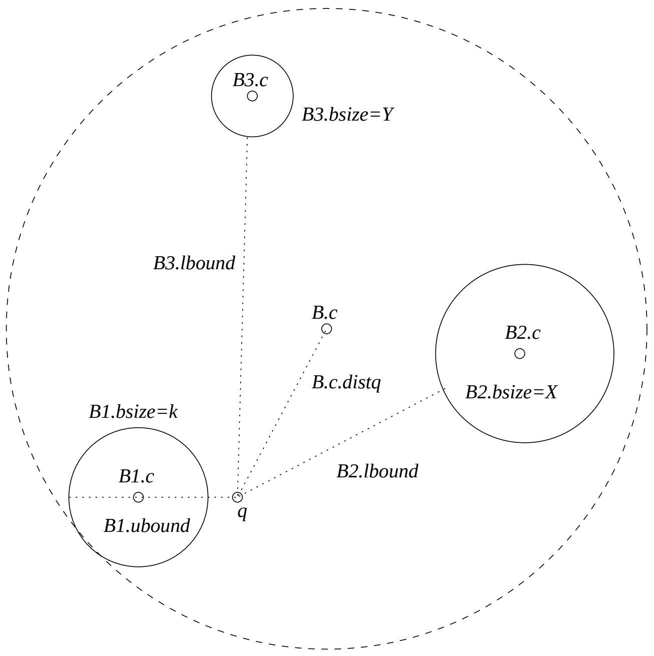

Figure 4.16: Example of a k-NN query

the process. The cost for our proposed algorithm is the same, noting that in our case $totQ$ and $maxQ$ will be smaller, given that we avoid some insertions into Q.

With respect to C, the CPU cost of the original algorithm is in the worst case $O(totQ \log(k))$, since all centers from balls in Q may be inserted into C, and it is ensured that only one object per iteration may be extracted from C. For our proposed algorithm, in the worst case it may be possible that $totQ$ objects and $totQ$ bubbles are inserted into C, but then the algorithm may extract up to k elements from C after an insertion (cf. Algorithm 4.3, lines 7–9 of the pseudocode). However, it is not possible to extract more elements that those inserted into C, therefore the total CPU cost in the worst case is also $O(totQ \log(k))$, i.e., it is the same CPU cost compared with the original algorithm. Note that the query "$B_b \in C$" in line 13 of Algorithm 4.2 requires a dictionary data structure built on top of C if one wants to avoid an $O(k)$ time linear traversal.

Thus, our proposed algorithm has the same CPU cost as the original one, but using always less memory for Q. As previously observed, our algorithm needs to know how many objects are within each ball of the index, which also uses some memory space (one float value per internal node). In most practical situations, there is always some extra space free on each index node, because it is almost impossible to completely use its assigned space (equal to the size of a disk data page), so the extra float can be stored "for free". Also, we experimentally observed that the memory savings are at least an order of magnitude higher than the extra used space. Although not an analytically interesting result, it is an interesting practical saving.

4.2.4 Experimental evaluation

For our experimental evaluation, we used several synthetic and real-world databases:

- *Gaussian*: This set of synthetic databases are formed by clusters in a vector space using different dimensionalities (8-D, 16-D, and 32-D), where the objects that form each cluster follow a Gaussian distribution. Each Gaussian database contains 1,000 clusters, and their centers are random points with coordinates uniformly distributed in $[0, 1]$. The variance for the Gaussian distribution was set equal to 0.001 for each coordinate, to produce compact clusters. The size of each cluster is similar but not necessarily equal, and the whole dataset contains 100,000 objects. We generated 1,000 random query points, which follow the same data distribution as the database.

- *Corel Features*: The *Corel features database* contains features from 68,040 images extracted from a Corel image collection. The features are based on the color histogram (32-D), color histogram layout (32-D), co-occurrence texture (16-D), and color moments (9-D). This database is available at the *UCI KDD Archive* [Hettich and Bay, 1999]. We used a subset of this database consisting on 65,615 images, because there were some missing features for some of the images (we included only those objects for which complete sets of feature vectors were available). We selected 1,000 objects at random from the database to be used as query objects. For the experiments presented in this section, we used the color histogram (*CH*) and the layout histogram (*LH*) databases.

- *Edge structure*: This database contains 20,197 feature vectors (edge structure, 18-D) extracted from the Corel image database. We selected 1,000 random objects from the database as query points.

The *number of objects per cluster* was selected depending on the dimensionality of the dataset, in a way that all objects from the cluster (plus a small header) could fit on a datapage. By setting the datapage size to 4 Kb, we obtained the following cluster size values: 127 (8-D), 63 (16-D), 56 (18-D), and 31 (32-D).

We compared the standard best-first k-NN algorithm (labeled *HS*) against ours (labeled *Ours*) and the best-first version proposed by Samet [2003] (labeled *Samet*)[3]. As representative index structures, we used the *List Of Clusters* [Chávez and Navarro, 2005] and the *M-tree* [Ciaccia et al., 1997]. The List of Clusters can be seen as a "list of balls", i.e., a search hierarchy with only one level, while the M-tree is a more general hierarchical index structure. To compare the storage requirements of each search algorithm, we computed the mean of the maximum queue length ($\max\{|Q|\}$) obtained on each query, and the average length of Q. The first measure indicates how much memory (on average) the search algorithm needs in order to answer the k-NN query. The second measure is related to the number of disk accesses made if the queue is stored on secondary memory. All results are shown as percentage of the database size.

Figures 4.17 to 4.22 show the results obtained with the Gaussian databases. Our algorithm needs considerably less memory than the standard algorithm, especially for high dimensions. For example, our algorithm only used 18% of the memory required by the standard algorithm with 32-D and using List of Clusters ($k = 50$). In low dimensions, the gain was smaller (48% of memory requirement compared with the standard algorithm), but still considerable. The List of Clusters performed better than the M-tree in terms of storage usage. In both indices, our algorithm was consistently better than the original version by Samet. The average length of the queue was up to 5 times shorter than the standard algorithm. The charts also show that the storage efficiency degrades with k, especially in the case of the M-tree.

We obtained similar results with real-world datasets (see Figures 4.23 to 4.28). For example, with the color histogram database, our algorithm only used 34% of the storage requirement of the standard algorithm with List of Clusters. The average length queue was always smaller than 32% compared with the standard algorithm. Similar improvements were obtained with the other two real-world databases.

Table 4.2 summarizes the improvements in storage requirements of our algorithm over the standard k-NN search for $k = 50$. Note that the better results were obtained with the List of Clusters. A possible explanation for this is that this index structure produces more compact balls than the M-tree (due to the dynamic nature of the last index structure), thus the search algorithm is able to find better distance estimations with the List of Clusters.

[3]Actually, Samet speaks mostly of depth-first algorithms, but the paper mentions that the best-first algorithm could be handled as well. As we are interested in the optimal traversal order, we compare only its best-first version.

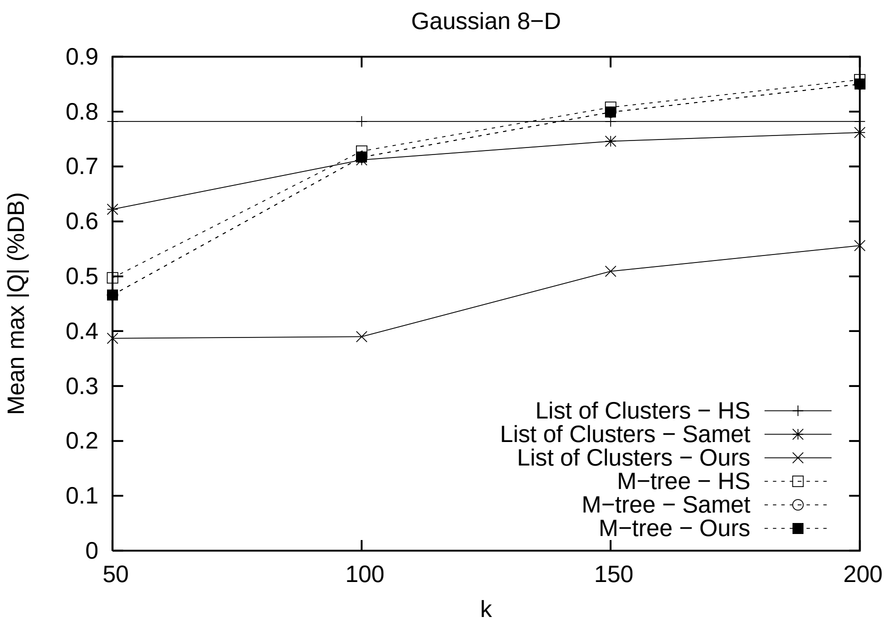

Figure 4.17: Gaussian 8-D: Mean of the maximum queue length

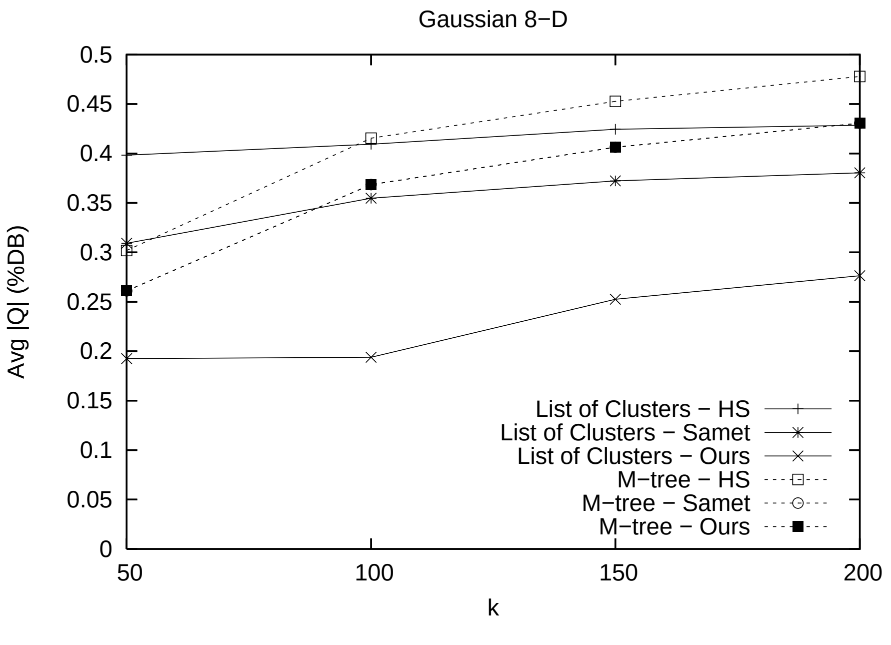

Figure 4.18: Gaussian 8-D: Average queue length

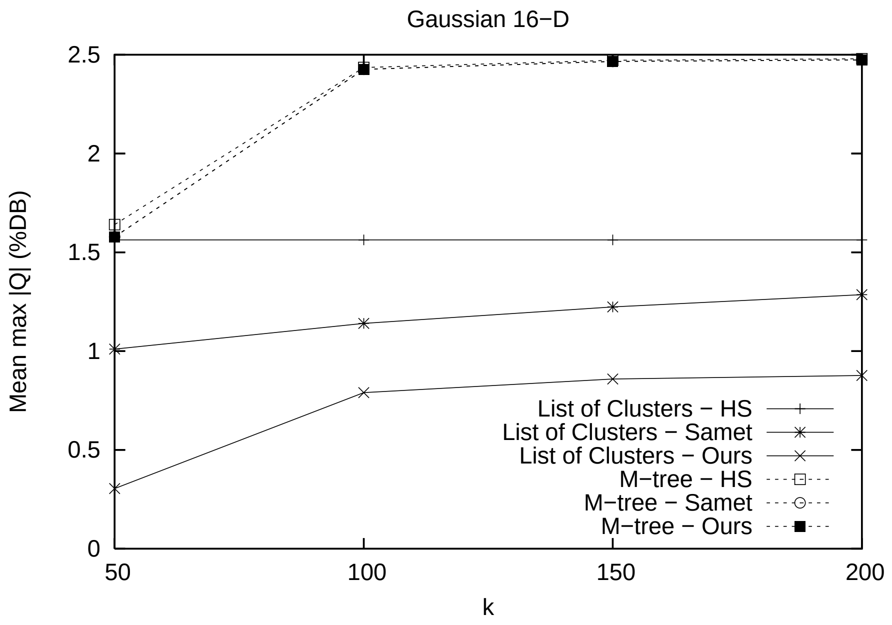

Figure 4.19: Gaussian 16-D: Mean of the maximum queue length

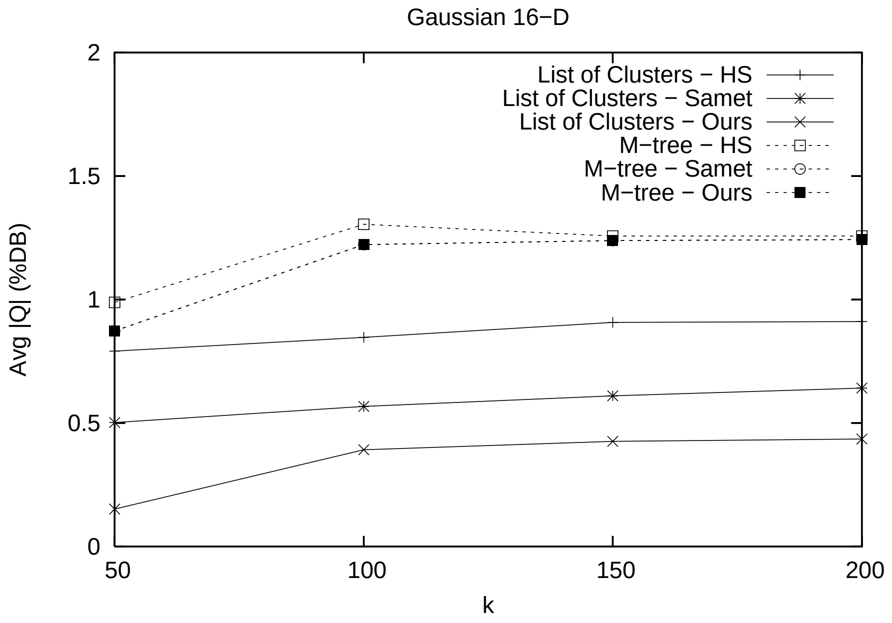

Figure 4.20: Gaussian 16-D: Average queue length

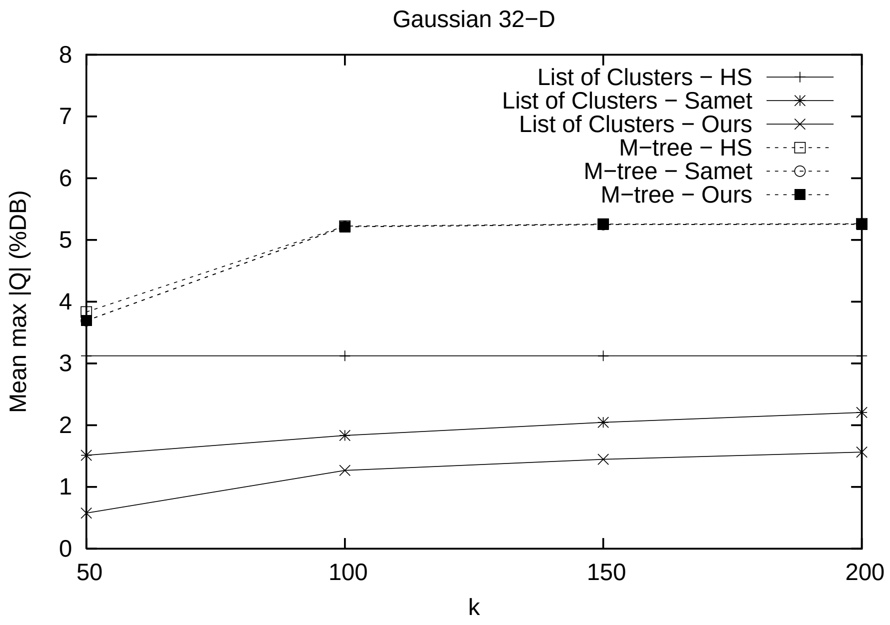

Figure 4.21: Gaussian 32-D: Mean of the maximum queue length

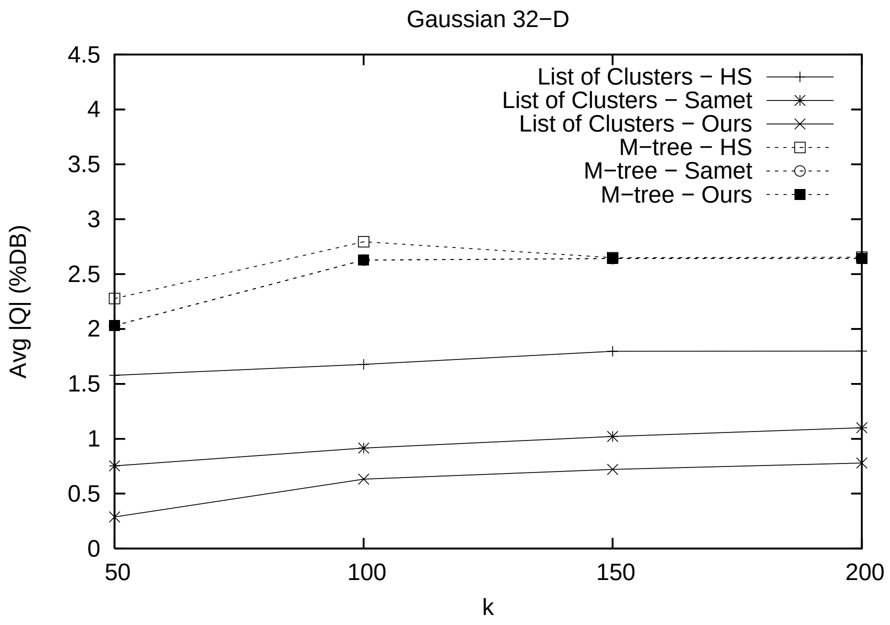

Figure 4.22: Gaussian 32-D: Average queue length

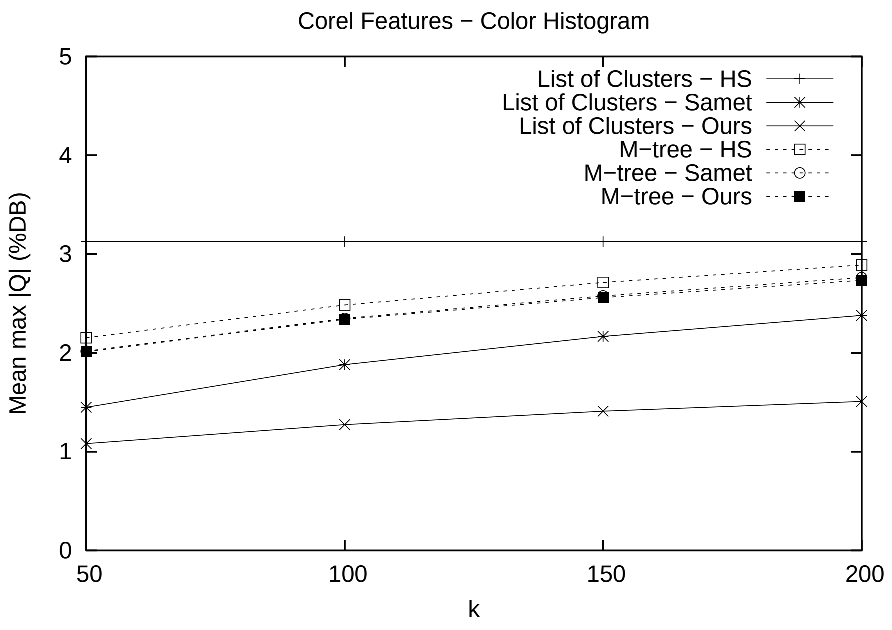

Figure 4.23: Corel features CH: Mean of the maximum queue length

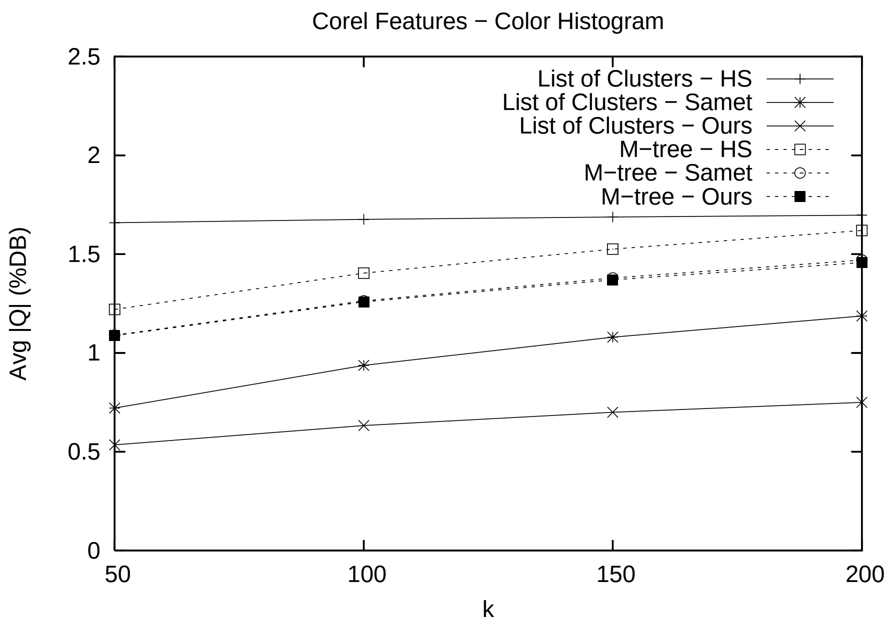

Figure 4.24: Corel features CH: Average queue length

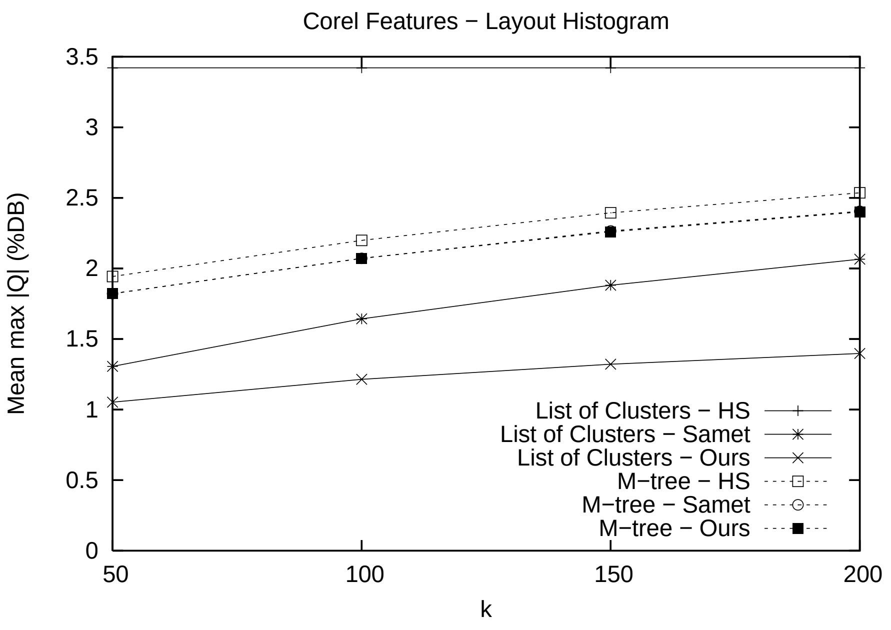

Figure 4.25: Corel features LH: Mean of the maximum queue length

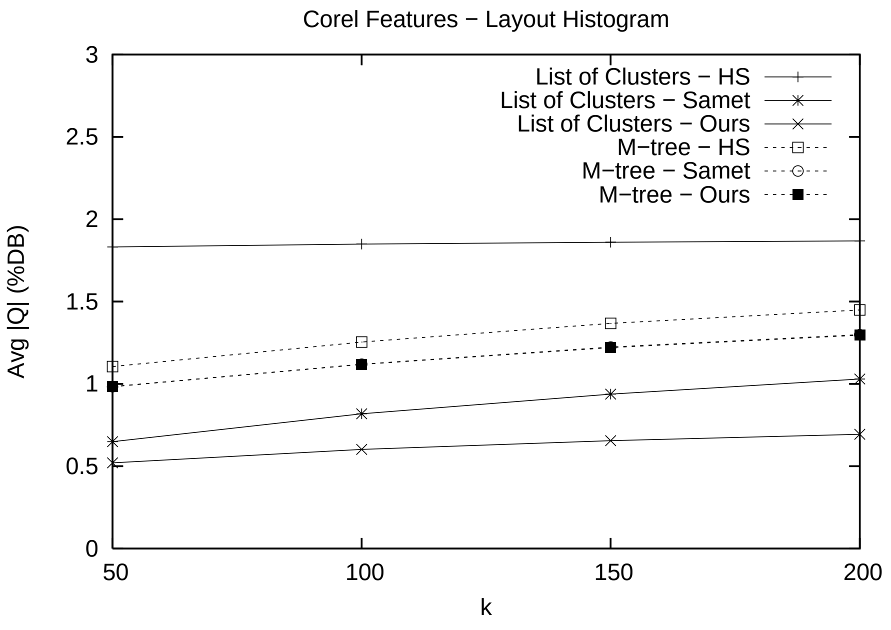

Figure 4.26: Corel features LH: Average queue length

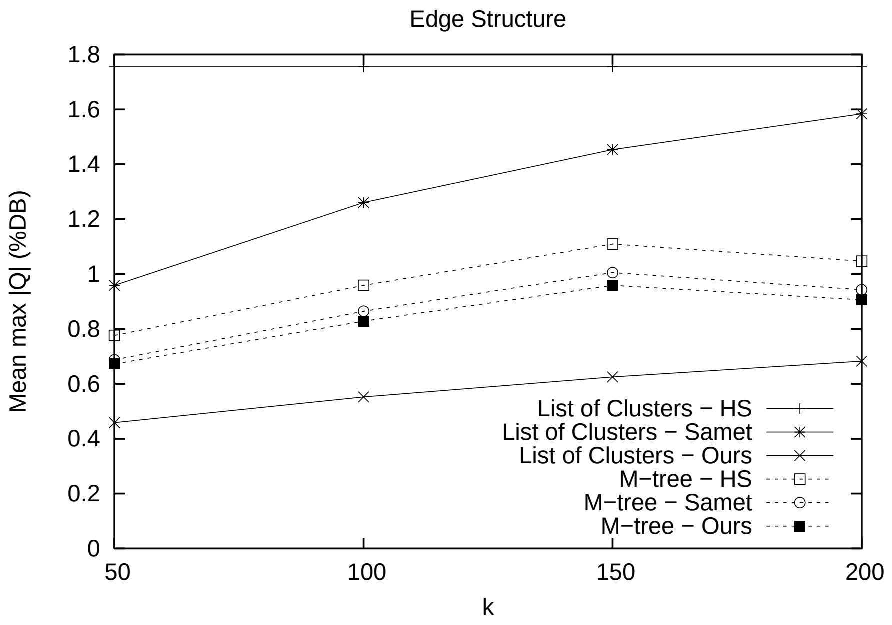

Figure 4.27: Edge structure: Mean of the maximum queue length

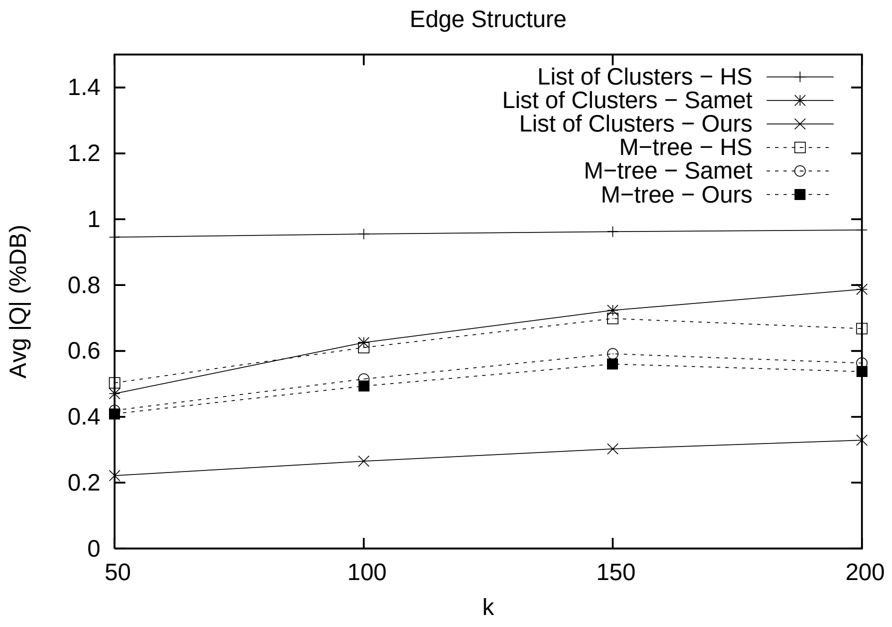

Figure 4.28: Edge structure: Average queue length

Database	LOC-max	LOC-avg	MT-max	MT-avg
Gaussian 8-D	49.4%	48.3%	93.7%	86.5%
Gaussian 16-D	19.5%	19.2%	96.2%	88.3%
Gaussian 32-D	18.5%	18.2%	96.3%	89.1%
Color Histogram	34.6%	32.3%	93.4%	89.1%
Layout Histogram	30.8%	28.4%	93.7%	88.9%
Edge Structure	26.1%	23.4%	86.8%	81.3%

Table 4.2: Storage requirements (maximum and average queue length) of our algorithm (standard algorithm: 100%, $k = 50$)

4.3 Pivot-based index for combinations of feature vectors

In this section, we present a novel index structure that provides efficient nearest-neighbor queries in multimedia databases that consist of objects described by multiple feature vectors. The benefits of the simultaneous usage of several (statically or dynamically) weighted feature vectors with respect to retrieval *effectiveness* have been previously studied in this thesis (see Section 3.2). Support for *efficient* multi-feature vector similarity queries is an open problem, as existing indexing methods do not support dynamically parameterized distance functions. We present a solution for this problem that relies on a combination of several pivot-based metric indices. We define the index structure, present an algorithm for performing nearest-neighbor queries on the new index, and demonstrate the feasibility of our technique by an experimental evaluation conducted on two real-world image databases.

To describe multimedia objects using the feature vector approach, numerical values are extracted from each object to form feature vectors of typically high dimensionality. For many multimedia data types (e.g., images, 3D models, audio tracks), a number of extraction algorithms have already been proposed. For example, in the case of 3D model retrieval, there are more than 30 proposed feature vectors [Bustos et al., 2005a].

We already showed in Section 3.2 that combinations of feature vectors may lead to significant improvements in the effectiveness of the similarity search. Now we want to address the efficiency problem, that is, how to efficiently perform similarity queries with dynamically weighted combinations of feature vectors.

Traditional index structures were primarily designed to index single feature vectors, and they cannot be directly used to index a set of dynamically combined features. Even if one concatenates the feature vectors and applies standard indexing techniques, the efficiency of these indices may be poor due to the curse of dimensionality. To solve this problem, we propose a pivot-based index structure that can be used to improve the efficiency of similarity search algorithms in multimedia databases, where each multimedia object is described by a set of different feature vectors.

This work has been published in Bustos et al. [2005b].

4.3.1 Index description

The proposed data structure aims to index dynamically weighted combinations of feature vectors. The combined distance function that compares two multimedia objects has the form

$$\delta_{dc}(x,y) = \sum_{i=1}^{F} w_i \cdot \frac{\delta_i(x,y)}{norm_i},$$

where the weights $w_i \in \mathbb{R}^+$ are dynamically assigned on each similarity query. The weights may be computed, for example, using the purity method or the entropy impurity method described in Section 3.3.

Theorem 1. *If $\forall i$, δ_i is a metric, the function δ_{dc} is also a metric.*

Proof. We prove this theorem using the principle of induction on F. Let x, y, and z be three multimedia objects. As the weights are positive real values, the strict positiveness property is direct (we do not consider the special case when $\forall i\ w_i = 0$, because the resulting combined distance function is not interesting for similarity search). Symmetry is also direct, because all δ_i distances are symmetric:

$$\delta_{dc}(x,y) = \sum_{i=1}^{F} w_i \cdot \frac{\delta_i(x,y)}{norm_i} = \sum_{i=1}^{F} w_i \cdot \frac{\delta_i(y,x)}{norm_i} = \delta_{dc}(y,x).$$

The rest of the proof concentrates on proving the triangle inequality.

Basis case: The case $F = 1$ is trivial. For $F = 2$, we observe that

$$w_1 \cdot \frac{\delta_1(x,z)}{norm_1} \leq w_1 \cdot \frac{\delta_1(x,y)}{norm_1} + w_1 \cdot \frac{\delta_1(y,z)}{norm_1}, \text{ and}$$

$$w_2 \cdot \frac{\delta_2(x,z)}{norm_2} \leq w_2 \cdot \frac{\delta_2(x,y)}{norm_2} + w_2 \cdot \frac{\delta_2(y,z)}{norm_2},$$

which are true because both δ_1 and δ_2 are metrics. Summing both inequations we get

$$\sum_{i=1}^{2} w_i \cdot \frac{\delta_i(x,z)}{norm_i} \leq \sum_{i=1}^{2} w_i \cdot \frac{\delta_i(x,y)}{norm_i} + \sum_{i=1}^{2} w_i \cdot \frac{\delta_i(y,z)}{norm_i},$$

which concludes the proof of the basis case.

Induction step ($F-1 \Rightarrow F$): We have that

$$\sum_{i=1}^{F-1} w_i \cdot \frac{\delta_i(x,z)}{norm_i} \leq \sum_{i=1}^{F-1} w_i \cdot \frac{\delta_i(x,y)}{norm_i} + \sum_{i=1}^{F-1} w_i \cdot \frac{\delta_i(y,z)}{norm_i}, \text{ and}$$

$$w_F \cdot \frac{\delta_F(x,z)}{norm_F} \leq w_F \cdot \frac{\delta_F(x,y)}{norm_F} + w_F \cdot \frac{\delta_F(y,z)}{norm_F}.$$

We assume that the first inequation is true (induction hypothesis). The second inequation is true because δ_F is a metric. Summing both inequations we obtain

$$\sum_{i=1}^{F} w_i \cdot \frac{\delta_i(x,z)}{norm_i} \leq \sum_{i=1}^{F} w_i \cdot \frac{\delta_i(x,y)}{norm_i} + \sum_{i=1}^{F} w_i \cdot \frac{\delta_i(y,z)}{norm_i},$$

which concludes the proof. $\square$

The canonical pivot-based algorithm stores in a table all the distances between objects $u \in \mathbb{U}$ and selected pivots $p \in \mathbb{P}$, with $\mathbb{P} \subset \mathbb{U}$. In the case of dynamically weighted combinations of feature vectors, what we would like to compute is a matrix of the form

$$M = \begin{bmatrix} \delta_{dc}(p_1,u_1) & \dots & \delta_{dc}(p_k,u_1) \\ \vdots & \ddots & \vdots \\ \delta_{dc}(p_1,u_n) & \dots & \delta_{dc}(p_k,u_n) \end{bmatrix}.$$

Such a matrix could be directly used to implement a standard pivot-based index. However, the combined distance function δ_{dc} is not static, because the weights depend on the query object. Therefore, it is not possible to precompute the matrix M with the combined distances between pivots and objects, because we do not know a priori the set of weights and they may change for each query.

To overcome this problem, we propose a novel index structure that builds the distance matrix at query time. The index consists of N matrices of the form

$$M_i = \frac{1}{norm_i} \cdot \begin{bmatrix} \delta_i(p_1,u_1) & \dots & \delta_i(p_k,u_1) \\ \vdots & \ddots & \vdots \\ \delta_i(p_1,u_n) & \dots & \delta_i(p_k,u_n) \end{bmatrix}, 1 \leq i \leq F.$$

With this information, it is possible to compute matrix M once the weights corresponding to a similarity query are known. It follows that

$$M = \begin{bmatrix} \sum_{i=1}^{F} w_i \cdot M_i[1,1] & \dots & \sum_{i=1}^{F} w_i \cdot M_i[k,1] \\ \vdots & \ddots & \vdots \\ \sum_{i=1}^{F} w_i \cdot M_i[1,n] & \dots & \sum_{i=1}^{F} w_i \cdot M_i[k,n] \end{bmatrix}.$$

Intuitively, at query time we dynamically build the pivot index table for the submitted combination of weights. This table is then used by the similarity search engine to discard objects during the search, thus avoiding a sequential scan of the database. Note that the proposed index structure is not restricted to the case of vector spaces. It can also index metric spaces where the distance function is formed by a weighted combinations of metrics.

Nearest-neighbor search algorithm

We use a modification of the NN search algorithm sketched in Chávez et al. [2001b] to perform this type of query using our proposed index. This algorithm can be easily modified to implement k-NN queries.

The idea of the NN search algorithm is as follows. Given a set of pivots $\mathbb{P} \subset \mathbb{U}$, we compute the distances between all pivots and the query object q, and the pivot whose distance to q is minimum (*mindist*) will be the first *NN candidate*. Then, for each object $u \in \mathbb{U}$ that is not a pivot, the exclusion criterion is applied, using as tolerance radius the distance from the candidate NN to the query object. If u cannot be discarded, we compute the distance between u and q. If this distance is smaller than *mindist*, we set u as the new NN candidate and update *mindist*. The process ends when all the objects from $\mathbb{U}$ have been checked. Algorithm 4.5 shows the pseudocode for this algorithm.

Another version of the NN algorithm that is optimal in the number of distance computations, but that needs $O(n)$ extra space, is as follows. We compute the distances between all pivots and the query object q, and the pivot whose distance to q is minimum (*mindist*) will be the first *NN candidate*. Then, we compute the lower bound distance from q to each object $u \in \mathbb{U}$ that is not a pivot. This lower bound is computed as $\delta_{lowerbound}(q, u) = \max_{i=1}^{|\mathbb{P}|} |\delta(p_i, u) - \delta(p_i, q)|$. Then, we sort the objects $u \in \mathbb{U}$ in ascending order according to their lower bound distances to q. Starting with the the object u with smallest lower bound distance, we apply the pivot exclusion criterion using as tolerance radius the distance from the candidate NN to the query object. If u cannot be discarded, we compute the distance between u and q. If this distance is smaller than *mindist*, we set u as the new NN

Algorithm 4.5: NN search algorithm (radius reduction)

```
   Input: $\mathbb{U}$, $\mathbb{P}$, $Index$, $q \in \mathbb{X}$, $w = \{w_i\}$
   Output: NN
1  foreach $u_i \in \mathbb{U}$ do
2      foreach $p_j \in \mathbb{P}$ do
3          $M[i,j] \leftarrow \sum_{k=1}^{F} w_k \cdot M_k[i,j]$;
4  $mindist \leftarrow \min_{i=1}^{k}\{\delta_{dc}(p_i, q)\}$;
5  $NN \leftarrow p_{\arg\min_{i=1}^{k}\{\delta_{dc}(p_i,q)\}}$;
6  foreach $u_i \in \mathbb{U} - \mathbb{P}$ do
7      foreach $p_j \in \mathbb{P}$ do
           // Note that $\delta_{dc}(p,q)$ is already computed
8          if $|M[i,j] - \delta_{dc}(p,q)| > mindist$ then
9              Discard $u_i$;
10             break;
11     if $u_i$ not discarded $\wedge$ $\delta_{dc}(q,u_i) < mindist$ then
12         $mindist \leftarrow \delta_{dc}(q,u)$;
13         $NN \leftarrow u$;
14 return $NN$
```

candidate and update *mindist*. The process ends when all the objects from $\mathbb{U}$ have been checked or if the lower bound distance of an object is greater than *mindist* (i.e., no other object can be closer to q than the actual NN candidate). Algorithm 4.6 shows the pseudocode for this algorithm.

4.3.2 Experimental evaluation

We performed a number of NN queries using two real-world databases, and computed the average response time. We used the Manhattan distance as the distance function for all feature vectors. For constructing the pivot-based indices, we used random as well as good pivots (cf. Section 4.1). All feature vectors were normalized by the estimated maximum distance between two points in the space (for each feature vector, respectively).

We implemented both proposed NN search algorithm and compared it against a linear scan. The platform on which the experiments were run has two Intel Xeon (2.80 GHz) CPUs and 4 GB of main memory. As efficiency measures, we used the CPU time, the number of discarded objects by the index, and the number of distance computations performed in average to

Algorithm 4.6: NN search algorithm (sorting by lower bound)

```
   Input: $\mathbb{U}$, $\mathbb{P}$, $Index$, $q \in \mathbb{X}$, $w = \{w_i\}$
   Output: NN
 1 foreach $u_i \in \mathbb{U}$ do
 2     foreach $p_j \in \mathbb{P}$ do
 3         $M[i,j] \leftarrow \sum_{k=1}^{F} w_k \cdot M_k[i,j]$;
 4 $mindist \leftarrow \min_{i=1}^{k}\{\delta_{dc}(p_i, q)\}$;
 5 $NN \leftarrow p_{\arg\min_{i=1}^{k}\{\delta_{dc}(p_i,q)\}}$;
   // Computing lower bound distances
 6 foreach $u_i \in \mathbb{U} - \mathbb{P}$ do
 7     $\delta_{lowerbound}(q, u) \leftarrow 0$;
 8     foreach $p_j \in \mathbb{P}$ do
           // Note that $\delta_{dc}(p, q)$ is already computed
 9         if $|M[i,j] - \delta_{dc}(p,q)| > \delta_{lowerbound}(q, u_i)$ then
10             $\delta_{lowerbound}(q, u_i) \leftarrow |M[i,j] - \delta_{dc}(p,q)|$;
   // Sorting objects in $\mathbb{U} - \mathbb{P}$ by ascending lower bound
11 $\mathbb{U}' \leftarrow Sort(\mathbb{U} - \mathbb{P})$;
   // Searching for NN
12 foreach $u_i \in \mathbb{U}'$ do
13     if $\delta_{lowerbound}(q, u_i) > mindist$ then break;
14     if $\delta_{dc}(q, u_i) < mindist$ then
15         $mindist \leftarrow \delta_{dc}(q, u)$;
16         $NN \leftarrow u$;
17 return $NN$
```

answer the NN queries. We performed a number of NN queries using two real-world databases, and computed the average response time. We used the Manhattan distance as the distance function for all feature vectors.

For the experimental evaluation, we used two real datasets:

- *Corel features* (described in Section 4.2.4).
- *Corel images databases:* Contains several features obtained from images of a subset of the Corel Gallery 380,000 package. The database contains 6,192 images classified into 63 categories. Six features vectors of very high dimensionality (184-D, 165-D, 784-D, 625-D, 784-D, and 30-D) were computed for each image. The feature vectors include color histogram, texture, and convolution descriptors (see Pickering

and Rüger [2003] and Howarth and Rüger [2004] for details on the feature vectors). We used 10% of the objects from this database, selected at random, as query objects.

All feature vectors were normalized by the estimated maximum distance between two points in the respective feature space. For each query object, the weights were generated at random (a value uniformly distributed in the range [0, 1]).

Experimental results

Figures 4.29 and 4.30 show the number of discarded objects (as a percentage of the respective database) as a function of the number of pivots used to build the index. The figure shows the results using random and good pivots. We consistently obtained better results using good sets of pivots with both datasets. With a few pivots, the proposed index structure was able to discard large fractions of the databases. Even with the Corel images database, which has a combined dimensionality of more than 2,500-D, the index was able to discard more than 50% of the database objects with only 10 pivots. Note that the algorithm based on sorting the objects from $\mathbb{U}$ by their lower bound distances discarded more objects, using the same number of pivots, as the algorithm based on radius reduction.

Figures 4.31 and 4.32 show the number of distance computations (as a fraction of the database size) as a function of the number of pivots used, using random and good sets of pivots. For the Corel features database, the optimum performance was reached with 16 good pivots, and for the Corel images database the optimum was reached with 220 good pivots (both with the NN algorithm based on sorting by lower bound distance).

Figures 4.33 and 4.34 show the average time (in milliseconds) needed to answer NN queries using the proposed index structure, using random and good sets of pivots. It follows that the total time is highly correlated with the number of distance computations needed to answer the NN queries (cf. Figures 4.31 and 4.32), that is, most of the CPU time used by the NN search algorithm is spent on computing distances. Also notice that the algorithm based on sorting by lower bound distance was slower than the algorithm based on radius reduction with the Corel features database. This is because the distance function is not too expensive, and therefore the time using for sorting the distances becomes also important. With the Corel images database, the distance function is more expensive and the time needed by both algorithms are similar (but still the algorithm based on radius reduction was faster for more than 100 pivots).

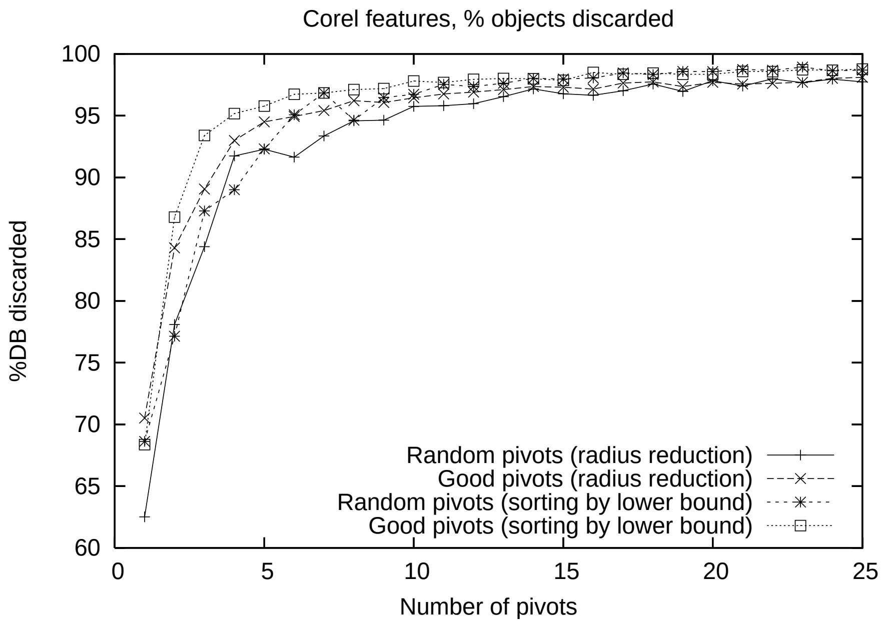

Figure 4.29: Corel features: Objects discarded as a function of the number of pivots

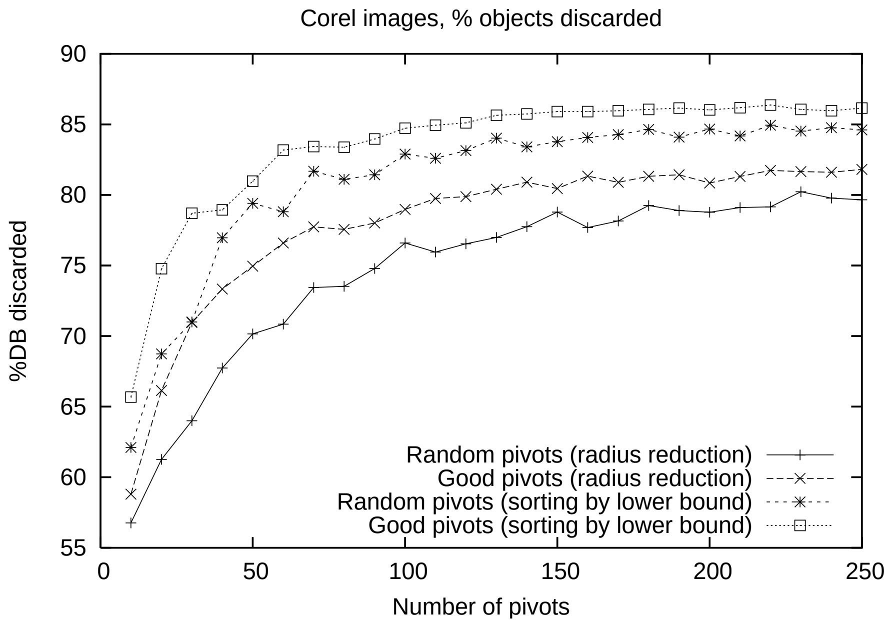

Figure 4.30: Corel images: Objects discarded as a function of the number of pivots

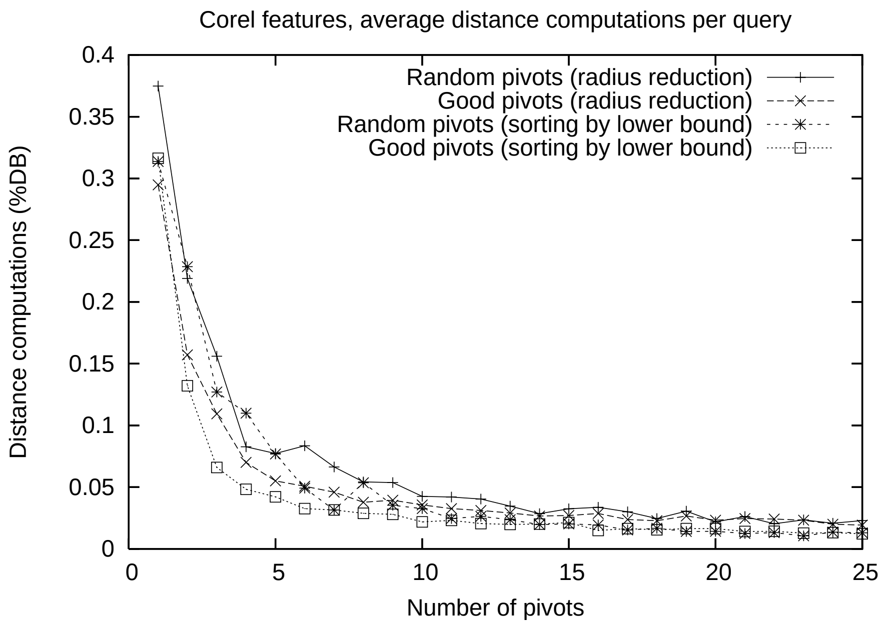

Figure 4.31: Corel features: Average number of distance computations per NN query as a function of the number of pivots

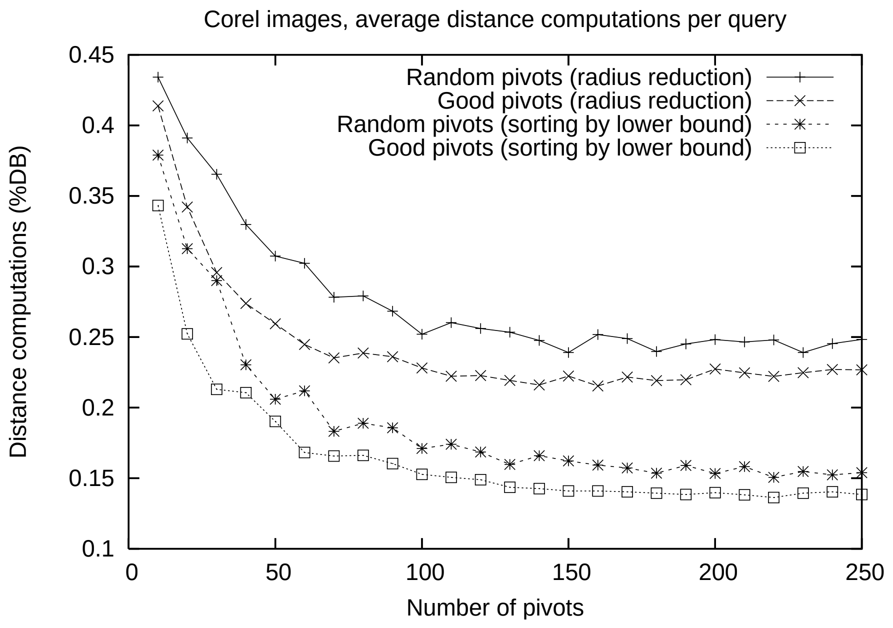

Figure 4.32: Corel images: Average number of distance computations per NN query as a function of the number of pivots

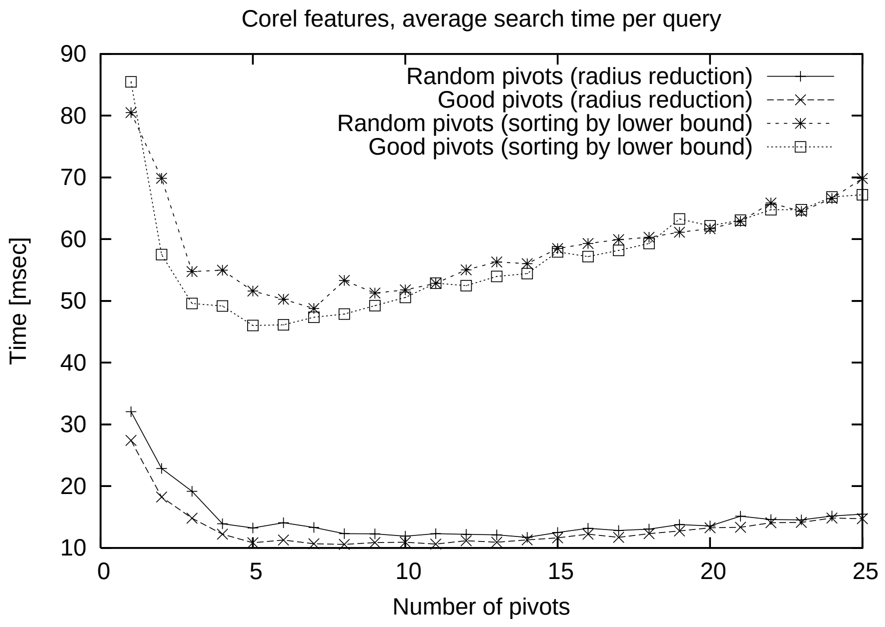

Figure 4.33: Corel features: Average time per NN query as a function of the number of pivots

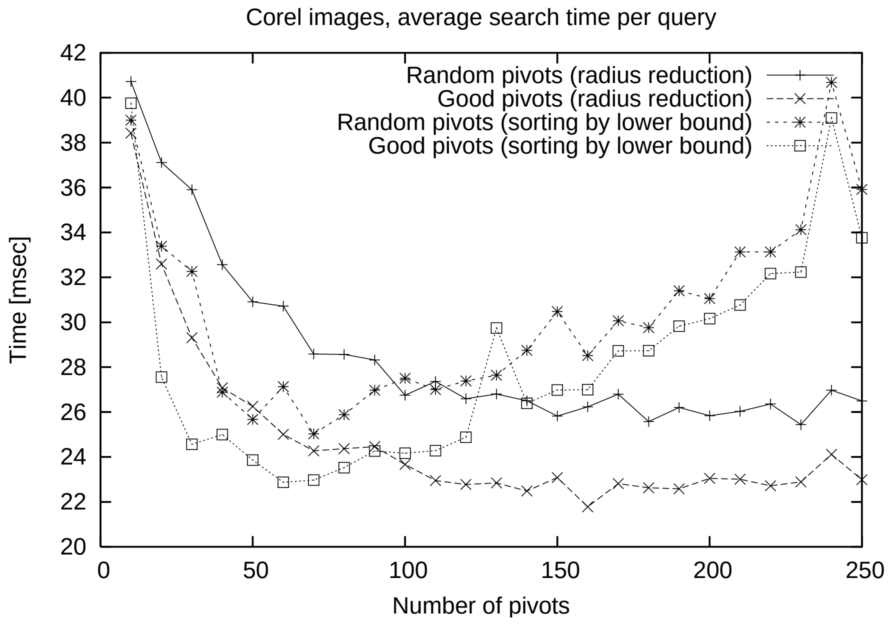

Figure 4.34: Corel images: Average time per NN query as a function of the number of pivots

Finally, Table 4.3 presents a summary of the improvements obtained by the proposed index structure compared with a sequential scan (naïve search method) of the database.

Database	Method	# pivots	Avg. time (msec)	Improvement
Corel features	Linear scan	-	57.22	-
	Random pivots	14	11.70	4.89x
	Good pivots	8	10.57	5.41x
Corel images	Linear scan	-	88.90	-
	Random pivots	230	25.44	3.49x
	Good pivots	160	21.78	4.08x

Table 4.3: Improvements obtained with the Corel features and the Corel images databases

4.4 Indexing frequently used combinations of feature vectors

Although many feature extraction functions are available for each multimedia domain, only one feature extraction function is usually used within a multimedia database for performing similarity queries. We have shown in Section 3.2 that by using *query dependent combinations of feature vectors* it is possible to significantly increase the effectiveness of the similarity search. The idea is to select some of the available feature vectors (depending on the query object and according to some predefined criteria), and to linearly combine them to perform the similarity query. Figure 4.35 illustrates this new approach for implementing effective similarity queries.

As explained in Section 3.3, we have observed that a linear combination of all extracted FVs will not provide the optimal results, because if one of the considered FVs has a very low effectiveness for a given query object, it will degrade the final result. By using dynamic combination methods (see Sections 3.3.1 and 3.3.2), we can avoid this problem by combining only those FVs that are most promising for the given query object.

We address the problem of *indexing combinations of selected feature vectors*. In real multimedia datasets, the similarity search system usually selects only a few of the possible combinations with high probability. Our solution aims to use the available space (e.g., in secondary storage) to build indices for those frequently used combinations, i.e., we propose to construct not only one index but a *set of indices* that optimally uses the available space. We model the problem as a binary linear program, which is able to find the set of

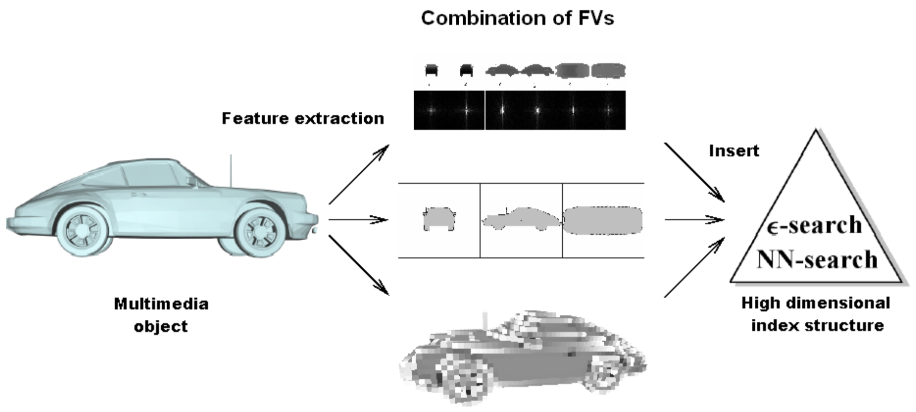

Figure 4.35: The new approach for similarity search in multimedia databases using combinations of feature vectors

indices that minimizes the expected search cost. Our model is general, in the sense that it can be used with any index structure, and it allows us to find the optimal set of indices. Unfortunately, it is not an efficient method (binary linear programming is NPO-Hard in the general case [Garey and Johnson, 1979; Hromkovic, 2001]). Therefore, we also propose fast algorithms that can find a good set of indices.

In this section, we assume that the similarity search engine of the multimedia database implements a *query processor*, which for a given query object selects t FVs (from the set of available FVs) to perform the similarity query.

This work has been submitted for publication [Bustos et al., 2006b].

4.4.1 Indexing combinations of feature vectors

Here we present the formal definition of the optimization problem to be solved. The idea is to build a set of indices that minimizes the expected search cost of similarity queries based on combinations of FVs. Table 4.4 shows the notation used in this section.

Assumptions

Let $\mathbb{X}$ be the universe of valid multimedia objects (which depends on the application domain), and let $\mathbb{U} \subset \mathbb{X}$ be the set of objects or database ($|\mathbb{U}| = n$). Let $\mathbb{F} = \{f\}$ be a set of FVs, each of them with dimensionality d.[4] A *combination of FVs* has the form $c \subseteq \mathbb{F}$. To perform similarity queries, the

[4]We made this assumption to consider all FVs equally important for the combination. Later on this section, we will discuss what happens if one relaxes this restriction.

Symbol	Description
$\mathbb{X}$	Set of valid objects
$\mathbb{U} \subset \mathbb{X}$	Database
$n = \|\mathbb{U}\|$	Database size
$q \in \mathbb{X}$	Query object
$\mathbb{F}$	Set of feature vectors (FVs)
$F = \|\mathbb{F}\|$	Number of FVs
$f \in \mathbb{F}$	A single FV
d	Dimensionality of the FVs
$c \subseteq \mathbb{F}$	A combination of FVs
t	Number of combined FVs
$T = \binom{F}{t}$	Total number of combinations
$\mathbb{C}$	Set of all combinations of t FVs
p_c	Probability of selecting combination c
$\mathbb{I}$	Set of indices (*iSet*)
$idx \in \mathbb{I}$	An index from the *iSet*
S	Available space for building indices

Table 4.4: Notation used in Section 4.4

search system combines t of the FVs, that is, it selects a combination c such that $|c| = t$. It follows that there are $T = \binom{F}{t}$ different combinations of FVs.

A query processor selects at query time one of the combinations to perform the similarity search. That is, given an object $q \in \mathbb{X}$, the query processor selects combination c_i ($1 \leq i \leq T$) with probability p_{c_i}, where $p_{c_i} \geq 0$ and $\sum_{i=1}^{T} p_{c_i} = 1$. Without loss of generality, in what follows we assume that $p_{c_i} \geq p_{c_{i+1}}$.

For example, the similarity query can be solved by linearly searching each of the FVs of the selected combination. The search cost of this linear scan is given by the function $LS(t, d, n)$, which is $O(tdn)$. Note that this linear scan *cannot be efficiently implemented*, because it is not known a priori which FVs will be selected for the combination for a given query q (we only know the probability p_{c_i} of selecting combination c_i). Thus, the FVs files cannot be optimally arranged on the secondary storage for the linear scan: If we had F files storing each FV independently, we would need to read t different files (one at the time) to compute the combined distance between the query and the objects from the database. Therefore, after reading each file we would need to store n partial distances (in main memory or in disk). The extra $O(n)$ space cost could be too expensive if the database is too large. Now, if we read only one disk page of each file at the time to avoid the $O(n)$ space cost (computing directly the combined distance of all objects stored on that disk page), we would not be performing a sequential scan on the disk (i.e., to read sequential disk pages), thus this could not be optimally performed.

If there is enough available space to construct and save indices, this space could be used to improve the efficiency of the search. The idea is to build indices for the most frequently used combinations of FVs, thus reducing the expected search cost. We define an *index of size* k as an index that stores combinations of k FVs (e.g., we could use the pivot-based index described in Section 4.3). The space cost of an index of size k is given by the function $\mathit{Space}(k, d, n)$, and its search cost is given by the function $\mathit{Search}(k, d, n)$. (As the values of d and n are assumed to be fixed for a given database, we will only write $\mathit{Space}(k)$, $\mathit{Search}(k)$, and $LS(t)$.) Both space and search cost functions depend on the currently used index structure and on the data distribution for each FV. For simplicity, we assume that all indices of size k use the same index structure and that they have the same search and space cost (independent of the indexed FVs). We also assume that the search cost always increases with the index size.

Indexing single combinations

We will first assume that we are only allowed to build indices of size t, which can index one combination of FVs. Let $\mathbb{I} = \{\mathit{idx}\}$ be the set of constructed indices. We refer to $\mathbb{I}$ as an *iSet* (*index set*). Given that there is an amount S of available space to build indices, it follows that $|\mathbb{I}| \leq \lfloor S/\mathit{Space}(t) \rfloor$. The total space cost $R(\mathbb{I})$ is equal to

$$R(\mathbb{I}) = |\mathbb{I}| \cdot \mathit{Space}(t).$$

The expected search cost depends on which combinations of FVs are indexed. If the query processor selects a combination c to perform the similarity query, the search engine checks if there is an index $\mathit{idx} \in \mathbb{I}$ that contains c. If this is the case, idx is used to perform the similarity query. Otherwise, the search engine resorts to a linear scan over the database. Without losing generality, let us suppose that the combinations of FVs are enumerated, and that the first $|\mathbb{I}|$ combinations in order of decreasing p_{c_i} were indexed. Then, the expected search cost $E(\mathbb{I})$ of the *iSet* is

$$E(\mathbb{I}) = \sum_{i=1}^{|\mathbb{I}|} p_{c_i} \cdot \mathit{Search}(t) + \left(1 - \sum_{i=1}^{|\mathbb{I}|} p_{c_i}\right) \cdot LS(t).$$

As the probabilities are ordered in descending order, it follows that the expected search cost is minimum when $|\mathbb{I}| = \lfloor S/\mathit{Space}(t) \rfloor$ (assuming that $\mathit{Search}(t) < LS(t)$, otherwise it would be always convenient to search using a linear scan). If $T \cdot \mathit{Space}(t) \leq S$, then the optimal solution is to build an index for every possible combination of FVs.

Indexing several combinations per index

If the indices of the *iSet* may contain more than t FVs, then one can do better. An index of size k ($t \leq k \leq F$) contains $\binom{k}{t}$ combinations of FVs. This means that with only one index we can index simultaneously many combinations of FVs. Note, however, that an index of size greater than t *will read more FVs than necessary* to perform the similarity query, thus making the search slower. Therefore, there is a trade-off between search time and number of indexed combinations. Also, note that an index of size F contains all combinations of those FVs.

The available space S may allow us to build many indices of size t or greater, and they do not need to be of the same size. As each *idx* may index more than one combination of FVs, it is possible that a specific combination c is contained by more than one index. If the query processor selects c, then the *index with smallest search cost* that contains c must be used to perform the similarity query. This is equivalent to selecting the index of *smallest size* that contains c. We use the function $ms_{\mathbb{I}}(c)$ to determine the size of this smallest index:

$$ms_{\mathbb{I}}(c) \quad = \quad \begin{cases} k & \text{if } k \text{ is the size of the smallest index in } \mathbb{I} \text{ that contains } c, \\ \infty & \text{if no index in } \mathbb{I} \text{ contains } c. \end{cases}$$

It follows that $t \leq ms_{\mathbb{I}}(c) \leq F$ if there is an index in the *iSet* that contains c. Thus, the expected search cost for the *iSet* $\mathbb{I}$ is

$$E(\mathbb{I}) = \sum_{c, ms_{\mathbb{I}}(c) < \infty} p_c \cdot Search(ms_{\mathbb{I}}(c)) + \sum_{c, ms_{\mathbb{I}}(c) = \infty} p_c \cdot LS(t).$$

The space constraint must be respected, thus

$$R(\mathbb{I}) = \sum_{i=1}^{|\mathbb{I}|} Space(size(idx_i)) \leq S,$$

where $size(idx_i)$ returns the size of idx_i, i.e., the number of FVs that idx_i contains.

Once the functions $Search(k)$ and $Space(k)$ are appropriately defined, the question is, what is the optimal *iSet* to build, given the probabilities of selecting each combination of FVs? To find the optimal solution, we have to take into account that:

- There is a trade-off between index size and search cost: An index that contains more FVs will index more combinations, but its search time will be *longer* than the search time of a smaller index.

- A combination may be indexed by many indices, but the search system must use the one with smallest search cost.
- We have a limited space S available to build indices.

Therefore, the problem can be formalized as the following optimization problem: *Given a set of combinations of FVs, their probabilities of being selected, and the search and space cost functions for the index structures, find the optimal* iSet *so that the expected search cost is minimized, given that there is a limited amount of space for building indices.*

Example using a compressed linear scan

Now we present a small concrete example that clarifies how an *iSet* works. We define the cost functions and calculate the optimal solution, given the probabilities of using a combination and the allowed amount of space.

Let us suppose that the VA-File (see Section 2.4.1) is used as the index structure. Its associated space (in bits) and search cost are

$$Space(k) = kdnb \text{ and } Search(k) = kdnb + RefinementStep,$$

where b is the number of bits used for each dimension and $RefinementStep$ is the cost of checking the non-discarded points (if b is too low, the index scan is fast but the refinement step may be expensive, because there will be many non-discarded points). Suppose also that $F = 6$, $t = 2$, $Search(F) = 1$ (which implies that $Search(k) = \frac{k}{F}$ if the same number of bits per dimension is used, as the cost of the VA-File is linearly dependent on the number of indexed FVs), $Space(F) = F$ (which also implies that $Space(k) = k$), and let $LS(t) = K \cdot Search(t)$, i.e., to search using a linear scan is K times slower than using an index of size t for the selected combination of FVs. For our computations, we used $K = 10$ [Weber et al., 1998; Chakrabarti and Mehrotra, 1999; Qian et al., 2003]. These selections were only done to facilitate the computation of the search cost, and other values can be used without affecting the behavior of the *iSet* (though, probably, the optimal solution may be different). Finally, let us suppose that the probabilities of selecting the combinations of FVs are as shown in Table 4.5.

If $S \leq 1$, there is not enough space to build an index. Thus, the search cost is trivially the cost of a linear scan, which is $10 \cdot \frac{2}{6} = \frac{20}{6} \approx 3.33$.

If $S = 2$, we have enough space to construct one index for one combination of FVs. It follows that the best decision is to index the combination $\{f_1, f_2\}$

Combination	Probability
$\{f_1, f_2\}$	0.34
$\{f_1, f_3\}$	0.33
$\{f_2, f_3\}$	0.32
All other combinations	$\approx 8.3 \cdot 10^{-4}$ (thus their sum is 0.01)

Table 4.5: Probabilities for combinations in the example

(the most frequently used one). If $\mathbb{I}_i$ denotes the optimal *iSet* for $S = i$, then the expected search and space cost for $\mathbb{I}_2$ are

$$E(\mathbb{I}_2) = 0.34 \cdot \frac{2}{6} + (1 - 0.34) \cdot \frac{20}{6} \approx 2.31, \;\; R(\mathbb{I}_2) = 2.$$

With one index we are able to reduce the search cost by a factor of 1.44x compared with a linear scan.

If $S = 3$, it follows that the best decision is to construct an index with three FV (f_1, f_2, and f_3), which contains combinations $\{f_1, f_2\}$, $\{f_1, f_3\}$, and $\{f_2, f_3\}$. In this case, the size of the index is 3 and therefore we use all the available space. The expected search cost and total space cost are

$$E(\mathbb{I}_3) = 0.99 \cdot \frac{3}{6} + (1 - 0.99) \cdot \frac{20}{6} \approx 0.53, \;\; R(\mathbb{I}_3) = 3.$$

The expected search time is reduced by a factor of 6.31x compared with a linear scan.

An interesting observation is that if we had $S = 4$, we cannot do any better. In that case, one possibility is to construct two indices for the most frequent combinations. In this case, the expected search cost is

$$0.67 \cdot \frac{2}{6} + (1 - 0.67) \cdot \frac{20}{6} \approx 1.32 > 0.53.$$

Even if the probability of using combination $\{f_3, f_4\}$ would be 0.01, it would not be a good idea to have only one index containing the four FVs:

$$\frac{4}{6} \approx 0.67 > 0.53.$$

This illustrates the fact that *it is not always optimal to use all the available space for constructing indices.*

If $S = 5$, the optimum is to build indices for $\{f_1, f_2\}$ and $\{f_1, f_2, f_3\}$. In this case, both indices contain combination $\{f_1, f_2\}$, but only the first one must be used in case the query processor selects this combination. The second index must be used for combinations $\{f_1, f_3\}$ and $\{f_2, f_3\}$. The expected search and space costs are

$$E(\mathbb{I}_5) = 0.34 \cdot \frac{2}{6} + 0.65 \cdot \frac{3}{6} + (1 - 0.99) \cdot \frac{20}{6} \approx 0.47.$$

$$R(\mathbb{I}_5) = 5.$$

The search time is reduced by a factor of 7.07x compared with a linear scan.

Finally, if we had $S = 6$, we can build three indices, one for each of the most frequently used combinations of FVs. The search and space costs are

$$E(\mathbb{I}_6) = 0.99 \cdot \frac{2}{6} + (1 - 0.99) \cdot \frac{20}{6} \approx 0.36, \ R(\mathbb{I}_6) = 6.$$

In this case, the expected search time is reduced by a factor of 9.17x compared with the linear scan. Comparing against an optimized linear scan, i.e., an index of size F (given that now we have enough space to build such an index), the speed up factor is about 2.75x.

Table 4.6 summarizes the optimal *iSets*, their space costs, and their corresponding improvement factors for the different values of S over a linear scan.

S	Optimal *iSet*	Space	Speedup
2	$\{f_1, f_2\}$	2	1.44x
3	$\{f_1, f_2, f_3\}$	3	6.31x
4	$\{f_1, f_2, f_3\}$	3	6.31x
5	$\{f_1, f_2\}, \{f_1, f_2, f_3\}$	5	7.07x
6	$\{f_1, f_2\}, \{f_1, f_3\}, \{f_2, f_3\}$	6	9.17x

Table 4.6: Improvements over a linear scan obtained with the optimal *iSet* for the example

This concrete example illustrates that the problem of finding the optimal *iSet* is not trivial. We can summarize our observations as follows:

- An incremental greedy algorithm does not guarantee that it will find the optimal solution. For example, the optimal *iSet* for $S = 6$ adds two indices to the optimal solution with $S = 5$, but it also deletes one of the constructed indices.

- It is not always optimal to use all of the available space (see the case for $S = 4$).

- The optimal solution for a given amount of available space S is not necessarily a subset of the optimal solution if one had space $S' > S$ (e.g., compare the optimal solutions for $S = 3$ and $S = 6$).

4.4.2 A binary linear program for the optimization problem

We model the *iSet* problem as a *binary linear program*, which allows us to find the optimal *iSet* using an integer linear programming optimization package (e.g., GLPK or CPLEX).

Let $\mathbb{C} = \{c\}$ be the set of combinations of t FVs. Let us define the set of all possible indices $\mathbb{J}_{K,L} = \left\{ idx_{k,\ell} : t \leq k \leq F, 1 \leq \ell \leq \min\left[\left\lfloor \frac{S}{Space(k)} \right\rfloor, \binom{F}{k}\right]\right\}$, so $idx_{k,\ell}$ is the ℓ^{th} index of size k. Figure 4.36 illustrates an example of a set $\mathbb{J}_{K,L}$ (with $F = 6$, $t = 2$, $S = 9$, and $Space(k) = k$). Note that $\mathbb{J}_{K,L}$ enumerates all the possible indices of different sizes that fit in the available space. We still need to decide which FVs will be inserted on the actually constructed indices.

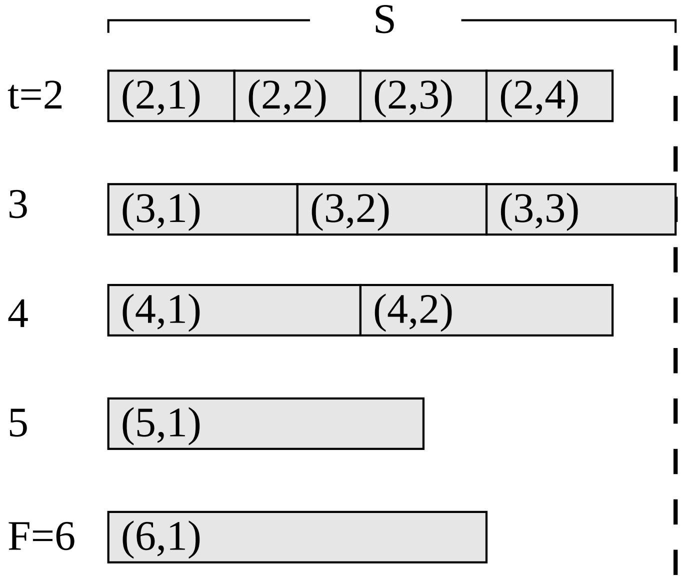

Figure 4.36: Example of set $\mathbb{J}_{K,L}$

We extend $\mathbb{J}_{K,L}$ with a special index $idx_{\infty,\ell}$ that contains non-indexed combinations of FVs (i.e., if $k = \infty$ then $Search(\infty) = LS(t)$) and uses no space. We also introduce the following binary variables:

$$x_{f,k,\ell} = \begin{cases} 1 & \text{if } f \text{ belongs to } idx_{k,\ell}, \\ 0 & \text{if not.} \end{cases}$$

$$y_{c,k,\ell} = \begin{cases} k & \text{if } idx_{k,\ell} \text{ is the cheapest index s.t. } c \in idx_{k,\ell}, \\ 0 & \text{if not.} \end{cases}$$

$$z_{k,\ell} = \begin{cases} 1 & \text{if index } idx_{k,\ell} \text{ exists}, \\ 0 & \text{if not.} \end{cases}$$

Variable $x_{f,k,\ell}$ associates FVs and indices, i.e., this variable is equal to 1 if f is in $idx_{k,\ell}$. Variable $y_{c,k,\ell}$ indicates which of the constructed indices is the cheapest one to perform a similarity query for a given combination c. Variable $z_{k,l}$ indicates which indices from $\mathbb{J}_{K,L}$ are constructed.

Given these variables and parameters, we describe the constraints of the problem. The space constraint can be written as:

$$\sum_{k,l} Space(k) \cdot z_{k,l} \leq S. \tag{4.1}$$

Next, we ask that every combination is indexed (notice that, in the worst case, this happens in the special index $idx_{\infty,\ell}$):

$$(\forall c) \sum_{k,\ell} y_{c,k,\ell} = 1. \tag{4.2}$$

Note that this only guarantees that exactly one index is considered in the search cost of the combination, but not that this index has the minimum search cost. We will show later that the actual cost will be associated with the cheapest index.

Now, we set the relation between a combination and the FVs. A combination belongs to an index if all the FVs in the combination are in the index:

$$(\forall c, k, \ell)\ y_{c,k,\ell} \leq \frac{1}{t} \sum_{f \in c} x_{f,k,\ell}. \tag{4.3}$$

The right-hand size of the equation is strictly smaller than 1 (thus it forces $y_{c,k,\ell} = 0$) unless the t FVs in the combination are present in the index $idx_{k,\ell}$.

Next, we need to fix the capacity of an index in number of FVs:

$$(\forall k, \ell) \sum_{f} x_{f,k,\ell} = k \cdot z_{k,\ell}. \tag{4.4}$$

This means that if the system decides to build an index of size k, this index contains exactly k FVs. Having less than k FVs would waste space, and having more than k FVs is infeasible.

Finally, we write the target function (to be minimized), which has the expected search cost:

$$E = \sum_{c,k,\ell} p_c \cdot Search(k) \cdot y_{c,k,\ell}. \tag{4.5}$$

At this point we can show that, as long as the search cost increases with the size of the indices, the cost of a combination c is counted only in the smallest (i.e., cheapest) index. Indeed, if the FVs of a combination c are present in two indices $idx_{k,\ell}$, $idx_{k',\ell'}$ such that $k < k'$ and $y_{c,k',\ell'} = 1$, then setting $y_{c,k,\ell} = 1, y_{c,k',\ell'} = 0$ is a feasible solution with a strictly smaller cost, i.e., in the optimum, for any fixed c, the variable $y_{c,k,\ell}$ equals 1 if and only if index $idx_{k,\ell}$ contains combination c and k is minimum.

Finally, we observe that the size of the above described binary linear program is polynomial in the size of the input of the problem. If we use b bits to represent each probability of $\mathbb{C}$, it follows that the input size of the problem is $\log_2(F) + \log_2(t) + \log_2(S) + b \cdot \binom{F}{t}$.

Note that $|\mathbb{J}_{K,L}| \le 1 + (F - t + 1) \cdot \left\lfloor \frac{S}{Space(t)} \right\rfloor \le 1 + F \cdot \left\lfloor \frac{S}{Space(t)} \right\rfloor$, that $S < \binom{F}{t} \cdot Space(t)$ (otherwhise, the solution is trivial: build an index for each combination of FVs), which implies that $\left\lfloor \frac{S}{Space(t)} \right\rfloor \le \binom{F}{t}$, and that $F \le |\mathbb{C}| = \binom{F}{t}$ (unless $t = 0$, which is not valid in our problem, or $t = F$, in which case the solution to the problem is trivial). Thus, the number of variables is

$$\begin{aligned}
variables &= |\mathbb{F}||\mathbb{J}_{K,L}| + |\mathbb{C}||\mathbb{J}_{K,L}| + |\mathbb{J}_{K,L}| \\
&\le F \cdot \left(1 + \cdot F \cdot \left\lfloor \frac{S}{Space(t)} \right\rfloor\right) + \binom{F}{t} \cdot \left(1 + F \cdot \left\lfloor \frac{S}{Space(t)} \right\rfloor\right) \\
&\quad + \left(1 + F \cdot \left\lfloor \frac{S}{Space(t)} \right\rfloor\right) \\
&< \left(F + \binom{F}{t} + 1\right) \cdot \left(1 + F \cdot \binom{F}{t}\right) \\
&\le \left(2 \cdot \binom{F}{t} + 1\right) \cdot \left(\binom{F}{t}^2 + 1\right) \\
&= 2 \cdot \binom{F}{t}^3 + \binom{F}{t}^2 + 2 \cdot \binom{F}{t} + 1
\end{aligned}$$

and the number of constraints is

$$\begin{aligned}
\mathit{constraints} &= 1 + |\mathbb{C}| + |\mathbb{C}||\mathbb{J}_{K,L}| + |\mathbb{J}_{K,L}| \\
&< 1 + \binom{F}{t} + \binom{F}{t} \cdot \left(1 + F \cdot \binom{F}{t}\right) + \left(1 + F \cdot \binom{F}{t}\right) \\
&\leq 1 + \binom{F}{t} + \left(\binom{F}{t} + 1\right) \cdot \left(1 + \binom{F}{t}^2\right) \\
&= \binom{F}{t}^3 + \binom{F}{t}^2 + 2 \cdot \binom{F}{t} + 2.
\end{aligned}$$

Therefore, the binary linear program instance that we build has polynomial size with respect to the original problem. An upper bound of the total size of the binary linear program is $\mathit{variables} \cdot b + (\mathit{variables} + \log_2(S)) \cdot \mathit{constraints} \approx O\left(\binom{F}{t}^6\right)$.

On the one hand, our model shows that finding the optimum *iSet* is *NPO* (note that we have not proved that our problem is NPO-Complete, for that we still need to prove that the problem is NPO-Hard). On the other hand, binary linear programs are *NPO-Hard* to be solved in the general case [Garey and Johnson, 1979; Hromkovic, 2001], so having this formulation does not provide an efficient way to find the optimum. Nevertheless, standard methods such as the *Branch and Bound* [Garfinkel and Nemhauser, 1972] can be used for small instances or to obtain an approximation to the optimal solution.

Also note that our model is general in the sense that it can be used with any index structure. One only needs to define the *Search* and *Space* cost functions appropriately, depending on which index structure is used. It is even possible to use different index structures for different index sizes: The binary linear program ensures that the optimal solution will be found, given the set of parameters (cost functions, available space, and probabilities for the combinations). Therefore, if more efficient index structures for combinations of FVs are created, our proposed model can always take advantage of them, finding the optimal use of the available space for building indices.

At the beginning of this section, it was assumed that all FVs had the same dimensionality. If this is not the case, the binary linear program is still able to find the optimal *iSet*. The only difference is that the cost functions $\mathit{Search}(k, d)$ and $\mathit{Space}(k, d)$ need to be defined for all the used dimensionalities.

4.4.3 Bounds for the optimal solution

Now we analyze the upper and lower bounds for the minimum expected search cost. Let $P(S) = \sum_{i=1}^{\lfloor S/Space(t)\rfloor} p_{c_i}$ and $M(t,S) = \min\{Search(t+1), LS(t)\}$.

Theorem 2. *Assume that*

1. *$Search(t) < LS(t)$, $Search(t+1) > Search(t)$.*
2. *$\mathbb{C} = \{c_i\}_{i=1,\ldots,T}$ is the set of combinations, with probabilities p_i, $p_{i+1} \geq p_i$.*

Let $E(\mathbb{I})$ be the expected cost of iSet $\mathbb{I}$ *and $OPT(S)$ be the minimum expected search cost when there is space S for indexing.*

Then, upper and lower bounds for the minimum expected search cost are given by

$$OPT(S) \leq Search(t) \cdot P(S) + LS(t) \cdot (1 - P(S)), \tag{4.6}$$

and

$$OPT(S) \geq Search(t) \cdot P(S) + M(t,S) \cdot (1 - P(S)). \tag{4.7}$$

Proof. We start proving inequality (4.6). Consider the *iSet* $\mathbb{I}$ that consists of $\lfloor S/Space(t)\rfloor$ indices of size t indexing the combinations with highest probabilities, and that does not index any other combination. Then $E(\mathbb{I}) = Search(t) \cdot P(S) + LS(t) \cdot (1 - P(S))$, but $OPT(S) \leq E(\mathbb{I})$, so we are done.

For inequality (4.7), we distinguish two cases:

i. Either $Search(t+1) \geq LS(t)$, in which case no index of size $t+1$ is created, so we find ourselves in the case of Section 4.4.1 (indexing single combinations) and the result follows; or

ii. $Search(t+1) < LS(t)$, and therefore the optimum solution may use indices of different sizes.

Let $A = \{c_i : c_i \text{ is contained by an index of size } t\}$ and $B = \mathbb{C} - A$. It follows that $|A| \leq \lfloor S/Space(t)\rfloor$. Let $ms_{\mathbb{I}}(c_i)$ be the size of the smallest index that contains combination c_i (to simplify the notation, here we assume $Search(\infty) = LS(t)$). Then

$$
\begin{aligned}
OPT(S) &= \sum_{c_i \in C} Search(ms_{\mathbb{I}}(c_i)) \cdot p_{c_i} \\
&= \sum_{c_i \in A} Search(t) \cdot p_{c_i} + \sum_{c_i \in B} Search(ms_{\mathbb{I}}(c_i)) \cdot p_{c_i} \\
&\geq \sum_{c_i \in A} Search(t) \cdot p_{c_i} + \sum_{c_i \in B} Search(t+1) \cdot p_{c_i} \\
&= Search(t) \cdot p(A) + Search(t+1) \cdot (1 - p(A))
\end{aligned}
$$

where $p(A) = \sum_{c_i \in A} p_{c_i}$.

Because $Search(t+1) > Search(t)$, the right side is minimized when $p(A)$ is maximum. But $|A| \leq \lfloor S/Space(t) \rfloor$, therefore the maximum value of $p(A)$ is attained when $|A| = \lfloor S/Space[t] \rfloor$, i.e., then $p(A) = P(S)$, which concludes the proof. □

4.4.4 Algorithms for solving the optimization problem

We propose three greedy algorithms to find good solutions for the optimization problem efficiently. In Section 4.4.5, we show that our algorithms find solutions close to the optimal values.

Algorithm A

The first algorithm starts with $\mathbb{I} = \emptyset$. On each iteration, the algorithm performs the best of:

- Adding a new index of size t (if there is enough available space), and
- Adding a FV to one of the indices already in $\mathbb{I}$.

For both possible actions, the algorithm selects the one that minimizes the expected search cost. The algorithm iterates until there is no more available space or none of the actions improves the expected search cost. Algorithm 4.7 depicts the pseudocode for this algorithm.

Algorithm B

The second algorithm starts with an *iSet* that contains indices of size t for all combinations of t FVs. Then, the algorithm searches for two indices to merge, such that the ratio of the increase in the expected search cost and the amount of saved space is minimal. The merge operation frees some of the

Algorithm 4.7: Algorithm A

```
Input: $\mathbb{F}$, $t$, $S$, probabilities $p_{c_i}$
Output: $\mathbb{I}$
$\mathbb{I} \leftarrow \emptyset$;
$avSpace \leftarrow S$;
$mincost \leftarrow LS(t)$;
repeat
    $flag \leftarrow STOP$;
    // Add an index
    if $Space(t) \leq avSpace$ then
        forall $idx$ of size $t$ do
            if $idx \notin \mathbb{I}$ then
                $cost \leftarrow E(\mathbb{I} \cup \{idx\})$;
                if $cost < mincost$ then
                    $mincost \leftarrow cost$;
                    $iSetNew \leftarrow \mathbb{I} \cup \{idx\}$;
                    $flag \leftarrow ADD$;
    // Expand an index
    forall $f \in \mathbb{F}$ do
        forall $idx \in \mathbb{I}$ do
            if $f \notin idx$ then
                $idx' \leftarrow idx \cup f$;
                $cost \leftarrow E((\mathbb{I} - \{idx\}) \cup \{idx'\})$;
                if $cost <$
                $mincost \wedge avSpace + Space(|idx|) - Space(|idx|+1) \geq 0$
                then
                    $mincost \leftarrow cost$;
                    $iSetNew \leftarrow (\mathbb{I} - \{idx\}) \cup \{idx'\}$;
                    $m \leftarrow |idx|$;
                    $flag \leftarrow EXPAND$;
    // Selecting best option
    if $flag \neq STOP$ then $\mathbb{I} \leftarrow iSetNew$;
    switch $flag$ do
        case $ADD$ $avSpace \leftarrow avSpace - Space(t)$;
        case $EXPAND$
        $avSpace \leftarrow avSpace + Space(m) - Space(m+1)$;
until $avSpace = 0$ or $flag = STOP$ ;
```

space used by the indices, but the method also takes care (implicitly) that the search cost does not increase too much. The algorithm iterates until the used space for the *iSet* is equal or smaller than S. Algorithm 4.8 depicts the pseudocode for this algorithm.

Algorithm 4.8: Algorithm B

```
   Input: $\mathbb{F}$, $t$, $S$, probabilities $p_{c_i}$
   Output: $\mathbb{I}$
1  $\mathbb{I} \leftarrow$ set of all indices of size $t$;
2  $usedSpace \leftarrow \binom{F}{t} \cdot Space(t)$;
3  while $usedSpace > S$ do
4      $ratio \leftarrow \infty$;
       // Merge two indices
5      forall $i, j \in \mathbb{I},\ i \neq j$ do
6          $idx \leftarrow i \cup j$;
7          $cost \leftarrow E((\mathbb{I} - \{i\} - \{j\}) \cup \{idx\})$;
8          $saved \leftarrow Space(|i|) + Space(|j|) - Space(|idx|)$;
9          if $saved > 0$ and $cost/saved < ratio$ then
10             $iSetNew \leftarrow (\mathbb{I} - \{i\} - \{j\}) \cup \{idx\}$;
11             $ratio \leftarrow cost/saved$;
12             $freed \leftarrow saved$;
13     $\mathbb{I} \leftarrow iSetNew$;
14     $usedSpace \leftarrow usedSpace - freed$;
```

Note that this algorithm only works if $S \geq Space(F)$. When $S = Space(F)$, it simply returns one index that contains all FVs. Thus, it is not possible to save space by merging indices once this solution is reached.

Algorithm C

The third algorithm is a slight modification of algorithm A. Instead of starting with an empty *iSet*, it starts with an *iSet* that contains $\lfloor S/Space(t) \rfloor$ indices of size t for the most frequently used combinations. Then, while there is available space, it tries to expand the indices if this further decreases the expected search cost.

Recall that

$$P(S) = \sum_{i=1}^{\lfloor S/Space(t) \rfloor} p_i \text{ and } M(t,S) = \min\{Search(t+1), LS(t)\}.$$

Theorem 3. *Algorithm C finds an* iSet *whose expected search cost is at most*

$$\frac{Search(t) \cdot P(S) + LS(t) \cdot (1 - P(S))}{Search(t) \cdot P(S) + M(t, S) \cdot (1 - P(S))}$$

times the minimum expected search cost.

Proof. It follows directly from Theorem 2 and the fact that if it is not convenient to expand any of the indices of size t, then the algorithm will return as solution $\lfloor S/Space(t) \rfloor$ indices of size t for the most probable combinations. This solution has an expected search cost equal to the upper bound (equation (4.6)) of the optimal solution. Thus, any solution returned by algorithm C which includes an index of size greater than t must have an expected search cost lower than the upper bound. □

Notice that if $M(t, s) = LS(t)$, the algorithm C finds the optimal solution. Also, notice that Theorem 3 also holds for an algorithm that returns the *iSet* that has a minimum expected search cost value between those returned by algorithms A, B, and C.

4.4.5 Experimental evaluation

We used a real dataset to compare the *iSets* obtained by the binary linear program and the proposed algorithms. The dataset consists of 3D models collected from the Internet. For this dataset, we implemented 16 different FVs for 3D models, which include volumetric descriptors, surface descriptors, and image-based descriptors. See Section 3.1.2 for a detailed explanation of the experimental framework.

We pre-processed the implemented FVs before computing the combinations. First, we applied a PCA-based dimensionality reduction that all FVs have the same dimensionality. We applied PCA to all FVs and then we kept the 32 principal axes of each FV (32-D was the smallest dimensionality in the original set of FVs). Then, we normalized the coordinate values of all FVs in the range $[0, 1]$.

To compute the probabilities of using a given combination of FVs, we used the *entropy impurity method* (cf. Section 3.3.2). The t FVs with the smallest entropy impurity values were selected for the combination. We ran our set of queries and let the query processor select the best combination, storing which combination was selected for each query. We used the frequency of selection of each combination as its probability of being selected by the query processor.

For the space and search cost functions, we used the same of Section 4.4.1, i.e., we used a VA-File as index structure. We considered two sets of FVs ($F = 6$ and $F = 16$) and two combination sizes ($t = 2$ and $t = 3$).

For all the values of F and t, we ran our algorithms, calculated the lower and upper bounds of Theorem 2, and solved the binary linear program over different values of S. Finally, we normalized the results by the search cost of a single index of size F.

To solve the binary linear program, we used the CPLEX linear optimization solver, version 7.5. Our machine has two Pentium IV 3.7 Ghz processors running Fedora Linux 4, with 1Gb of RAM.

Experimental results

Figure 4.37 shows the results for the binary program in the case $F = 6, t = 2$, and $S = 6, \ldots, 12$, as well as the search cost of a single index containing all FVs, the general lower and upper bound, and the search cost when each combination is indexed in its own index of size t (which is feasible only if $S \geq \binom{6}{2} \cdot Space(t) = 30$). The reduction in the search cost is important. For instance, having twice the space of the naïve solution (one index of size 6) reduce the expected search cost by a factor of 2x.

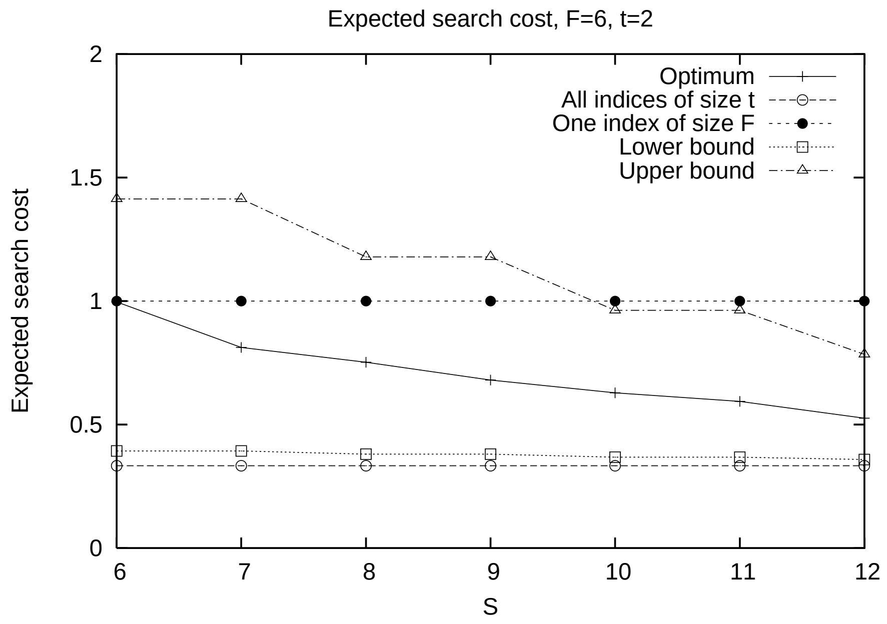

Figure 4.37: Expected search cost of the optimal *iSet*, $F = 6,\ t = 2$

Figure 4.38 (respectively 4.39) shows the solutions obtained by the algorithms for the case $F = 6, t = 2$ (respectively $t = 3$) and $S = 6, \dots, 12$, as well as the optimum value (from the binary linear program) and the lower and upper bounds of Theorem 2. From these figures, it can be seen that, even though algorithm C did not find good *iSets*, it was worthwhile to have indices of size greater than t. Indeed, this was the reason for the drastic reductions in the search cost obtained by algorithm C when S is not divisible by t. The results of algorithm A for the case $t = 3$ were optimal over the whole range of S values.

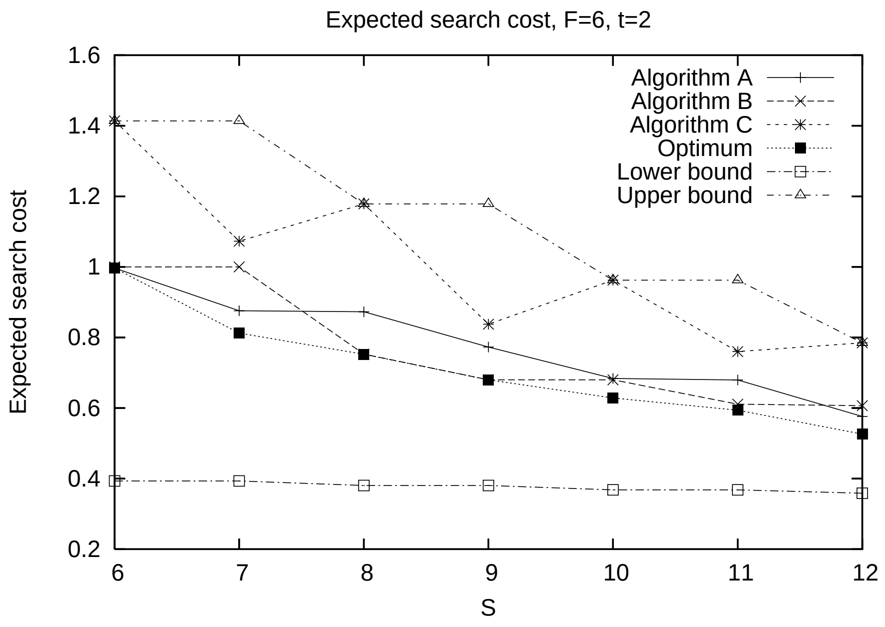

Figure 4.38: Expected search cost of the *iSets* returned by the algorithms, $F = 6,\ t = 2$

For the case $F = 16$, neither the binary linear program, nor algorithm B (for the case $t = 3$) gave a solution within a reasonable amount of time (less than 1 day), so we present results only regarding algorithms A, B (for $t = 2$), and C, as well as the theoretical lower and upper bounds. Figure 4.40 (respectively 4.41) shows the results for $t = 2$ (respectively $t = 3$) when $S = 12, \dots, 24$. Here it becomes clear that in some important cases the only practical way to get a good solution is by resorting to approximation algorithms.

Finally, Table 4.7 shows the running times of the algorithms and the binary linear program for $F = 6$ and $t = 2$. The first column shows the available space S. The second, third, and fourth columns show the running time for algorithms A, B, and C, respectively. The fifth column shows the

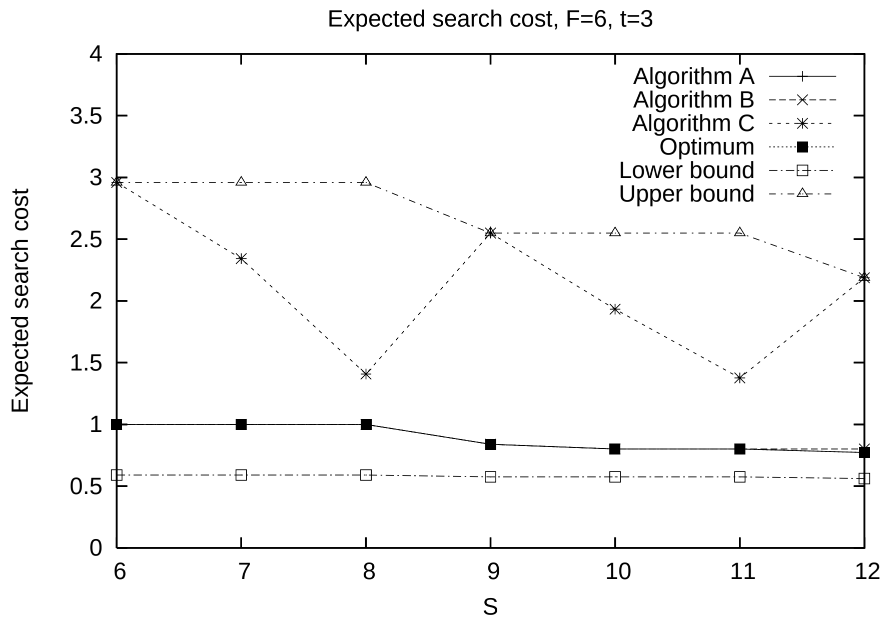

Figure 4.39: Expected search cost of the *iSets* returned by the algorithms, $F = 6,\ t = 3$

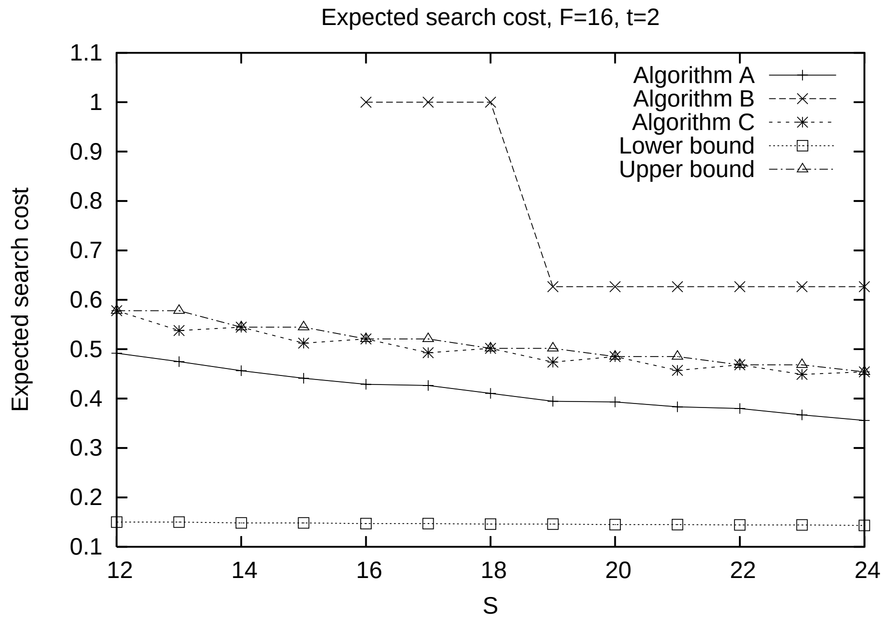

Figure 4.40: Expected search cost of the *iSets* returned by the algorithms, $F = 16,\ t = 2$

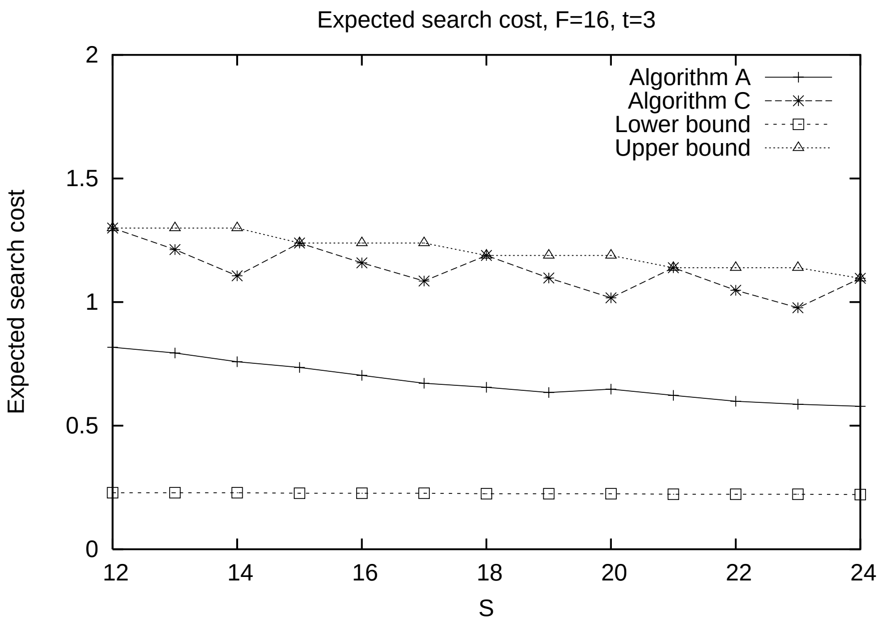

Figure 4.41: Expected search cost of the *iSets* returned by the algorithms, $F = 16,\ t = 3$

running time of the binary linear program. It follows that the time needed by the binary linear program increased steeply with S, and that the approximated algorithms are at least one order of magnitude faster than the binary linear program.

S	A	B	C	BLP
6	0.037	0.379	0.052	0.13
7	0.038	0.344	0.028	1.41
8	0.042	0.346	0.025	3.15
9	0.046	0.349	0.030	16.51
10	0.054	0.350	0.025	76.83
11	0.060	0.344	0.032	451.70
12	0.066	0.347	0.025	765.95

Table 4.7: Time (in seconds) needed for the algorithms and the binary linear program to find the solution

Analysis of the results

From algorithms A, B, and C, the best overall was algorithm A. It was fast to compute and returned nearly optimal solutions: About 4% in average from the optimum in the cases where we could calculate the optimum using the binary linear program.

Algorithm B also returned good *iSets* (sometimes better than A), but it became too slow when the total number of combinations T became *large* (when $F = 16, t = 3$, then $T = \binom{16}{3} = 560$). This is because the initial solution for this algorithm contains exactly one index per combination and it is $O(T^2)$ on each iteration. Also, this algorithm does not work when $S < Space(F)$.

Algorithm C was the fastest, but it produced the worst results compared to the other algorithms. Its only advantage is that we can prove a guarantee (Theorem 3) on the relative error of its output. Even though algorithm A always performed better than C in our experiments, it is possible to construct instances in which algorithm C performs better than A, thus it is not possible to apply Theorem 3 to algorithm A.

Therefore, a good compromise would be to use the *iSet* with minimum expected search cost between the outputs of algorithms A and C.

Finally, notice that in Figure 4.41 algorithm A produced a better solution for $S = 19$ than for $S = 20$. This may happen when the algorithm does not have more space for building a new index of size t and therefore it expands an existing index ($S = 19$). When more available space is allowed ($S = 20$), the greedy algorithm may decide to create a new index, missing a chance to expand afterwards.

4.5 Conclusions

This chapter presented several contributions to improve the effectiveness of similarity search in multimedia databases. Here we present some conclusions with regard to each of the proposed techniques.

In Section 4.1, we defined an efficiency criterion to compare two sets of pivots, and have experimentally shown that this criterion consistently selects good sets of pivots in a variety of synthetic and real-world metric spaces, reducing the total complexity of pivot-based proximity searching when answering range queries. The proposed efficiency criterion is based on a formal theory, that takes into account the distance distribution of the mapped space defined by the selected pivots. This formalism is crucial, in contrast to simple heuristics, to consistently obtain good results in a wide scope as the one of metric spaces.

We presented three different pivot selection techniques, which use the efficiency criterion defined, and showed that the so-called *incremental selection* is the best method in practice. It was found that good pivots are outliers, but outliers are not necessarily good pivots. It is interesting to note that outlier sets have a good performance in uniformly distributed vector spaces, but have a poor performance in general metric spaces, even worse than random selection in some cases. This result raises the question if it is valid to test pivot selection techniques in uniformly distributed vector spaces.

In Section 4.2, we presented an optimized version of the optimal-order k-NN search algorithm, using distance estimators (such as the upper and lower bound distance to the query object) in order to reduce the storage requirements of the search algorithm. Our proposed algorithm aims to filter out from the active page list, as soon as possible, all nodes from the index where it is ensured that no relevant objects can be found. We also introduce the concept of *bubble*, which are "abstract" index nodes with no elements inside. Bubbles can be used to filter out index nodes from the active page list using the distance estimators, even if the algorithm has not yet visited the index nodes which actually contain the objects inside the bubble. We tested our algorithm with several synthetic and real-world datasets, using two state-of-the-art index structures for metric spaces. The experimental results confirmed that the storage requirements of our proposed algorithm are considerably smaller compared with the standard k-NN algorithm (up to 5 times smaller).

Although we focused on indices for metric spaces, the improved k-NN algorithm is general and can be adapted with minimal effort to be used with spatial access methods. In that case, each subtree is usually bounded by a hyperrectangle. Our heuristic translates into the following rule: *Assume a tree node contains $bsize$ elements within a hyperrectangle. Then there are $bsize$ elements at a distance* $\max d(q, c)$*, where c ranges among all the corners of the hyperrectangle.* This heuristic is different from the usual MinMaxDist, which gives a better distance estimator but only holds for one object per tree node. If we use the simpler rule from Samet [2003] and translate it to a spatial data structure, the result is always inferior to the MinMaxDist heuristic.

In Section 4.3, we proposed a pivot-based index for combinations of feature vectors. As we showed in Chapter 3, combinations of features may improve the effectiveness of the similarity search. The presented index structure addresses the efficiency problem of similarity searching using more than a single feature to represent a multimedia object. We especifically described the data structure of the index and a NN search algorithm for the case of dynamically weighted combinations of feature vectors. Our experimental evaluation showed that the proposed pivot-based index improves the efficiency of the search up to a factor of more than 5x when compared to the sequential scan.

Finally, in Section 4.4 we presented methods for finding the set of indices (*iSet*) that minimizes the expected search cost of similarity queries that use dynamic combinations of feature vectors. We model the problem of finding the optimal *iSet* as a binary linear program. This provides us with a tool to find the optimal solution for small instances of the problem. We also proposed fast algorithms that are able to find good sets of indices.

The applicability of the proposed model is not restricted to the particular cases that we presented in this section. Our approach is very flexible in the sense that is not restricted to a particular dimensionality of the space, to a particular index structure, or to specific cost functions. The model can be used to evaluate different indexing schemes, different dimensionalities of the FVs, and so on: It suffices to define the cost functions, available space, and probabilities of using the combinations, and the model will return the optimal solution for that set up. If LS were to be replaced by more efficient techniques, our model can still be applied to find the best indexing. It suffices to change the corresponding parameter in the model. The same applies if, for instance, more efficient index structures become available.

Chapter 5

Hardware acceleration of feature-based similarity search

Modern graphics processor units (GPUs) consist of several parallel working stream processors, and the memory bandwidth of state-of-art graphic cards is much higher than in any high-end conventional desktop PC. These GPUs are now capable of processing several million vertex coordinates or fragments per second, and actually provide more processing power than the last generation of CPUs. At this time, only a few publications focus on graphic hardware acceleration for database related algorithms.

In this chapter, we explore the practical usage of GPUs as co-processors for database applications and show its high potential. We present detailed descriptions of GPU implementations of two basic database algorithms, namely high-dimensional nearest neighbor search and clustering. Since GPUs are designed to perform graphic primitives, the implementation of database algorithms requires some innovating and challenging data encoding and GPU programming. The experimental results show that the proposed GPU algorithms are an order of magnitude faster than their CPU versions.

This work has been published in Bustos et al. [2006a].

5.1 Introduction

In the last few years, GPU technology has improved much faster than CPU technology. For example, the processing power of the *GeForce 6800 Ultra* GPU peaks at 40 GFlops (the Intel Pentium IV CPU (3 GHz) provides about 6 GFlops peak performance), its processing speed has improved to 600 million vertices per second, and its memory bandwidth has increased to more than 35 GBytes/s [Fan et al., 2004; NVIDIAa, 2005]. The main driving force behind this rapid development are computer games and other multimedia applications, which require extensive graphic processing capabilities to generate realistic scenes in real time. Due to the fast growing computer game industry, this development will undoubtedly continue. Currently, graphic cards connect to a very high performance bidirectional data bus (PCI Express (PCIe)) with an acceptable price-performance ratio. Also, it is now possible to use multiple graphic cards in a single host computer. It is therefore interesting to develop algorithms which exploit the power of current GPUs to speed up general computations.

Recently, there have been research projects which have dealt with new programming interfaces, that allow GPUs to be used for general purpose computations. For example, the recently proposed GPU programming interface *Brook* [Buck et al., 2004] is a generic system, which extends the C language to facilitate the usage of GPUs as streaming co-processors. The idea is to code special functions called *kernels* that run on the GPU using a high level C-style programming language. Brook transforms and compiles the kernels into assembly language for the GPU, thus allowing a fast development of general purpose programs for GPUs. Application examples are the Fast Fourier Transform [Moreland and Angel, 2003], a framework for linear algebra operators [Krüger and Westermann, 2003], and an algorithm for computing 3D distance fields [Sud et al., 2004]. Unfortunately, by using Brook there are some performance drawbacks.

Recent publications study the usage of GPUs as co-processors for database applications. Not surprisingly, the first papers mainly focused on graphics related operations in spatial databases. Sun et al. [2003] propose methods to accelerate the refinement step of spatial selections and joins using the GPU; and following the same trend, Bandi et al. [2004] show how to integrate the hardware acceleration provided by GPUs with a commercial DBMS for spatial operations. One of the first papers which focused on general database operations, such as predicate evaluation, boolean combination, and aggregation, is by Govindaraju et al. [2004]. Their experimental results showed that GPUs can be used as a co-processor for some of these common database operations. However, the limited scale experiments also showed that some of the operations were much slower than with the CPU implementation.

We present GPU implementations for two important database applications: nearest neighbor search and clustering. The aim of these proposed implementations is to achieve the maximum performance with these algorithms. The proposed GPU-based solutions are evaluated using large real and synthetic datasets. In contrast to most of the previous work, this research is not focused on general conception or theory, but rather illustrates a more efficient and practical approach to implement GPU-based algorithms.

5.2 Graphics hardware

A modern graphics adapter can be seen as a parallel computer with its own memory, that works on a large stream of data records. This section outlines the data processing pipeline currently used in such adapters, and the resulting requirements and restrictions for programming GPUs to deal with database operations.

5.2.1 The rendering pipeline

Nowadays, high end graphics accelerators, such as *NVIDIA's GeForce series* [NVIDIAa, 2005], contain programmable parallel working stream processor units. To properly take advantage of these powerful units and to use their full potential for any kind of computational work, it is first necessary to take a look at the *rendering pipeline*, that explains how a geometric data stream is transformed into images. Figure 5.1 shows a diagram of this pipeline in modern graphics hardware. The algorithms presented in this chapter work almost completely within the rasterization stage. These units are the most efficient/parallel parts of the data pipeline in today's graphics hardware.

Data for rendering consist basically of geometric primitives and connected *texture* information. Textures are digital images that are projected on the primitives to enhance realism by using *texture coordinates.* These coordinates control the fine addressing of texture elements (called *texels*): A texture is a 2D array of texels (see Figure 5.2). Each texel $t_{i,j}$ contains 4 color channels (red, green, blue, and alpha), i.e., it can store up to 4 float values. The proposed GPU-based algorithms will use textures to encode the database information.

The first stage in the pipeline is called *geometry setup stage.* Here, geometry data may be altered (e.g., it can be rotated, scaled, or translated in the 3D space). The pre-processed 3D data is then processed by the *rasterizer*, which samples the geometry data into a set of *fragments.* It is useful to think of fragments as potential pixels to be rendered on the screen. According to the texture information, the *color* and *depth* values of these fragments may be altered. This transformation process can be controlled with the aid of *programmable fragment processors.* A state-of-art graphics card, such as the NVIDIA GeForce 6800 Ultra, includes *16 parallel working fragment processors*, which are attached to DDR video memory with a high-speed, 256 bit wide, 550 MHz data bus. Finally, the processed fragments may become part of the final picture as pixels.

The set of fragment processors can either execute a set of fixed set of instructions defined in a graphic API (e.g., OpenGL), or a user-specified

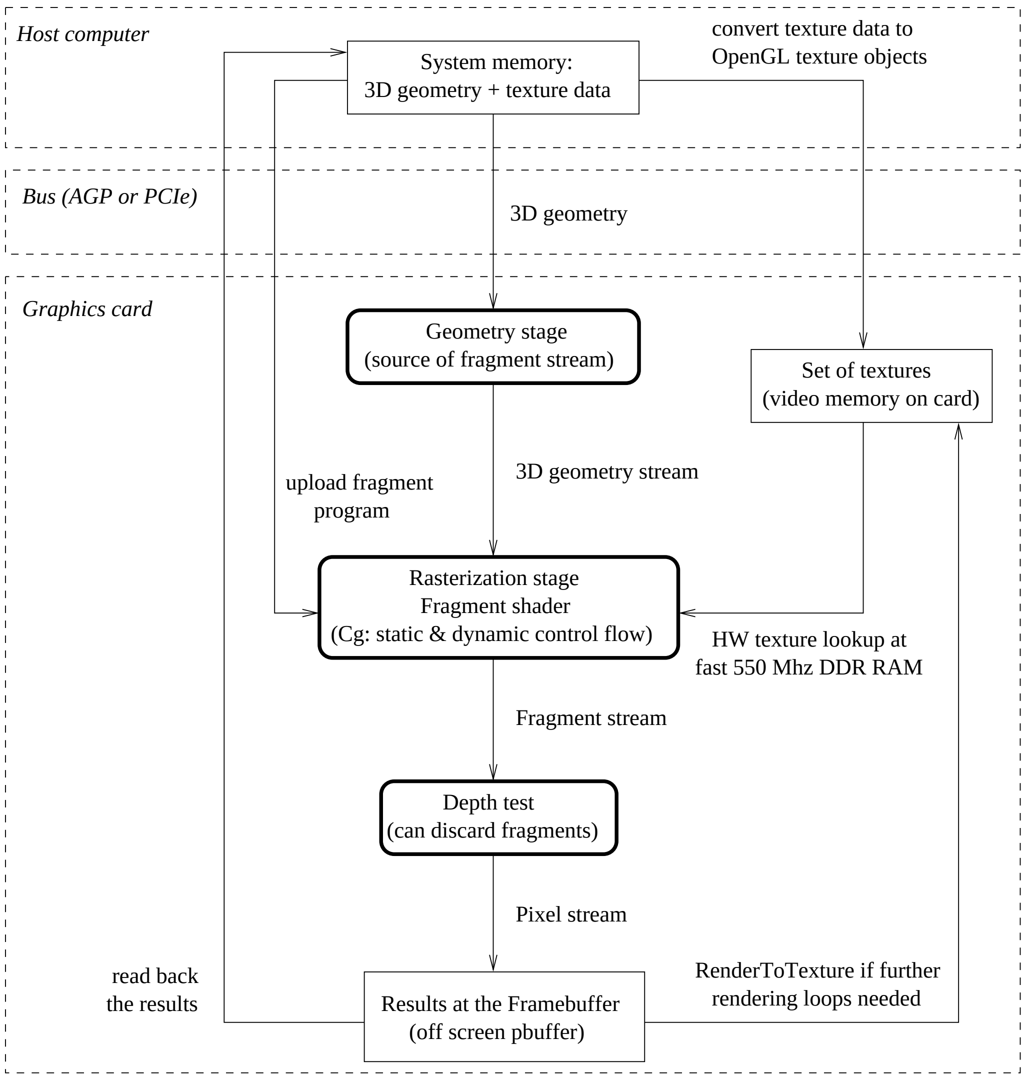
Host computer
System memory: 3D geometry + texture data
convert texture data to OpenGL texture objects
Bus (AGP or PCIe)
3D geometry
Graphics card
Geometry stage (source of fragment stream)
Set of textures (video memory on card)
3D geometry stream
upload fragment program
Rasterization stage Fragment shader (Cg: static & dynamic control flow)
HW texture lookup at fast 550 Mhz DDR RAM
Fragment stream
Depth test (can discard fragments)
Pixel stream
read back the results
Results at the Framebuffer (off screen pbuffer)
RenderToTexture if further rendering loops needed

Figure 5.1: Dataflow within GPU stages

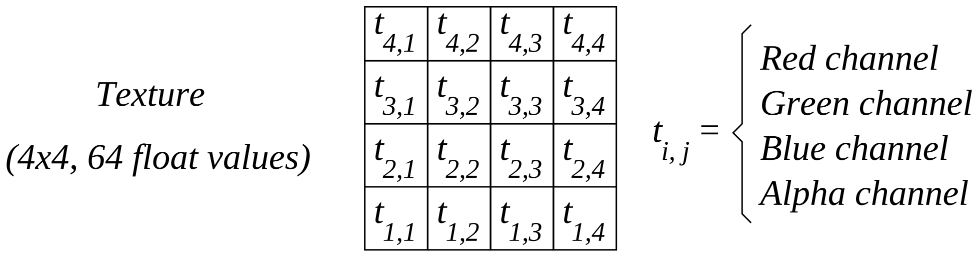
Texture
(4x4, 64 float values)
$t_{4,1}$ $t_{4,2}$ $t_{4,3}$ $t_{4,4}$
$t_{3,1}$ $t_{3,2}$ $t_{3,3}$ $t_{3,4}$
$t_{2,1}$ $t_{2,2}$ $t_{2,3}$ $t_{2,4}$
$t_{1,1}$ $t_{1,2}$ $t_{1,3}$ $t_{1,4}$
$t_{i,j}$ =
Red channel
Green channel
Blue channel
Alpha channel

Figure 5.2: Illustration of a 4×4 texture

assembly-level program called *fragment program*, which is a more flexible solution. Fragment programs consist of 4-way SIMD (Single Instruction Multiple Data) instructions. As GPUs are designed for different kinds of graphical calculations, the instructions include standard mathematical operations and a few special purpose instructions in combination with texture operations.

With a correct setup of geometry and texture data, every texel becomes a fragment in the rendering pipeline and can be processed by a fragment program. This is necessary to ensure that *all* data in the stream are processed correctly. Hence, the major task during the implementation of the algorithms is the proper data en- and decoding, to correctly convert the data into graphic pipeline compatible records.

5.2.2 GPU programming

Every fragment processor works with the same code, but on different parts of the fragment stream. A fragment program has simultaneous access of up to 16 different textures on current GeForce hardware. Fragment programs do not have any access to previous or future fragments in the data stream, which allows the hardware to work in parallel on parts of the data stream. It is possible to transfer additional parameters to the fragment programs for each stream processing pass, but only before the fragment program starts.

The general approach here is to set up the fragment programs and their input data once and run them using the highest possible amount of input data, because the data transfer from the host main memory to the graphics card is slow compared to video memory access, even if the data is located in the AGP memory buffer. Such transfers must be optimized, which can be done, for example, by partitioning the data or clustering the queries. To set up the input stream, the input data is converted and copied to at least one texture, making use of the modern floating point texture formats. After that, the rendering pipeline is fed with basic geometry to map the textures. Additionally, the fragment programs are loaded into hardware using the *NVIDIA's Cg toolkit* [NVIDIAb, 2005]. After doing so, a stream of data is attached to fragments, which relays the input data. The fragment programs combine these and additional data (encoded in additional textures) to compute intermediate or final results. Often, more than one rendering step is required for the whole computing process. This is a common process in computer graphics (e.g., games use up to seven rendering steps until the final (visible) image is shown on the screen). Consequently, GPU algorithms consist of a set of stream processing rendering steps. The fragment programs only work together with surrounding data encode/decode and render control routines (e.g., C/C++ and OpenGL framework).

Once the fragments have been generated, the *depth buffer*, which is associated with all fragments, is used to determine the visible pixels (*depth test*). In its standard setup, a fragment is converted into a visible pixel if its depth is smaller than the actual value in the depth buffer. In this case, the pixel is set and the new depth is adjusted. This setup can be changed in a flexible way to create special renderings and, in our case, to help compute complex database operations.

5.2.3 The Cg language

The *Cg* Language [NVIDIAc, 2005], "C for graphics", is a high level language that can be used to code fragment programs. Cg code looks very similar to C code, but it is enhanced to make it easy to code and compile programs that run on the GPU. Some of the Cg commands that will be used in the following fragment programs are:

- *float4 fv*: Declares a 4-D vector *fv.* Each coordinate can be accessed as follows: *fv.x*, *fv.y*, *fv.z*, and *fv.w.* Scalar value are declared as *float.*
- *samplerRect tex*: Addresses the different texture units.
- *float4 texRect(samplerRect tex, float2 coords)*: Returns a 4-D vector with the data stored in the texture *tex* at coordinates *(coords.x, coords.y).*
- *Swizzle operator '.'*: Efficiently converts a scalar value into a vector. E.g., if *c* is a scalar, then *c.xxxx* is a 4-D vector with all coordinates values equal to *c.*

A fragment program should always return a 4-D vector with color information. Each coordinate value of this vector corresponds to one of the 4 color channels. If the depth test is used, the fragment program also needs to return a depth value. These variables are declared as parameters of the fragment program with the keyword *out.*

- *out float4 color*: This variable returns the processed data.
- *out float depth*: This variable stores the depth value of the fragment.

5.3 Fast linear scan using the graphics unit processor

As already discussed in Chapter 2, nearest neighbor search in high dimensional spaces is an important, but challenging problem. Several indexing algorithms have been proposed for implementing this similarity query (cf. Section 2.4). However, most of the experiment results reported with spatial access methods showed that the performance of the linear scan is highly competitive for high-dimensional datasets, and that it can be faster than any index structure in such spaces (cf. Section 2.5.3). In this section, we describe a GPU version of the linear scan based NN search algorithm, which will help to keep the presentation as simple and clear as possible.

5.3.1 GPU implementation of nearest neighbor search

The first step of the algorithm is to load the vectors into the graphics card texture memory. For this purpose, d textures are created. Each of them will store one coordinate value of all vectors. The four available color channels will be used to store the data, thus each texel contains the i^{th} coordinate value of four different vectors (e.g., a 512x512 texture is needed to store 1,048,576 coordinate values). Figure 5.3 illustrates how the textures are used to store the data, using a 6-dimensional database with 4 vectors. $V_{i,j}$ represents the j^{th} coordinate of the i^{th} vector.

Three different fragment programs are used to implement the GPU-based linear scan. The first fragment program computes the distance between each object u and the query q. As distance metric we use the Manhattan distance. This simple metric was chosen because it is useful for many multimedia databases and its very fast to compute, but other metrics are also possible (e.g., any Minkowski distance). To fully exploit the potential of the GPU, the difference between coordinates is simultaneously computed for several dimensions. Algorithm 5.1 shows the actual fragment program code used for the experiments. During each pass of the algorithm, t textures, i.e., t dimensions, are processed in parallel (line 15 of the FP) for a total of d/t passes (we obtained the best results with our hardware using $t = 8$). Variables tex_i represent each texture, which contain coordinate values. Texture tex_R contains the result from previous iterations that is aggregated with the results of the current pass (initially, tex_R only contains zeros). Figure 5.4 illustrates how this fragment program works.

In the next rendering pass, the NN to q is determined by using two different fragment programs. The first one computes the minimum distance

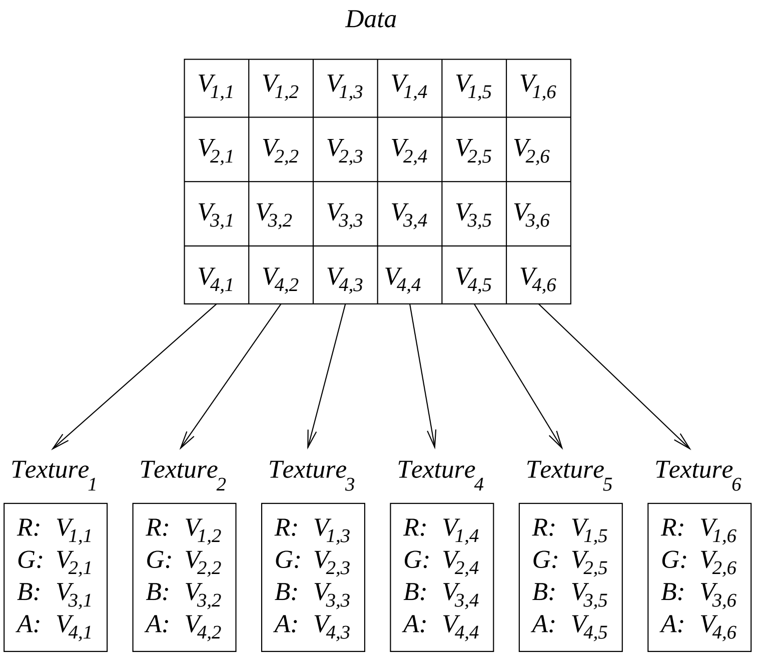
Data
$V_{1,1}$ $V_{1,2}$ $V_{1,3}$ $V_{1,4}$ $V_{1,5}$ $V_{1,6}$
$V_{2,1}$ $V_{2,2}$ $V_{2,3}$ $V_{2,4}$ $V_{2,5}$ $V_{2,6}$
$V_{3,1}$ $V_{3,2}$ $V_{3,3}$ $V_{3,4}$ $V_{3,5}$ $V_{3,6}$
$V_{4,1}$ $V_{4,2}$ $V_{4,3}$ $V_{4,4}$ $V_{4,5}$ $V_{4,6}$
$Texture_1$ $Texture_2$ $Texture_3$ $Texture_4$ $Texture_5$ $Texture_6$
R: $V_{1,1}$ G: $V_{2,1}$ B: $V_{3,1}$ A: $V_{4,1}$
R: $V_{1,2}$ G: $V_{2,2}$ B: $V_{3,2}$ A: $V_{4,2}$
R: $V_{1,3}$ G: $V_{2,3}$ B: $V_{3,3}$ A: $V_{4,3}$
R: $V_{1,4}$ G: $V_{2,4}$ B: $V_{3,4}$ A: $V_{4,4}$
R: $V_{1,5}$ G: $V_{2,5}$ B: $V_{3,5}$ A: $V_{4,5}$
R: $V_{1,6}$ G: $V_{2,6}$ B: $V_{3,6}$ A: $V_{4,6}$

Figure 5.3: Data organization for the linear scan algorithm

Algorithm 5.1: Fragment program 1: Computing Manhattan distance

```
void FragmentProgram1(
float2 coords : TEXCOORD0, // fixed texture coordinates
uniform samplerRECT tex0 : TEXUNIT0, // DB data
...
uniform samplerRECT tex7 : TEXUNIT7,
uniform samplerRECT texR : TEXUNIT8, // partial results are stored here
uniform float qc0, ..., uniform float qc7, // query vector data
out float4 color : COLOR ) {
  // fetch data related to linear interpolated tex coords, process every record once
  float4 fv0 = texRECT(tex0, coords); // reads one texel from tex0
  ...
  float4 fv7 = texRECT(tex7, coords); // reads one texel from tex7
  float4 fv = texRECT(texR, coords); // reads partial result stored on texR
  // compute partial distance over 8 dimensions (aggregated into texR)
  color = fv+abs(fv0-qc0.xxxx)+...+abs(fv7-qc7.xxxx); }
```

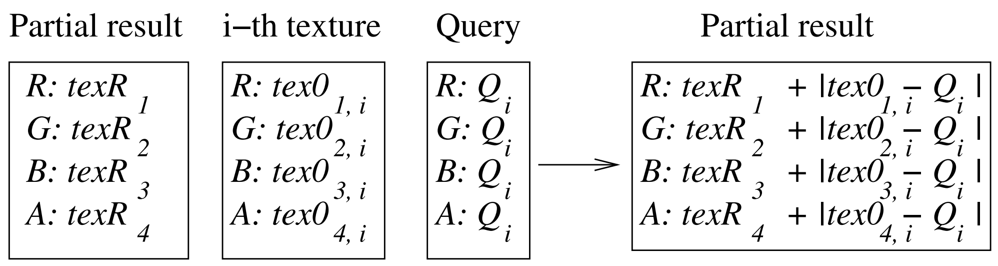

Figure 5.4: How does fragment program 1 work (one texture shown)

values within the color and alpha channels (variable min), and associates these distance values with the index of the corresponding object (variable $fvidx$). Algorithm 5.2 shows the code. Lines 9–11 compare the four values stored in each texel, and keeps the minimum value in the red channel (coordinate x). In the green channel (coordinate y), the fragment program stores the index associated with the object which has the minimum distance to the query (the index is simply an integer value between 1 and n). Figure 5.5 illustrates.

Algorithm 5.2: Fragment program 2: Computing min value of the texel attached vector of 4 distances, and associating an index value to these objects

```
1  void FragmentProgram2(
2  float2 coords : TEXCOORD0,
3  uniform samplerRECT data : TEXUNIT0,
4  uniform samplerRECT index : TEXUNIT1,
5  out float4 color : COLOR) {
6    float4 fv = texRECT(data, coords); // computed distances with
   fragment program 1
7    float4 fvidx = texRECT(index, coords); // respective indices
8    float4 min = float4(fv.x,fvidx.x,0.0,0.0);
9    if (min.x > fv.y) {min=float4(fv.y,fvidx.y,0.0,0.0);}
10   if (min.x > fv.z) {min=float4(fv.z,fvidx.z,0.0,0.0);}
11   if (min.x > fv.w) {min=float4(fv.w,fvidx.w,0.0,0.0);}
12   color = min; } // min value: (distance, index, 0, 0)
```

The last fragment program searches for the minimum distance between four appropriately selected texels, and iteratively reduces the texture size by a factor of 4 during each pass. The minimum distance and its associated index are stored in the red and green channels, respectively. This iterative reduction

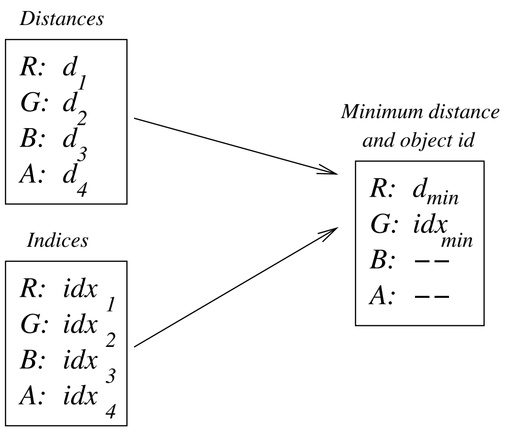

Figure 5.5: Texel processing performed by fragment program 2

is the tricky part in the minimum search algorithm, and it is simulated by storing the results on the first quarter of the original texture, and only this quarter is used at the next rendering step. Algorithm 5.3 shows the code, and Figure 5.6 illustrates how the texture reduction is performed. Note that the offset value (which depends on the texture size) is received as a parameter, and it is calculated on the CPU before invoking the fragment code. These min-searches seem somewhat complicated, but because of the restrictions given by the GPU hardware there is no other way to determine the minimum value inside a data stream. Within a single stream processing run there is no way to communicate the actual minimum value to other running fragment program instances.

The algorithm stops when the texture has been reduced to size 1×1 (if the texture generated by fragment program 2 contains $s \times s$ texels, the algorithm needs $\log_2(s)$ iterations to reduce the texture). Thus, only one texel (16 bytes) is read back from the graphics card memory, hence saving data transfer time. The index of the object with the minimum distance to q is returned as the NN.

The algorithm can easily be extended to compute the k nearest neighbors. For the case $k = 2$, the blue and alpha color channels can be used to store the distance to the 2-NN candidate. For $k > 2$, we have to use $k/2$ additional textures to store the k data points with smallest distances. At the end, the k remaining data points have to be sorted.

Algorithm 5.3: Fragment program 3: Computing minimum distance between objects stored in 4 different texels precomputed by fragment program 2

```
void FragmentProgram3(
float2 cds : TEXCOORD0, // primary tex coords
uniform float2 os, // offset tex coords
uniform samplerRECT tex : TEXUNIT0,
out float4 color : COLOR) {
  float4 min, fv2, fv3, fv4;
  // os is an offset value used to access the "right" 4 texels
  min=texRECT(tex,float2(cds.x+os.x,cds.y)); // gets 1st texel
  fv2=texRECT(tex,float2(cds.x+os.x,cds.y+os.y)); // gets 2nd texel
  fv3=texRECT(tex,float2(cds.x,cds.y+os.y)); // gets 3rd texel
  fv4=texRECT(tex,float2(cds.x,cds.y)); // gets 4th texel
  if (min.x > fv2.x) {min = fv2;} // Compute min value within
  if (min.x > fv3.x) {min = fv3;} // the 4 read texel, then
  if (min.x > fv4.x) {min = fv4;} // return only one texel
  color = min; } // min value: (distance, index, 0, 0)
```

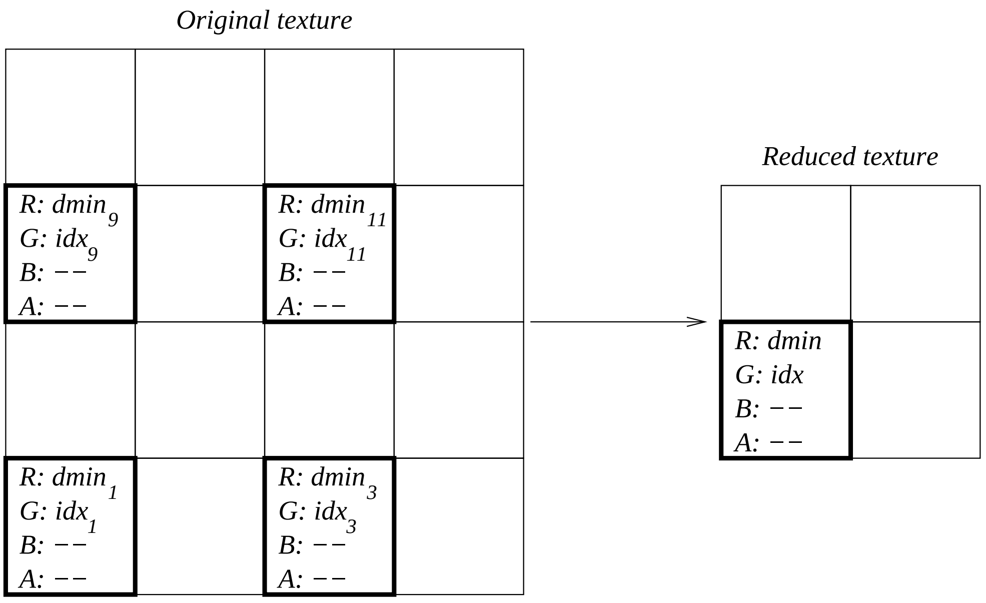

Figure 5.6: Texture reduction performed by fragment program 3

5.4 A hardware accelerated clustering algorithm

Cluster analysis is an essential task in many applications. It allows one to find *natural* clusters and describe their properties (*data understanding*), find useful and suitable groupings (*data class identification*), find representatives for homogeneous groups (*data reduction*), find unusual objects (*outliers detection*), find random perturbations of the data (*noise detection*), and so on. A clustering algorithm identifies a set of categories, classes, or groups (called *clusters*) in the database, such that objects within the same cluster shall be as *similar* as possible, and objects from different clusters shall be as *dissimilar* as possible. More formally, the clustering problem may be defined as the problem of partitioning a d-dimensional set of data vectors $\mathbb{D} = \{x_1, \ldots, x_N\} \subset \mathbb{R}^d$ into a set of clusters $\{C_1, \ldots, C_k\}$ and noise (C_0) such that $\forall_{m=1,\ldots,k}\forall_{i,j=1,\ldots,N} : x_i, x_j \in C_m \Rightarrow \mathit{similar}(x_i, x_j)$ and $\forall_{m=1,\ldots,k}\forall_{l=0,\ldots,k,m\neq l}\forall_{i,j=1,\ldots,N,i\neq j} : x_i \in C_m, x_j \in C_l \;\Rightarrow\; \mathit{not\ similar}(x_i, x_j)$.

5.4.1 Previous work

A large number of clustering algorithms have been proposed in the literature of statistics, machine learning, knowledge discovery and databases. These methods can be classified into model-based and optimization-based methods [Fukunaga, 1990; Fritzke, 1997; Zhang et al., 1999], linkage-based methods [Bock, 1974; Zhang et al., 1996; Xu et al., 1998; Ankerst et al., 1999], density-based methods [Silverman, 1986; Scott, 1992; Sheikholeslami et al., 1998; Hinneburg and Keim, 1998, 1999], and hybrid methods [Zhang et al., 1996; Agrawal et al., 1998]. Many recent approaches, developed in the Knowledge Discovery and Data Mining (KDD) community, aim at improving efficiency and effectiveness. These are based on sampling techniques [Palmer and Faloutsos, 2000], incremental techniques [Zhang et al., 1996; Ester et al., 1998], hierarchical techniques [Zhang et al., 1996; Wang et al., 1997], or multidimensional indexing [Xu et al., 1998; Ankerst et al., 1999].

One of the most widely used approaches is the *k-means* algorithm and its variants [MacQueen, 1967; Hamerly and Elkan, 2002]. The basic k-means algorithm starts with an initial selection of k centroids (e.g., k random objects from $\mathbb{D}$). Each $x \in \mathbb{D}$ is assigned to the cluster of its closest centroid. At each iteration, k new centroids are computed as the mean vector of all objects in each cluster, and the objects are again assigned to the cluster of its new closest centroid. The algorithm iterates until there is no further change in the cluster assignment or until a user-given maximum number of iterations t

is reached. One of the reasons for the popularity of the k-means algorithm is the adequate performance properties with a running time of $O(k \times N \times d \times t)$. A problem is that the type of clusters that can be found by k-means is limited to mixtures of Gaussian distributions. However, this problem can be solved by using the k-means clustering algorithm with a rather large value for k as a preclustering step, and then merge close-by or overlapping clusters in a postprocessing step. Experimental results have shown that this approach provides very competitive results, comparable to the best available clustering algorithms. For this reason, the proposed GPU implementation of the k-means algorithm aims to be capable of running efficiently for large values of k.

5.4.2 GPU implementation of k-means

The *depth test* (see Section 5.2.2) is used to efficiently implement the k-means algorithm on the GPU. For a better understanding, we first describe the algorithm for up to four-dimensional vectors, and then we explain the extension to arbitrary dimensional vectors. The first step of the GPU algorithm is to encode the data into textures in an appropriate way. For this algorithm, all coordinate values of each vector are stored in a single texel. This allows one to run the algorithm on vectors with up to four dimensions. As a result, the dataset is stored in a single texture with each texel representing a vector of the database. Figure 5.7 illustrates how the data is organized into the texture when using 4-D vectors. $V_{i,j}$ represents the j^{th} coordinate of the i^{th} vector. In the example, the original data is encoded into a 2×3 pixels image.

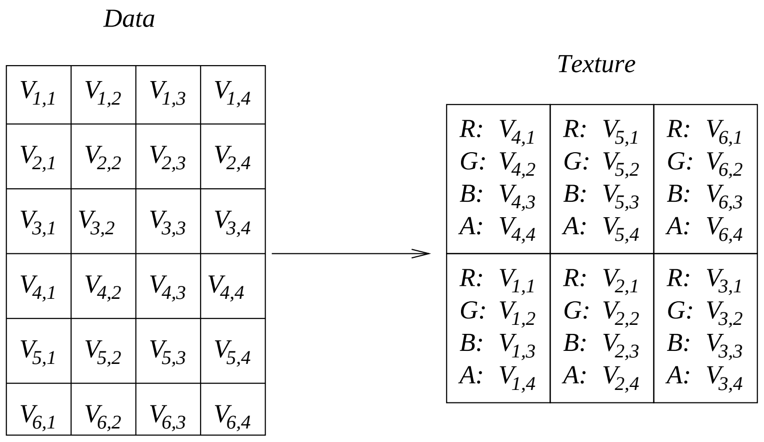
Data
$V_{1,1}$ $V_{1,2}$ $V_{1,3}$ $V_{1,4}$
$V_{2,1}$ $V_{2,2}$ $V_{2,3}$ $V_{2,4}$
$V_{3,1}$ $V_{3,2}$ $V_{3,3}$ $V_{3,4}$
$V_{4,1}$ $V_{4,2}$ $V_{4,3}$ $V_{4,4}$
$V_{5,1}$ $V_{5,2}$ $V_{5,3}$ $V_{5,4}$
$V_{6,1}$ $V_{6,2}$ $V_{6,3}$ $V_{6,4}$
Texture
R: $V_{4,1}$ G: $V_{4,2}$ B: $V_{4,3}$ A: $V_{4,4}$
R: $V_{5,1}$ G: $V_{5,2}$ B: $V_{5,3}$ A: $V_{5,4}$
R: $V_{6,1}$ G: $V_{6,2}$ B: $V_{6,3}$ A: $V_{6,4}$
R: $V_{1,1}$ G: $V_{1,2}$ B: $V_{1,3}$ A: $V_{1,4}$
R: $V_{2,1}$ G: $V_{2,2}$ B: $V_{2,3}$ A: $V_{2,4}$
R: $V_{3,1}$ G: $V_{3,2}$ B: $V_{3,3}$ A: $V_{3,4}$

Figure 5.7: Data organization for the k-means algorithm

During each iteration, the algorithm performs k passes, one for each centroid. Each pass computes the distances between a centroid and all of the vectors of the database, and stores these values in the depth buffer if they are smaller than the previously computed distances. This task is accomplished by the depth test, automatically executed when the *depth* variable is assigned. Only after a positive depth test (i.e., when the distance is smaller than the already stored value), the fragment program proceeds and the respective cluster identifier is stored. After the k passes, the respective cluster identifier is read back for each vector.

Algorithm 5.4 shows the code of the fragment program. As can be seen, the final code is very compact. The hardware optimized *dot* function, which returns the dot product of two vectors, is used to efficiently compute the sum of the square difference of coordinates between a vector and a centroid (line 11). The depth value needs to be divided by *dim* to normalize it in the depth range $[0.0, 1.0]$. The fragment program stores the cluster identifier in the four color channels for efficiency reasons (line 13).

Algorithm 5.4: Fragment program for k-means that uses the depth test

```
1  void k-means(
2  float2 coords : TEXCOORD0,
3  uniform samplerRECT tex : TEXUNIT0,
4  uniform float4 centroid,
5  uniform float centroidID,
6  uniform float dim,
7  out float4 color : COLOR,
8  out float depth : DEPTH) {
9    // Compute the difference of coords. between object and centroid
10   float4 fv=texRECT(tex, coords)-centroid;
11   // Compute square of distance and apply depth test
12   depth=dot(fv,fv)/dim;
13   color=centroidID.xxxx; } // id saved only if depth test passed
```

The only task of the algorithm that is performed on the CPU is the computation of the new centroids at each iteration. Implementing this part of the algorithm on the GPU would be rather difficult, because of the mentioned hardware restrictions (no communication between parallel running fragment program instances).

An extension of this algorithm for vectors with more than four coordinates can be carried out as follows: Each vector is split into blocks of four dimensions, and the Euclidean distance is computed in the same way as the

Manhattan distance is computed for the NN search algorithm. Only in the last pass, the depth test is used to obtain the closest centroid to the corresponding vector.

An alternative implementation of this GPU algorithm uses an auxiliary texture to store intermediate data, thus avoiding the usage of the depth test. An advantage of this alternative implementation is that the data does not need to be normalized to the range $[0.0, 1.0]$. However, the experimental results will show that this implementation is slower than the one using the depth test.

Algorithm 5.5: Fragment program for k-means without using the depth test

```
void k-means2(
float2 coords : TEXCOORD0,
uniform samplerRECT tex : TEXUNIT0,
uniform samplerRECT texR : TEXUNIT1,
uniform float4 centroid,
uniform float centroidID,
out float4 color : COLOR) {
  float4 vec=texRECT(tex, coords)-centroid;
  float4 fv=texRECT(texR, coords);
  float dist=dot(vec,vec);
  if (dist < fv.y) {color=float4(centroidID, dist, 0.0, 0.0);}
  else {color=fv;} }
```

5.5 Experimental evaluation

In this section, we present an experimental evaluation of the proposed GPU algorithms. We compare their efficiency with CPU implementations of the same algorithms.

5.5.1 Experimental framework

The graphics card used to perform the experiments is an NVIDIA GeForce 6800 Ultra graphics card, with 256 MBytes of video memory. The CPU is a Pentium IV 3.0 GHz. To compare the actual running time between conventional PCs and the CPU implementation, the database transfer time is not included in our performance measurement diagrams. Instead, we used data sizes which completely fit the graphics card memory. This approach was already used in previous studies [Govindaraju et al., 2004]. We did measure query upload time, computation time, and texture download time (the databases are uploaded only once into the graphics memory, thus this upload time is amortized over the queries). This constitutes a fair measurement to compare raw performance between CPU and GPU implementations of the same algorithm.

The CPU algorithms were implemented in C++ and compiled with the best available C++ compiler, the Intel C++ Compiler v8. All optimization flags were activated at compilation time to produce SSE2 enhanced CPU code. To compile the GPU fragment programs, we used the Cg compiler version 1.3.

5.5.2 Nearest neighbor algorithm

For the first experiment, databases of 262,144 vectors were used, with varying dimensions ranging from 16 to 256. All the coordinates values were random values uniformly distributed in the range $[0.0, 1.0]$. For each dimension, 1,000 random query vectors were created. Figure 5.8 shows the results of the comparison between the CPU and the GPU implementations. The GPU implementation clearly outperforms the CPU algorithm. With 256 dimensions, a speed-up factor of about 7x is observed, and on average the observed speed-up factor is about 6.4x. The GPU algorithm also scales well when using different database sizes. If the data did not fit into one texture (textures have a limited maximum size), they were partitioned into blocks of about 1 million objects and the algorithm run on each block iteratively. Figure 5.9 shows the results for 1 to 7.5 million vectors. In this case, the GPU algorithm was also several times faster than the CPU algorithm.

The GPU algorithm was also tested using real-world databases. The first database was the *Forest CoverType database* (*UCI-KDD-A*) which contains data about different forest cover types obtained by the U.S. Forest Service. Each observation is composed of 54 attributes, and the database consists of about 250,000 observations. The second database was the *Corel image features database* (*UCI-KDD-B*), which contains features of about 65,000 images extracted from a Corel image collection. The features are based on color histograms of 32-D. Both sets *UCI-KDD-A* and *UCI-KDD-B* are available at the *UCI KDD Archive* [Hettich and Bay, 1999]. The third dataset were feature vectors computed from a 3D CAD database. The database consists of about 16,000 CAD models, and the feature vectors (512-D) were computed using the spherical harmonics descriptor [Funkhouser et al., 2003]. For each dataset, 1,000 random objects were selected as queries for the NN search. Figure 5.10 shows the results for the three real-world databases. The GPU

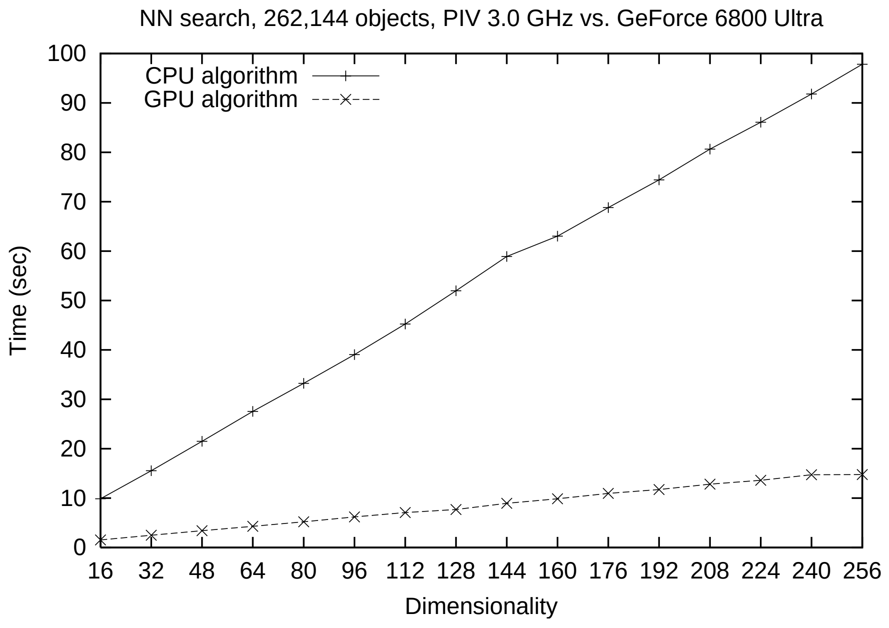

Figure 5.8: Experimental results for the nearest-neighbor algorithm varying the dimensionality of the space

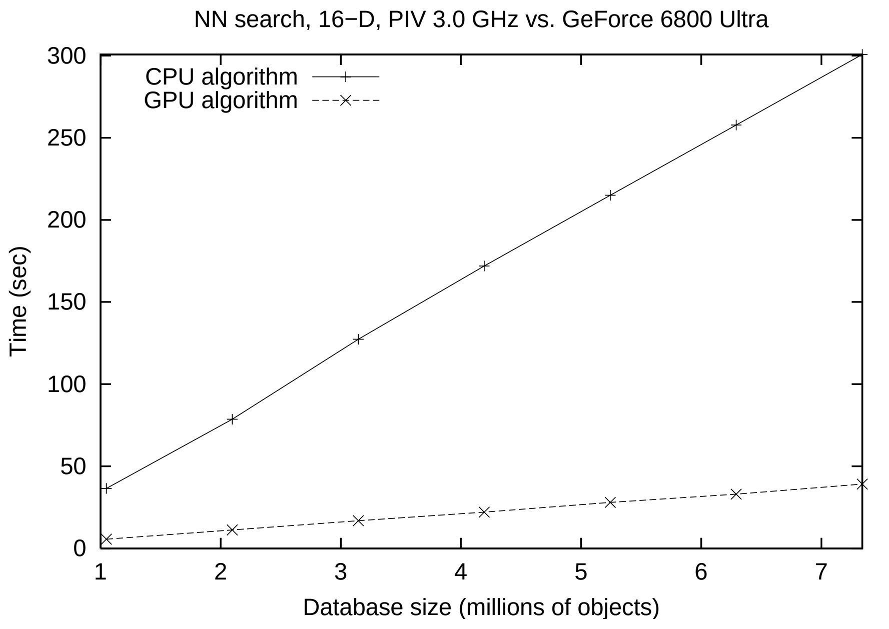

Figure 5.9: Experimental results for the nearest-neighbor algorithm varying the database size

algorithm shows improvement factors of 6.4x, 4.5x, and 4.2x respectively over the CPU algorithm.

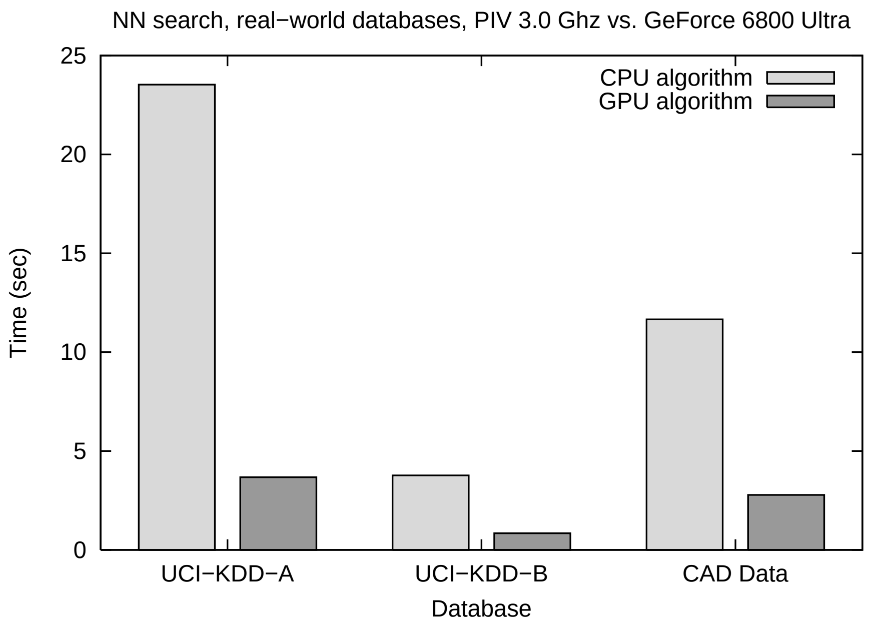

Figure 5.10: Experimental results for the nearest-neighbor algorithm with real-world datasets

5.5.3 Clustering algorithm

The GPU implementation of the k-means algorithm was evaluated using three different databases. High k values in the range $[100, 1000]$ were used, and the first k centroids were selected at random from the database. The first database consisted of 1,048,576 4-D random vectors (coordinate values between $[0.0, 1.0]$). For this database, 20 iterations were sufficient to obtain a stable clustering. Figure 5.11 shows the experimental results. The GPU algorithm using the depth test is on average an order of magnitude faster than the CPU algorithm, and it is on average 26% faster than the GPU algorithm that does not use the depth test.

Below, only the results for the GPU algorithm that uses the depth test will be shown.

The second database consisted of 1,048,576 4-D vectors with Gaussian distributions: It had 37 clusters with their centers being randomly selected. The variance of each cluster was also randomly selected between 0.0001 and 0.001. All vector coordinates were in the range of $[0.0, 1.0]$. 20 iterations were

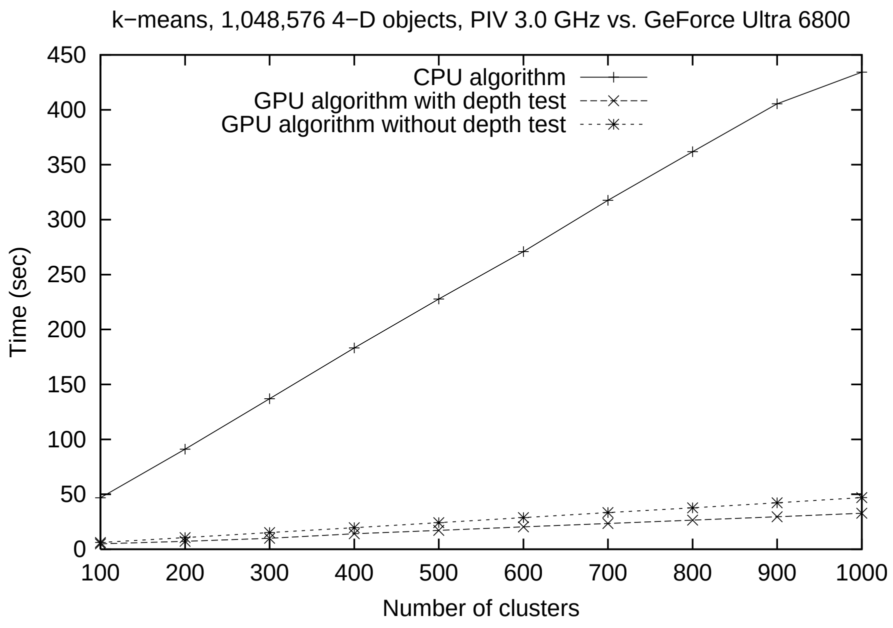

Figure 5.11: Experimental results for k-means algorithm with uniformly distributed data

needed to get stable clustering results. Figure 5.12 shows the experimental results. The speed-up factor of the GPU algorithm over the CPU is on average 12x.

The third database is a U.S. Census 2000 dataset, which consists of about 230,000 4-D vectors with the geographic location, median household income, and interest dividends income of citizens from New York. The vector's coordinates were normalized to the range $[0.0, 1.0]$. 50 iterations were required to obtain a stable clustering. Figure 5.13 shows the experimental results. The speed-up factor of the GPU algorithm over the CPU algorithm is on average about 15x.

The scaling of the GPU algorithm to large datasets was also tested. Figure 5.14 shows how the GPU algorithm scales for database sizes between 1 and 4.3 million data points. The data is split into blocks of about 1 million objects, and each block is processed in one rendering pass of the GPU. The results show that, for k-means, the GPU is at least one order of magnitude faster than the CPU.

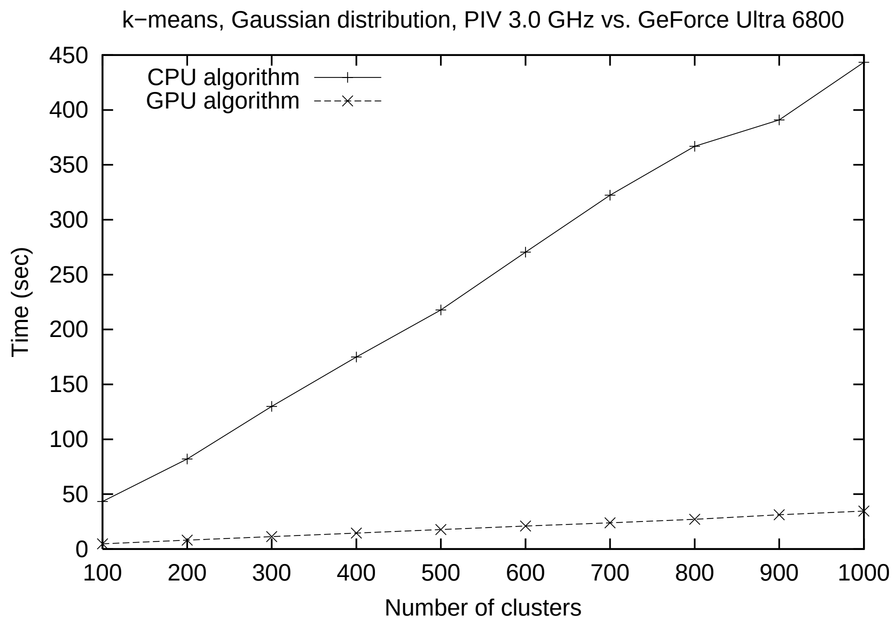

Figure 5.12: Experimental results for k-means algorithm, data with Gaussian distribution

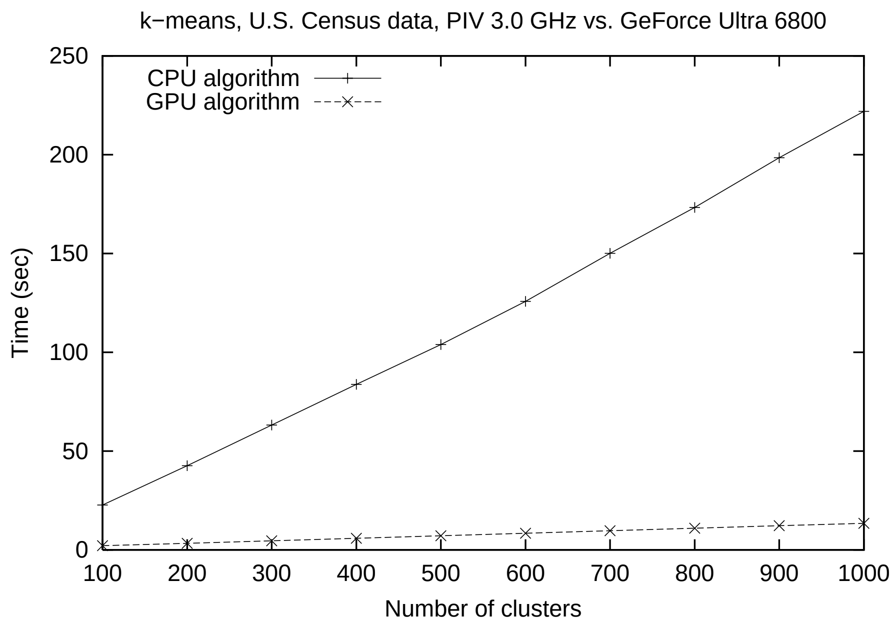

Figure 5.13: Experimental results for k-means algorithm with U.S. Census data

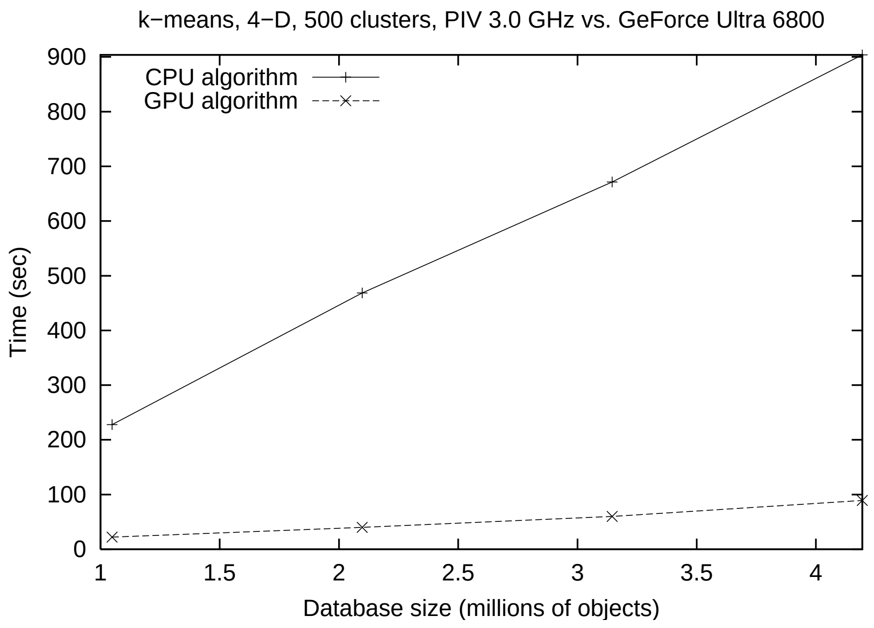

Figure 5.14: Experimental results for k-means algorithm varying the database size

5.6 Analysis of the performance results

The experimental results presented in the last section showed that a state-of-the-art graphics card easily outperforms the CPU. The main reasons behind this fact are the parallelism of the fragment processors and the highly optimized vector operations inside the GPU. Both factors contribute to the much faster execution of the nearest neighbor and k-means algorithms, compared with their CPU implementations.

What kind of improvements can one expect for the future? Figure 5.15 shows the floating-point performance of GPUs from NVIDIA and ATI compared with the performance of CPUs from Intel. The figure shows that GPU performance has increased considerably over the last years, and at a much faster rate than CPUs (even considering dual core CPUs, as shown in the figure). As GPUs are being used nowadays for general purpose computations in many application domains [Owens et al., 2005], we expect that the trend shown in Figure 5.15 will continue for the next years.

To assess how much the performance may improve between different generations of graphics cards, with respect to the GPU algorithms proposed in this chapter, we performed a comparison between the GeForce 6800 Ultra card, used in the experiments, with one of the latest cards from the last

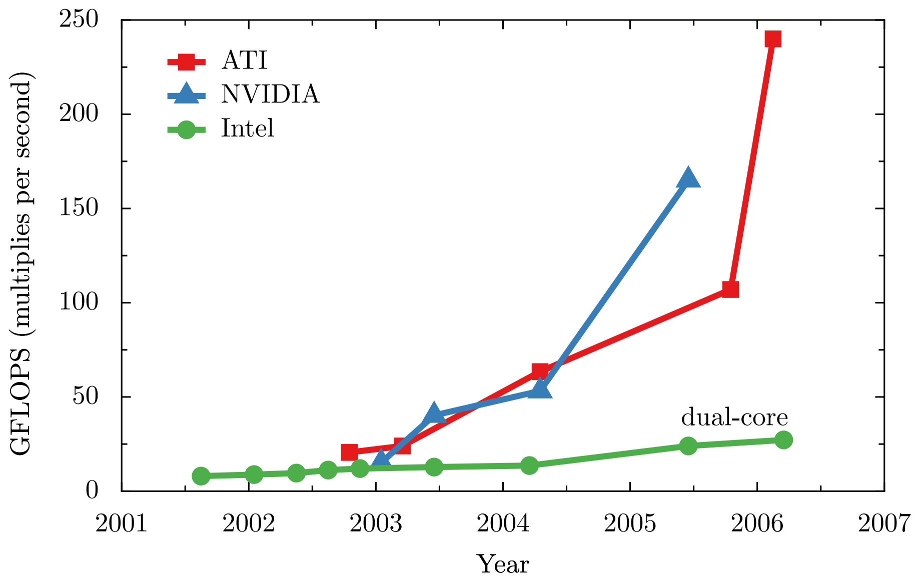

Figure 5.15: Performance of GPUs compared with CPUs over the last 5 years [Owens et al., 2005]. Figure courtesy of John D. Owens, University of California, Davis.

generation, namely the GeForce 5900 FX.

One important metric to compare two graphic cards is the number of pixels that the GPU can process per clock cycle. The GeForce 5900 FX can process up to 8 pixels at the same time, while the GeForce 6800 Ultra can process up to 16 pixels. Another important metric is the *memory bandwidth* of the graphics card. Figure 5.16 shows that the memory bandwidth of the GeForce 6800 Ultra doubles the memory bandwidth of the GeForce 5900 FX as long as the data fits into the graphics card memory (note that the figure shows the *inverse* of the memory bandwidth, thus a lower value is better). Figure 5.16 further shows that the memory bandwidth of the GeForce 6800 Ultra quickly decreases if the texture data is larger than the video memory. This occurs because the data must be re-loaded onto the graphics card during each pass. This problem will be alleviated in the near future in two ways. Firstly, the introduction of the PCI Express bus will increase the transfer memory bandwidth from the host to graphics card (it is two times wider than the AGP 8x bus). Secondly, there are already new graphic cards with 512 MBytes of video memory, and in the near future graphic cards with a minimum of 1 GByte video memory will be available. It is also interesting to note that by using textures with 256x256 resolution, we achieve the best performance results. It seems that the hardware is optimized to this texture size, mainly because today's video games use this texture resolution.

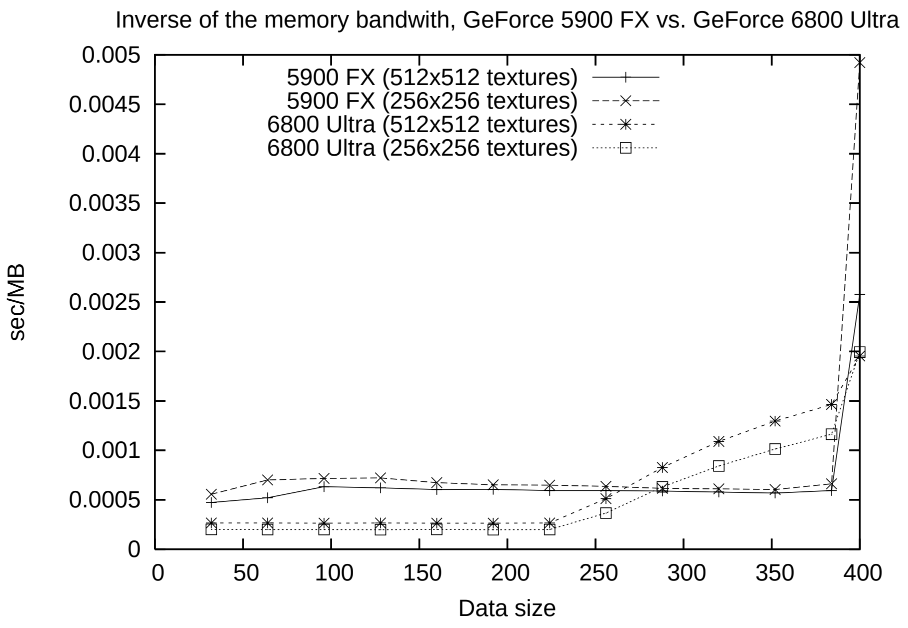

Figure 5.16: Memory bandwidth comparison between two different graphic cards

Thus, with regard to hardware, the GeForce 6800 Ultra should be at least twice as fast as the GeForce FX 5900. To validate this hypothesis, the proposed GPU algorithms were run on both graphic cards to compare their performance. Figure 5.17 shows the results for the NN search algorithm. It is obvious that the GeForce 6800 Ultra is twice as fast as the GeForce FX 5900 using this algorithm.

Figure 5.18 shows the results for the k-means algorithm. In this case, the GeForce 6800 Ultra is 3 times faster than the GeForce FX 5900. An explanation for this result is that while the GeForce FX 5900 can process up to 8 pixels per clock cycle, it only contains 4 "real" (hardware coded) fragment processors, but it can process two textures at the same time. But the GeForce 6800 Ultra has 16 "real" fragment processors. For this reason, the k-means algorithm cannot take full advantage of the capabilities of the GeForce FX 5900, and thus the speed-up is higher on the GeForce 6800 Ultra. For the next generation of graphic cards, a similar speed-up factor compared with the actual state-of-the-art is expected.

Another possibility to gain more performance is using the CPU while the GPU is processing data, because during this time the CPU is idle. A load-balancing algorithm between CPU and GPU will provide additional improvements. Current graphic cards drivers do not allow such a load balancing.

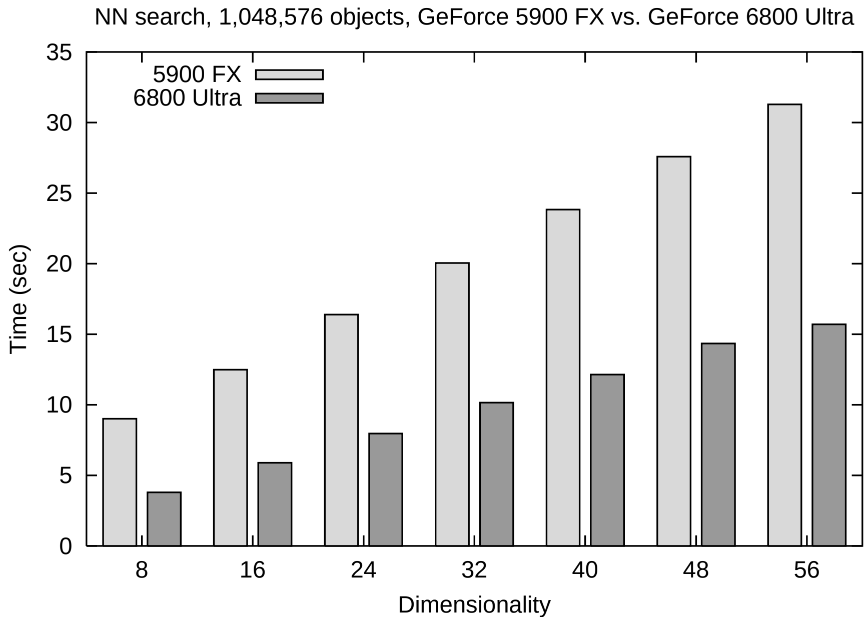

Figure 5.17: Comparison between two different graphic cards with the nearest neighbor algorithm

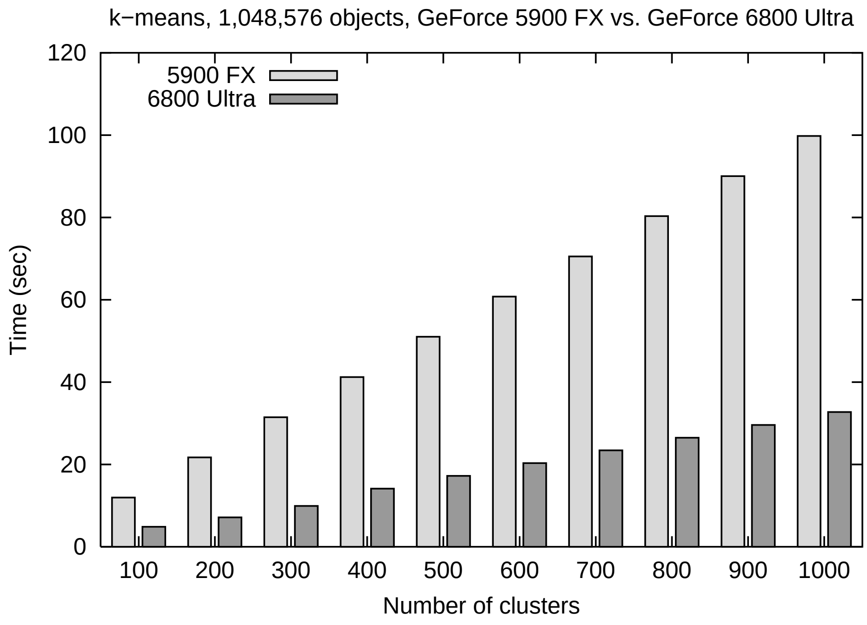

Figure 5.18: Comparison between two different graphic cards with the k-means algorithm

One way to overcome this problem would be to run another thread on the CPU, in order to use the idle time for performing some of the computations.

5.7 Conclusions

In this chapter, we presented hardware accelerated algorithms for nearest neighbor search in high-dimensional vector spaces and for k-means clustering. Firstly, an introduction into GPU programming was given, and it illustrated that data encoding into texture memory is the key to efficient and compact GPU programming. Several experimental results using synthetic and real-world datasets showed that GPU implementations provide an order of magnitude better performance than corresponding CPU implementations with linear scalability in dimensionality and in database size. The good price-performance ratio and fast improvements of GPUs guarantee significant additional performance improvements in the near future.

The coding of the input data as geometric primitives is the most complex task for implementing GPU-based algorithms for database operations. For the studied applications, two ways for mapping vectors into texture data were described. These data mappings are general and do not have restrictions on the dimensionality of the data. Once this problem was solved, it was shown that relatively simple fragment programs can efficiently process the data and return the nearest neighbor of a dataset or perform clustering on a set of points. Also, the integration of GPU-based algorithms in commercial DBMS has been proved to be feasible [Bandi et al., 2004]. The proposed GPU algorithms may be the starting point for a new research area focusing on using GPUs for improving database performance.

Chapter 6

Conclusions

In this thesis, we presented several techniques that aim to improve the effectiveness and the efficiency of similarity search in multimedia databases. As the amount of multimedia data is rapidly growing, the development of efficient and effective techniques for searching and indexing multimedia databases has become indispensable. Therefore, we believe that this thesis makes a valuable contribution to this research area.

The main contributions of this thesis can be summarized as follows:

- We have presented an experimental evaluation of the effectiveness of feature vectors for 3D model retrieval. We implemented and compared 16 different descriptors. We concluded that there are a few descriptors that have good effectiveness on average, but that there is no feature vector that outperforms all others in all possible situations: The best feature vector to use depends highly on the query object.

- We have found that combinations of feature vector may significantly improve the effectiveness of similarity search. We introduced the purity and entropy impurity methods, which aim to assess a priori the goodness of a feature vector given a query object. Both methods use a small training dataset, and can be used to implement dynamically weighted combinations of feature vectors. We obtained the best experimental evaluation using these dynamic combinations.

- We have presented several pivot selection techniques for pivot-based indices. We proposed an efficiency criterion based on distance distributions for selecting good sets of pivots, and tested several optimization techniques that uses the proposed efficiency criterion. We tested the proposed techniques with several synthetic and real datasets, showing that our techniques are better than previously proposed heuristics.

- We have described an improved version of the best-first k-NN search. Our proposed algorithm reduces the memory requirements of the original algorithm by using distance estimators. These estimators can be used to define a better distance upper bound that can be used for early discarding zones in the space, which cannot have any relevant object for the query.

- We have proposed a pivot-based index structure that allows us to index dynamically weighted combination of feature vectors. We provided a NN-search algorithm that uses the proposed index structure, and showed that it is several times faster than the naïve search algorithm.

- We have introduced an approach to index the most frequently used combinations of feature vectors by using set of indices. The resulting optimization problem can be modeled as a binary linear program, which can be used to find the optimal solution. We also propose some heuristics that quickly find good sets of indices.

- We have presented hardware accelerated algorithms for NN-search and a clustering algorithm. Both implementations use the graphics processor unit (GPU) as a co-processor for both applications. We developed data encoding techniques to map the original data into graphics primitives, which can be processed by the GPU. The experimental evaluation showed that our proposed GPU algorithms are an order of magnitude faster than the CPU implementations of the same algorithms.

Each of the techniques described and analyzed in this thesis intends to improve the state-of-the-art of different aspects of similarity search in multimedia databases. As we already said, the effectiveness aspects of the search are as important as its efficiency aspects, thus it is fundamental to have methods that improve both aspects and can be used jointly. That is exactly what this thesis pursued: By using together all the proposed techniques (e.g., using entropy impurity to select the best feature vectors to perform a similarity query, supported by a pivot-based data structure selected from an *iSet* and whose pivots were chosen with the incremental selection technique, and using the improved nearest neighbor algorithm to optimize the space used while performing the query), we are effectively improving the efficiency of the similarity search as well as the quality of the retrieved result.

We would like to emphasize that the different contributions presented in this thesis can be used jointly, providing an overall improvement to the area of similarity search in multimedia databases. We showed that the effectiveness of a similarity query in a multimedia database may be improved by using dynamic combinations of feature vectors. Later, we showed how to index these dynamic combinations of feature vectors using a pivot-based index, and showed how to improve the efficiency of this index by using our proposed pivot selection techniques. If the search system selects some of the feature vectors to perform the dynamic combination, we showed how to use the available space for building indices to minimize the expected search cost (by constructing the optimal set of indices or *iSet*). Finally, the similarity query may be further optimized (in terms of space cost) by using our improved k-NN search algorithm.

Here we outline possible trends and improvements that can be performed based on the results presented in this thesis.

- With respect to the effectiveness of similarity search:
  - It would be interesting to define methods for computing dynamically weighted combinations of feature vectors that do not rely on a training dataset. For example, self-organizing maps could be used for the analysis of feature vectors, to determine their suitability for a given query object.
  - A question that remains open is the optimal size of the training set. Unfortunately, ground-truth sets for multimedia databases are not common and are in general very small. Indeed, a great amount of work was invested to obtain our own classified set of 3D objects, because the classification was done manually. We observed experimentally that by using only a half of our classified set of 3D objects we could still improve significantly the effectiveness of the search, but our 3D database is too small to infer conclusions from this result. Further experiments with larger databases are needed to answer this question.
  - Feature-engineering may be used to further improve the effectiveness of single feature vectors. Mathematical as well as visual analysis tools could be useful for this purpose.
  - The definition and effective implementation of partial similarity search notions among multimedia objects remains a big challenge. This problem is far more complex than the similarity search problem studied in this thesis, because in partial similarity only a fraction of the multimedia object is considered for the match. Even the concept of “match” in this context must be properly defined: We may want to look for similar parts or for complementary parts (e.g., protein docking problem).

- With respect to the efficiency of similarity search:
  - We plan to continue exploring the trade-off between the index size and the storage requirements for the active page list in our improved k-NN search algorithm. Further improvements in the average length of Q may be obtained if one had more structural information about the balls, at the cost of storing more information on each index node.
  - We made the observation that our improved k-NN algorithm is also relevant in the vector space case. Further experimental evaluations are needed to assess the real gains that could be achieved in this case by our algorithm compared with the standard techniques.
  - The main drawback of the proposed pivot-based index for combinations of feature vectors is that it is a main-memory index. It would be interesting and very useful to have a secondary-storage implementation of this index structure. For this case, one would like not only to minimize the number of distance computations, but also the number of disk accesses performed during the search.
  - Another open problem is the adaptation of hierarchical access methods for indexing combinations of feature vectors. One possibility is to adapt a tree-like structure (e.g., the M-tree) to support queries that uses combinations of feature vectors. In this case, the index structure must provide a correct partitioning of the space independent of the set of weights used while providing at the same time a good discriminative power.
  - We plan to study the complexity of finding the optimal *iSet*. We have shown that the problem is NPO, but its complexity is still unknown. We presume that finding the optimal *iSet* is NPO-Complete, but we still need to prove it formally. We also want to improve the proposed algorithms, by analyzing their weaknesses, and to improve the presented lower bound for the optimal solution.
  - Also related with the *iSet* is the definition of the *Space* and *Search* cost functions. In some cases, these functions are not difficult to define (e.g., in the case of VA-File). However, it is not clear a priori how to define these functions for other index structures. One possible solution is to compute empirically the cost values using different database sizes and dimensionalities, and then extrapolate analytical cost functions from these obtained values.

- With respect to hardware accelerated algorithms for similarity search in multimedia databases:
  - We proposed a sequential scan of the database on the GPU to implement a NN-search. It would be interesting to implement GPU accelerated index structures and their associated search algorithms, and to compare them against the brute-force approach. The pivot-based index is a good candidate to start with, because the data structure is a matrix which can be directly mapped to a texture. However, the implementation of the search algorithm could be more complicated.
  - It would also be interesting to implement a complete similarity search engine that runs entirely on the GPU. For example, in the case of 3D model retrieval this would mean implementing the transformation function (including a normalization processing, feature extraction, post-processing, etc.) and the similarity search (which is already partly done). For static databases that fit on the graphics card memory, we expect that such an implementation would be several times faster than the existing CPU-based solutions.