Approximate similarity search in metric spaces

Dissertation

zur Erlangung des Grades eines
Doktors der Naturwissenschaften
der Universität Dortmund
am Fachbereich Informatik

von

Giuseppe Amato

Dortmund
2002

Tag der mündliche Prüfung:	14.06.2002
Dekan:	i. V. Prof. Dr. Bernhard Steffen
Gutachter:	Prof. Dr. Norbert Fuhr
	Prof. Ing. Pavel Zezula
	Prof. Dr. Joachim Biskup

To my wife Pina and my sons Niccolò and Giacomo

Table of Contents

Abstract

There is an urgent need to improve the efficiency of similarity queries. For this reason, this thesis investigates approximate similarity search in the environment of metric spaces. Four different approximation techniques are proposed, each of which obtain high performance at the price of tolerable imprecision in the results. Measures are defined to quantify the improvement of performance obtained and the quality of approximations. The proposed techniques were tested on various synthetic and real-life files. The results of the experiments confirm the hypothesis that high quality approximate similarity search can be performed at a much lower cost than exact similarity search. The approaches that we propose provide an improvement of efficiency of up to two orders of magnitude, guaranteeing a good quality of the approximation.

The most promising of the proposed techniques exploits the measurement of the proximity of ball regions in metric spaces. The proximity of two ball regions is defined as the probability that data objects are contained in their intersection. This probability can be easily obtained in vector spaces but is very difficult to measure in generic metric spaces, where only distance distribution is available and data distribution cannot be used. Alternative techniques, which can be used to estimate such probability in metric spaces, are thus also proposed, discussed, and validated in the thesis.

Acknowledgements

It would have been very difficult to produce this thesis without the help and support of a number of people. In particular, I should like to express my gratitude to the following colleagues, friends and members of my family.

Above all, I am particularly grateful to Norbert Fuhr, who trusted in my research activity and encouraged me. He gave me the possibility of pursuing my PhD at his university and accepted to review my thesis.

Of course I must also thank, Costantino Thanos, the head of my research group, who offered me the opportunity to work on this thesis. I should like to acknowledge my appreciation to him for believing in me and encouraging me in the difficult world of scientific research.

Then, Pavel Zezula and Pasquale Savino for their invaluable contributions to the development of the research work described in this thesis. Their enthusiasm, clarity and above all willingness to listen and to provide suggestions have been of great importance to me. I must especially thank Pavel Zezula for also have accepted to be my reviewer.

Joachim Biskup for his patience in reading and reviewing this thesis, and his valuable suggestions. Piero Maestrini, the director of my research institute, for encouraging me in the development of this research activity.

I should like to thank a number of colleagues for their support. Fausto Rabitti and Claudio Gennaro for their willingness to discuss and exchange opinions on issues related to the topics of my thesis. Donatella Castelli and Paola Venerosi for their continuous support and encouragement. Carol Peters for her encouragement and precious suggestions regarding the text. Umberto Straccia for his patience and courage in sharing an office with me during this time. Stefano Chessa for his great friendship and openness. We took our first steps in the research world together and we have shared several important lessons in life.

My parents for always supporting and believing in me. My brother Sandro for teaching me to run rather than walk in order to achieve more. My sister-in-law Imma with whom I have had many important and motivating discussions on the meaning and difficulties of life.

And finally Pina for her limitless patience in assisting me continuously and sustaining me in the production of this thesis. During this period, she has not only helped me by creating a warm and happy home life but most important of all, she has given me two wonderful sons, Niccolò and Giacomo, thus filling our lives with joy.

List of Symbols

Symbol	Description
$\mathcal{B}$	Generic ball region.
$\mathcal{B}(O, r)$	Ball region with center O and radius r.
$cost(\mathbf{oper})$	Cost of executing search operation **oper**.
d or $d(O_1, O_2)$	Distance function.
dim	Number of dimensions in a vector space.
$d_{it}^{O_q,k}(iter)$	Discrete function returning the distance of the current k-th object from the query object O_q at the iteration $iter$ of the k nearest neighbors search algorithm.
d_m	Maximum distance in the distance bounded metric space.
d_{xy}	Distance between objects O_x and O_y.
$\mathcal{D}$	Domain of the metric space.
$\mathcal{DS}$	Data set containing objects of the domain $\mathcal{D}$.
D_{XY}	Continuous random variable corresponding to the distance $d(\mathbf{O_x}, \mathbf{O_y})$, with $\mathbf{O_x}$ and $\mathbf{O_y}$ random objects of $\mathcal{D}$.
EP	Error on the position, used to determine the accuracy of approximate nearest neighbors algorithms.

Symbol	Description
ϵ	Relative error on distances or upper bound of the relative error on distances.
$\epsilon(r_x, r_y, d_{xy})$	Absolute error of $X_{d_{xy}}^{appr}(r_x, r_y)$ with respect to $X_{d_{xy}}^{actual}(r_x, r_y)$.
$\epsilon'_{\mu}(d_{xy})$	Average of $\epsilon(r_x, r_y, d_{xy})$ varying r_x and r_y.
$\epsilon''_{\mu}(r_x, r_y)$	Average of $\epsilon(r_x, r_y, d_{xy})$ varying d_{xy}.
$\epsilon'_{\sigma}(d_{xy})$	Variance of $\epsilon(r_x, r_y, d_{xy})$ varying r_x and r_y.
$f(x)$	Overall distance density.
$f_O(x)$	Density of distances with respect to object O.
$f_X(x), f_Y(y)$	Density functions of continuous random variables X and Y.
$f_{XY}(x, y)$	Joint density function of continuous random variables X and Y.
$f_{XY\|D_{XY}}(x, y, d_{xy})$	Joint conditional density function of continuous random variables X and Y given D_{XY}.
$F(x)$	Overall distance distribution.
$F_O(x)$	Distribution of distances with respect to object O.
IE	Improvement of efficiency, used to determine the performance of approximate search algorithms.
k	Number of objects retrieved in a nearest neighbors query.
$\mathcal{M}$	Metric space. $\mathcal{M} = (\mathcal{D}, d)$, such that distance function d is a metric.
$\mathbf{nearest}(O_q, k)$	Set of objects returned by the nearest neighbors search algorithm.
$\mathbf{nearest}^{x_p, x_s}(O_q, k)$	Set of objects returned by the approximate nearest neighbors search algorithm with approximation parameters x_p and x_s.
N or N_i	node of a tree.

Symbol	Description
NE	Number of exact results, used to determine the accuracy of approximate range search algorithms.
O, O_x, O_y, O_z, O_i, O_j	Objects of the metric space or centers of ball regions.
O_q	Query object.
oper	Exact similarity search operation. It can be either **range**(O_q, r_q) or **nearest**(O_q, k).
oper^A	Approximate version of **oper**.
p_i	Pointer to a record in an entry of a tree node.
Q, Q_1, Q_2, Q_3	Query regions.
r, r_x, r_y, r_i	Radii of ball regions.
range(O_q, r_q)	Set of objects returned by the range search algorithm.
range$^{x_p}(O_q, r_q)$	Set of objects returned by the approximate range search algorithm with approximation parameter x_p.
$reg_d(iter)$	Continuous function that approximates $d_{it}^{O_q,k}(iter)$, obtained by using linear regression.
r_q	Radius of the query region.
$\mathcal{R}$, $\mathcal{R}_i$	Region.
x_s	Parameter for the approximate stop condition.
x_p	Parameter for the approximate pruning condition.
X	Continuous random variable corresponding to the distance $d(\mathbf{O}, \mathbf{O_x})$, with $\mathbf{O}$ and $\mathbf{O_x}$ random objects of $\mathcal{D}$.

Symbol	**Description**
$X(\mathcal{B}(O_x, r_x), \mathcal{B}(O_x, r_y))$	Proximity of ball regions $\mathcal{B}(O_x, r_x)$ and $\mathcal{B}(O_x, r_y)$
$X_{d_{xy}}(r_x, r_y)$	Overall proximity of any pairs of regions having radii r_x and r_y, and whose distance between centers is d_{xy}.
$X_{d_{xy}}^{actual}(r_x, r_y)$	Overall proximity computed using the formal definition.
$X_{d_{xy}}^{appr}(r_x, r_y)$	Overall proximity computed using one of the proposed heuristics.
$X^{trivial}(\mathcal{B}(O_x, r_x), \mathcal{B}(O_x, r_y))$	Proximity of ball regions $\mathcal{B}(O_x, r_x)$ and $\mathcal{B}(O_x, r_y)$ computed using a trivial technique.
Y	Continuous random variable corresponding to the distance $d(\mathbf{O}, \mathbf{O_y})$, with $\mathbf{O}$ and $\mathbf{O_Y}$ random objects of $\mathcal{D}$.
$\|exp\|$	Absolute value of expression exp.
$\|\|v\|\|$	Euclidean norm of vector v.
$\#S$	Cardinality of set S.

List of Figures

(page left intentionally blank)

Chapter 1

Introduction

The performance capabilities of computer systems are rapidly increasing. Faster CPUs, larger secondary storage, and larger network bandwidth make it possible to handle more complex data types. Nowadays, people buy computers attracted by the promise of using audio, image, video, and 3D data. In professional and scientific areas such as medicine, computational biology, signal processing, or finance, new large data types in the form, for example, of time series, huge sequences of symbols, or raw streams of data are processed, analyzed and contrasted.

A very important issue on any kind of data management is searching. Traditional database systems efficiently search for structured records by using the *exact match* paradigm. However, the new data types, cannot be effectively represented as structured records. In these cases, the *similarity search* paradigm [JMM95] is used instead of exact match. Similarity searching consists in retrieving data that are similar to a given query. The measure of similarity is specifically defined with respect to the target application.

Techniques for improving performance of exact match search cannot be used for similarity search, and even if other alternative approaches were proposed, current

techniques are far from satisfactory [WSB98, BGRS99]. Therefore, the issue of *approximate similarity search* [AM95, AMN+98, CP00, FTAA00, FTAA01, IM98, PL99, PAL99, ZSAR98, Kle97, Cha97, KOR99, PMD01] has recently become an important research topic. Approximate similarity search provides an improvement in similarity search performance at the price of some imprecision in the results.

In this thesis, we have investigated techniques for approximate similarity search when data are represented in generic metric spaces. The techniques proposed offer an improvement of efficiency, with respect to exact similarity search techniques, up to two orders of magnitude, guaranteeing a high degree of accuracy.

In particular, one of the approximate similarity search techniques that is proposed relies on the measurement of the so-called *proximity of regions*. This measurement provides an estimation of the amount of objects shared by two regions of the space. In generic metric spaces, proximity of regions is not easily computed, given the lack of a coordinate system. We have also developed some techniques for computing the proximity of ball regions defined in metric spaces. The techniques proposed are efficient and their estimation of the proximity is extremely accurate.

1.1 Similarity search

Traditional database systems maintain files consisting of sequences of homogeneous records. In these databases the search paradigm used is the *exact match search* paradigm. Even more sophisticated searches, where a range of possible values is specified, such as a numerical range or a string prefix, rely on the possibility of exact comparison among attribute values. The exact match search divides the whole set of records into a subset that contains all records that satisfy the query, and another

subset that contains the records that do not satisfy the query.

On the other hand, a *similarity search* query orders the whole set of managed data with respect to the similarity of the individual items to the query. Therefore, ideally, the result set returned by a similarity search contains all items contained in the database along with the degree of similarity of each item to the query.

In order to limit the size of the returned result set, two different techniques are typically considered: cutting off items whose similarity to the query is below a specified threshold, or specifying the maximum number of items to be retrieved. Correspondingly, there are two basic types of similarity queries: *range queries* and *nearest neighbors queries*. A range query is composed of a query object and a similarity threshold. All data whose similarity to the query object is greater than the threshold satisfy the query. A nearest neighbors query, on the other hand, is composed of a query object and a value k indicating the maximum number of elements to be retrieved. The k elements most similar to the query satisfy the query.

1.1.1 Similarity and distance functions

At the basis of the similarity search process lies the ability of measuring the similarity between two items. This is obtained by defining a *similarity function*. There is not a general, universal definition of this function, since it depends on the specific application. Similarity functions are typically defined by application experts depending on the aspect that interests them.

Note that in some applications a *distance function* can be easily and more intuitively defined than a similarity function. However, it is easy to obtain a similarity function given a distance function and vice versa. In fact, similarity functions and

distance functions have an exactly opposite behaviour: the smaller the distances the higher the similarity.

The similarity function is typically considered as a black box provided by the application expert. General techniques developed for similarity search should not be tied to a particular similarity function. However, the similarity functions might be required to satisfy some specific properties.

In the following we briefly introduce two typical frameworks used to handle similarity search.

1.1.2 Vector spaces

Several similarity search applications represent data as high dimensional vectors. As an example consider image databases. Typically, the similarity between images is assessed by comparing features extracted by them, instead of using the raw images themselves. For instance, a typical image feature is a color histogram. This can easily be represented as a vector, where the value contained in each dimension is the density of the color associated with the dimension as a consequence of the quantization of the color space [Smi97].

When data are represented in vector spaces, similarity can be assessed by measuring the distance between two vectors, using a function of the Minkowski family, such as for instance, the Euclidean distance.

1.1.3 Metric spaces

In certain applications requiring similarity search either data cannot be effectively represented as vectors, or distance (similarity) cannot be effectively defined in terms

of functions of the Minkowski family. Consider again, for instance, color histogram similarity. Minkowski functions compare values in each dimension (colors) independently of the others. However it has been shown that color histogram similarity can be better assessed by using the *quadratic form* distance [FSA+95, HSE+95, NBE+93], which does not belong to the Minkowski family.

In such cases data are typically represented in *metric spaces*. The advantage of metric spaces is that there are no specific requirements on the representation of data, while the only constraint for the distance function is that it should comply to the metric postulates: it should satisfy the symmetric, positivity, identity, and triangular inequality properties.

Note that, when distances are measured by using Minkowski functions, vector spaces are in fact specific metric spaces. Therefore, techniques developed for generic metric spaces can also be applied to them. However metric spaces are generally more difficult to handle than vector spaces. This is due to the fact that vector spaces provide more information, such as geometric properties and coordinate systems.

Given their generality, in this thesis, we suppose that data are represented in a metric space; our results are thus widely applicable.

1.2 Approximate similarity search

Though the problem of similarity search seems to be very well defined and several storage structures have been proposed – see for example the survey in [CNBYM01] for metric spaces and the numerous methods for the vector spaces surveyed in [GG98] – current technology cannot certainly be considered stable. In fact, in some situations, performance of existing access methods is worse than a sequential scan of the whole

data set [BKK97, WSB98, BGRS99].

Given this inefficiency problem, the notion of *approximate similarity search* [AM95, AMN+98, CP00, FTAA00, FTAA01, IM98, PL99, PAL99, ZSAR98, Kle97, Cha97, KOR99, PMD01] has emerged as important research issue. The basic idea behind the approximate similarity search is that queries are processed faster, at the price of some imprecision in search results.

In general, approaches to approximate similarity search are motivated by the following three observations. First, typically a good data partitioning of metric data sets is simply not possible, so resulting data regions have typically a high overlap, and many regions must be accessed to answer a single query. Second, similarity-based search processes are intrinsically iterative. In many cases, users redefine queries depending on the results of previous searches. In such cases, an efficient execution of elementary queries is of particular importance and users easily accept some imprecision, especially in the initial and intermediate search results, if much faster responses can be achieved. Third, introducing some controlled imprecision in the result of a similarity search query may not be noticed by users or will be accepted when the increase in performance obtained is high. Note that similarity is an intuitive and subjective measurement. People, for instance, judge the similarity between two images differently. However, when similarity search algorithms are designed, the similarity measure must be defined using a rigorous mathematical formula and intuitiveness and subjectivity is lost.

Approaches to approximate similarity search can be classified into two broad categories [FTAA01]:

(1) approaches that reduce the size of data objects;

(2) approaches that reduce the data set that needs to be examined.

Approaches of the first category are mainly based on techniques of dimensionality reduction, and assume that a few dimensions are enough to represent the most relevant information of data items.

Two classes of approaches can be distinguished in the second category:

(a) approaches based on heuristics that stop the search algorithms before natural termination, when the current result set is judged to be satisfactory;

(b) approaches based on heuristics that do not access regions judged as not to containing promising results.

The advantage of the second category of approximate algorithms is that the original data representation is used and the heuristics may be tuned by using specific approximation parameters at query time: the higher the performance, the lower the accuracy of the approximation. Approaches belonging to the first category, on the other hand, modify the original data set and no tuning can be typically performed at query time.

1.3 Contribution of this thesis

This thesis proposes *four new techniques for approximate similarity search*, starting from the general case that data are represented in a metric space.

In addition we have developed some *techniques to measure the proximity of ball regions in generic metric spaces* efficiently and accurately. This measurement gives

the amount of data shared by two regions. One of the algorithms proposed exploits the measurement of the proximity of ball regions in its approximation strategy.

In the following we list the main issues that will be developed in the thesis.

1.3.1 Approximate similarity search

All the methods proposed belong to the category of approximation algorithms that reduce the data set that needs to be examined (category 1). Two of the proposed techniques are based on heuristics of early terminations (class a). The other two are based on heuristics for discarding unpromising regions (class b).

A brief description of these heuristics follows:

First method The first technique discards regions, containing data, guaranteeing that the maximum relative error on distances introduced, when regions containing qualifying data are discarded, is smaller than a threshold defined by the user. This method can be used both for range and nearest neighbors queries.

Second method The second technique of approximation stops the search algorithm when the current result set belongs to a user-defined percentage of the most similar data in the whole database. This method can be used for nearest neighbors queries only.

Third method Similarity search algorithms are based on an iterative process where a current result set is improved at every iteration. The third approximation method stops the search algorithm when the improvement of the current result set slows down. This method can be used for nearest neighbors queries only.

Fourth method The fourth approximation method discards regions when it is judged that they do not contain qualifying objects. This estimation is obtained by measuring the proximity of ball regions. This corresponds to the amount of data shared by two ball regions in a metric space. This method can be used both for range and nearest neighbors queries.

The results were extremely promising. The improvement of efficiency offered by these techniques reaches some orders of magnitude in correspondence with high accuracy of the retrieved result sets. Our techniques are more general than other existing techniques for approximate similarity search. In fact, they can be used in metric spaces and some can be used both for range and nearest neighbors queries. Most of the existing technique are, on the other hand, typically limited to vector spaces and to nearest neighbors queries. Their performance is generally worse than that of our techniques. As far as we know, there are no other techniques that can be used for both range and nearest neighbor queries.

1.3.2 Proximity of ball regions in metric spaces

It is easy to infer that a large overlap between regions does not always imply that a large amount of data are shared by them. The amount of data contained in the intersections depends on the data distribution. Therefore, there may be regions with a large intersection and few data in common, but also regions with a small intersection and many data in common. This happens when the intersection covers a dense area of the data space.

In this thesis, the measurement of the amount of data shared by two ball regions, called the *proximity of regions*, is analyzed and techniques for its quantification are

proposed. The proposed techniques are developed for general metric spaces, which naturally subsume the case of vector spaces. These techniques satisfy the following criteria: (1) the proximity is measured with sufficient precision; (2) the computational cost is low; (3) it can be applied to different metrics and data sets; (4) storage overhead is moderate.

Note that in a generic metric space it is not easy to obtain such measurements since we are not able, for instance, to compute volumes or areas. Therefore, the problem of the proximity of ball regions is analyzed using a probabilistic approach. After an analysis of the computational difficulties of the proximity measurement, heuristics to compute it in an effective and efficient way are proposed. An extensive validation was performed to prove the high accuracy of the proposed approaches. It was also shown that the accuracy of the proximity measurement has a high impact on the application involved. In fact, it was observed that, when the probabilistic approach was used, the performance of the approximate similarity search algorithm based on the proximity was much higher than the same algorithm when a trivial measurement of the proximity was adopted.

1.4 Outline of the thesis

Chapter 2 introduces the issues of similarity search. We discuss applications that need this approach instead of exact match search, discuss differences between handling similarity search using vector spaces or with generic metric spaces, and introduce similarity search queries. Finally, the data sets used for the tests performed to validate the approaches proposed in the thesis are described.

Chapter 3 describes the most important access methods for improving performance

of similarity search. In particular tree-based access methods were described in detail, as the results of this thesis are mainly applicable to them. These access methods were discussed giving a general definition of their structure and their similarity search algorithms. Specific techniques for one dimensional data, multi dimensional data, spatial data, and metric data are then presented.

Chapter 4 discusses the measurement of the proximity of ball regions in metric spaces. This measurement is needed by one of the approximate similarity search algorithms proposed later. Given the difficulty of computing this measurement efficiently in metric spaces, we propose some heuristics to estimate it efficiently and accurately. Tests performed to validate the techniques proposed are discussed and analyzed.

Chapter 5 deals with the issue of approximate similarity search. After an introduction to this technique, some of the most important existing approaches are described.

Chapter 6 proposes four new techniques for approximate similarity search in metric spaces. These techniques are defined by relaxing similarity search algorithms on tree-based access methods for metric data so that the trade-off between the accuracy of results and performance can be effectively controlled. The results obtained from extensive testing are discussed and analyzed.

Chapter 2

Similarity search in metric spaces: overview and preliminaries

2.1 Introduction

Traditional databases use exact match to search for data records. Retrieved records are those whose attributes exactly match the ones specified in the query. However, for several applications exact match search is not suitable and similarity search should be used instead. In these cases, data that are similar to the query are retrieved.

In some applications requiring similarity search, data are represented in a vector space. Similarity is obtained by computing the distance between vectors: the smaller the distance, the higher the similarity. Several access methods were defined to improve performance of similarity search for data represented in vector space. These, typically, allows efficient similarity search to be performed when the number of dimension of the vector space is below 15.

There are several other applications where geometric interpretation of distance between vectors cannot be effectively used to asses similarity between represented data or, simply, where data cannot be represented in a vector space. One example

is image similarity. Even though image's features are represented by vectors, for instance color histograms, sometimes vectors that are geometrically distant could be more effectively considered similar, because of the phenomenon of cross-talk between dimensions representing individual colors. An example where data cannot be even represented in a vector space is the string similarity. The similarity between two strings is typically assessed by computing the number of operations needed to transform one into another. Strings cannot be effectively represented by real valued vectors and no geometric interpretation can be inferred from their distance function.

In all these cases geometric properties and information on the coordinates are not available – only pair-wise distances can be computed. For this reason, the study of data represented in metric spaces and access methods for similarity search in metric spaces has recently attracted much attention.

In this section we justify the need of techniques for similarity search, when data are represented in a metric space, and give some preliminary definitions useful for the remainder of the thesis.

We first discuss some applications where exact match search cannot be applied and similarity search should be used instead. We compare similarity search when data are represented in metric spaces and vector spaces, and we discuss the different basic statistic techniques that can be respectively used to build optimized access methods.

Finally we describe the specific data sets that we have used as test beds for validating the techniques proposed in this thesis.

2.2 Similarity search and its applications

Traditional database systems typically manage files of homogeneous records. Each record is composed of a specified set of attributes that are used to characterize the entities they represent. Values associated with attributes are supposed to describe each record unequivocally and completely. For instance records representing customers of a bank may be characterized by the attributes `FirstName`, `FamilyName`, `SocialSecurityNum`, and probably a few others. Each record, by means of the values associated with its attributes, unambiguously represents a particular customer of the bank.

In these systems, typically *exact match search* is used to retrieve interesting records from the database. Queries specify values for a subset of record attributes and only records whose attribute values exactly match those specified in the query are retrieved. Not retrieved records contain at least one attribute, among those specified in the query, whose value differs from that of the query attribute. For instance, a typical query submitted to a traditional database system would be, informally, *retrieve the records corresponding to the customer whose name is John Smith.* In this case, the system would retrieve the records contained in the database, where the value associated with the `FirstName` field is exactly John, and the value associated with the `FamilyName` field is exactly Smith. Eventually, when there are several matching records, the query can be defined to be more selective by adding additional convenient attribute values, like for instance the date of birth, the city of birth, or the social security number.

Exact match search basically partitions the set of records, contained in the database, in two subsets. One, sometime called the *result set*, contains the records that exactly

match the query. The other contains the records where at least one attribute value does not exactly match the corresponding query attribute value. However, there are some applications where partitions distinguishing qualifying from not qualifying records cannot be defined. In some of these applications all records possibly match the query and a "degree" of matching can be computed instead, comparing each record with the query. In these circumstances, the concept of *similarity search* is introduced as opposite to the exact match search, to indicate that retrieved records are those having high degree of matching with the query. Qualifying and not qualifying records can be identified, for instance, by specifying a threshold on the degree of similarity or on the number of records retrieved – see Section 2.4 for a formal definition of similarity search queries. In the following we outline some typical applications where similarity search capabilities are needed.

Multimedia document retrieval: Probably one of the most common application of similarity search is the retrieval of multimedia documents. Application in this area are for instance face recognition, object recognition, finger print matching, voice recognition, and in general multimedia databases [ABH97, YI99]. In this case retrieval is not performed by using directly the raw content of document, for instance, comparing pixel by pixel two images. Documents of the database are processed and salient *features*, which describe the content of documents, are extracted. Search is performed by comparing features extracted from the queries with features extracted from documents of the database. Let us consider image documents for instance. In this cases, typically, the feature extracted are dominant colors, textures, and shapes [Cas96, SO95, Smi97], generally represented as vectors. In case of color features, for instance, the color spaces is

quantized in a certain number of colors, the quantity of each individual color in the whole image is stored in each dimension of the vector, obtaining a color histogram [Smi97]. Searching for images similar to a query image corresponds to retrieve images whose color histogram is similar to the query color histogram.

Computational biology: Another very interesting application comes from the computational biology. The basic objects managed and studied in this area are DNA and protein sequences. Typically they are modeled as huge sequences of symbols, that can be easily represented by very long text strings. The problem here is finding a given sequence of letters, corresponding to some particular characteristic, inside a longer sequence. In most cases it is not possible to find an exact match of the given short sequence, and computational biologist are more interested in finding parts of a longer sequence which are similar to a short sequence. In fact, there could be minor differences in genetic sequences that somehow identify the species which they belong to. In this area, the similarity among sequences is based on the probability of mutations that somehow rearrange the sequences themselves [Wat95].

Web search engines: Web search engines traverse the World Wide Web and index the Web sites that they visit. Users may express queries to search for web sites they are interested in. Suppose, for instance, that a user wants to buy a vacation package to Hawaii that costs less that 1000$. Probably, he would specify the keywords `"vacation Hawaii less than 1000$"`, or `"low price vacation package to Hawaii"`, or perhaps `"please, search for web sites that offer vacation packages to Hawaii that cost less than 1000 dollars"`, or other similar variants. Of course all previous queries do not exactly specify

what the user is interested in and similarity search is needed to retrieve web sites that are judged to be relevant. Web search engines are a particular applications of the more generic *information retrieval systems* [SM83], which allow the retrieval of text documents to be performed by specifying list of keywords as queries.

Filtering and recommender systems: Filtering systems [BC92, MMLP97] and recommender systems [RV97] build and maintain a profile of their users. This profile is built by using various strategies, ranging from analysis of the behaviour of the user [MS94] using the system, to techniques of relevance feedback [All96]. In case of filtering systems, the profile is used to filter out what is not interesting to the user and to provide him only with interesting documents (for instances sport news from new agencies). In case of recommender systems, it is used to suggest something to the users (for instance a book to buy, a web site to visit, an advertisement). In both cases what the system has to do is to compute the similarity between the profile of the user and elements to be suggested or to be filtered.

Stock quotes: A further example is suggested by systems that manage stock quotes, which are typically represented as time series [AFS93]. In most cases, in order to take reliable decisions on investments, it is useful to check if circumstances similar to the current one (represented by some time series) occurred in the past, provided that experts of the fields have defined how to compare time series representing stock quotes. In fact, if a certain situation occurred in the past, evaluating what occurred in the following periods helps to understand what could be the trend in the future.

In addition to the previous examples, several other applications might need the use of similarity retrieval. An exhaustive analysis of these applications is out of the scope of this thesis, however we can broadly identify three circumstances where exact match search provides little help and similarity search should be used instead:

i) data cannot be precisely described and represented

ii) queries cannot be precisely specified

iii) data that exactly match a query do not exist in practice

In several cases, a combination of circumstances i), ii), and iii) occur.

Let us consider case i). Sometimes, it is not possible to have a precise description of represented entities. Consider for instance, images, videos or audio documents. Typically these documents are not searched by directly using their raw content. They are associated with description of their content, generated either manually, or automatically. These description might be subjective and ambiguous, in case of those manually generated, or can be too generic and imprecise, in case of those automatically generated. In both cases, what can be retrieved, is the set of descriptions of documents that present some high degree of coincidence with the query, or simply that are similar to the query.

In case ii) the imprecision applies to the query specification. In some cases, users are not capable of precisely expressing their needs with a query. This may happen, for instance, because the system does not offer expressive and unambiguous query specification facilities, or simply because users have a foggy idea of what they are really looking for. This is the case of all techniques of *query by example*, which are largely used in image retrieval systems, for instance. In these systems, the query

image is either sketched by the user, or an arbitrary preexisting image is used as query. In both cases what the user searches for is a set of images that are somehow similar to the one used as a query.

Case iii) occurs when the probability of finding something that is exactly equal to the query is practically 0. This is, for instance, the case of stock quotes management. Of course, the probability that time series of the past exactly match the current ones is very low. However, as we said above, it is very interesting to retrieve situations similar to the current one.

All applications described above require as a basic functionality the possibility to compute the similarity between representation of data managed by the retrieval system. In the remainder of the thesis we generally refer to these representation as *objects* or *data objects*, without distinguishing and defining the nature of the representations themselves. Accordingly data used as queries will be here called *query objects*. Systems supporting previous applications should be able to efficiently retrieve data objects stored in the database which are similar to the query object.

2.3 From vector spaces to generic metric spaces

In several similarity search applications, data are represented by tuples of real numbers (for instance color histograms). In these cases, data are defined in a *vector space*, or better in a *finite dimensional vector space*, since the number of elements in a tuple is supposed to be finite. Similarity between objects can be indirectly obtained by using the *distance* between vectors: the smaller the distance the higher the similarity.

There are several possibilities to measure the distance between vectors. The most

widely used distance functions, however, are those of the family of Minkowski distances (see Section 2.3.3) where the distance between vectors is interpreted geometrically. For instance, the Euclidean distance belongs to the family of Minkowski distances.

In order to speed up retrieval in large collections of data, with respect the trivial sequential scan, access methods have been developed. The basic idea, which they are based on, is the definition of convenient partitions of objects such that only few subset of objects have to be accessed in order to answer to similarity queries (see Chapter 3 for details). The use of vectors and Minkowski distances offers a great advantage to this purpose since it is possible to exploit geometric and coordinate information to obtain optimal partition and to achieve high performance. In fact, access methods for vector spaces such as k-d-Trees [Ben75, Ben79], R-Trees [Gut84, BKSS90], quad-Trees [Sam95], X-Trees [BKK96], to mention some, rely heavily on geometric and coordinate information. In this context, optimal algorithms have been proposed both for the average and the worst case [BWY80] to answer to nearest neighbor queries. An excellent survey of access methods for data represented in vector spaces can be found in [GG98] and [BBK01]. However, it has been seen that all these techniques degrade their performance when the number of dimensions of the vector space increases. In fact, these access methods suffer of the so called *dimensionality curse*. That is, they have an exponential dependency on the dimensionality of the space. When the number of dimensions is greater than 10–15, a linear scan over the whole data set would perform better [BGRS99, WSB98].

However, in many cases, it has been observed that, even if objects are represented in a high dimensional vector space, fewer dimensions are in fact needed to represent

effectively their geometric distributions [Fal96]. For instance, let suppose that we have objects represented in a 100 dimensional vector space, analyzing them in most case we might notice that they are in fact mostly distributed on an hyper-plane so that, by some ad-hoc transformations, we may preserve the geometric relationships among objects, but we use less dimensions to represent them. This is also the direction followed by some dimensionality reduction techniques that use, for instance, Karhunen-Loeve Transformation (KLT) [Fuk90], Discrete Fourier Transform (DFT) [OS89], Discrete Cosine Transform (DCT) [Kai85], or Discrete Wavelet Transform (DWT) [Cas96], to transform data from a vector space with a high number of dimensions to another with fewer dimensions. All these methods assume that a few dimensions are enough to retain the most important information regarding the represented data objects. However, they might introduce some approximation since not always the geometric relationships are exactly preserved.

In the past, given these established techniques that can be used in vector spaces, the problem of similarity search has always been approached by enforcing the use of vectors to represent objects. However, this strategy, that somehow guarantees that similarity searches can be executed efficiently, does not guarantee effectiveness as well. In fact, in several cases, vectors and Minkowski distances fail to accurately represent similarity relationships among objects. For instance, in the image processing field, several functions used to compare two images, cannot be easily expressed as distance between two vectors. An important example is the *quadratic form distance* [FSA+95, HSE+95, NBE+93] (see Section 2.3.3) that takes into account the so called *cross-talk* between different dimensions. The quadratic form distance does not belong to the family of Minkowski distances and it does not preserve geometric properties

of the vector space which it is applied to. Other important examples that cannot be handled by using vector distances are the string similarity, used for instance to compare DNA sequences in genetic databases, or the set similarity (see again Section 2.3.3).

Notice that when arbitrary distance functions are used to compute similarity between objects (even if represented as vectors), the geometric properties and the coordinates cannot be exploited and, as a consequence, the access methods for vector spaces indicated above cannot be used as well. Recently, the study of objects represented in generic *metric spaces* and consequently the design of access methods for metric spaces, as M-Trees [CPZ97], Slim-Trees [TTSF00], Vantage Point Trees [Chi94, Uhl91, Yia93, Yia99, BÖ99], or GNAT [Bri95] has attracted much attention in the field of similarity search. In metric space the only information that can be obtained about objects is their distance from other objects. Formal definition of metric spaces is given at the end of this section. Of course these new approaches are also able to deal with vector spaces and the Minkowski distances, since these are special cases of metric spaces: only the distances between vectors are considered and the fact that they are vectors is ignored, so no geometric information on coordinates are exploited. By doing this, an immediate advantage is the independence on the dimensionality of objects. In fact, in [CPZ97] is shown that M-Trees perform better than R*-Trees [BKSS90] when applied to high dimensional vector spaces. An excellent survey of techniques for searching in metric spaces is provided in [CNBYM01].

In this thesis we suppose that objects are represented in generic metric spaces, given their generality, their wide applicability, and their increasing demand in the similarity search area. In this way, we are not tied to specific cases of similarity

search problems, as with vector spaces, even if our results can also be applied to them.

In the following we give some definitions that are needed in the remaining of the thesis.

2.3.1 Metric spaces

Let $\mathcal{D}$ be a domain of objects, for instance image features or DNA sequences. Let d a function defined as follows:

$$d : \mathcal{D} \times \mathcal{D} \rightarrow \Re$$

that for each triple of objects $O_x, O_y, O_z \in \mathcal{D}$ satisfy the following properties:

(1) $d(O_x, O_y) = d(O_y, O_x)$ *(symmetry)*

(2) $0 < d(O_x, O_y) < \infty, O_x \neq O_y$ *(positiveness)*

(3) $d(O_x, O_x) = 0$ *(reflexivity)*

(4) $d(O_x, O_y) \leq d(O_x, O_z) + d(O_z, O_y)$ *(triangle inequality)*

We define

$$\mathcal{M} = (\mathcal{D}, d)$$

a *metric space*. $\mathcal{D}$ is the domain of the metric space, while d is the distance function of the metric space. For practical reasons, in this thesis, we assume that the maximum possible distance between objects of the considered domain $\mathcal{D}$ never exceeds a value d_m, thus we consider a *bounded metric space*. This is not a restriction, since all practical applications, which we have dealt with up to now, consider distances to be bounded.

Notice that there are no assumptions on the nature of the objects of the domain $\mathcal{D}$. There are only some constraints on the distance function d. It is easy to show that the family of Minkowski distances satisfies these properties, so a vector space is also a metric space when they are used to measure the vector distances.

In the rest of the thesis, when we use the the term *distance function* we suppose that it always satisfies the properties defined above. We also use simply $\mathcal{M}$ to indicate a metric space, that implicitly is defined by a domain $\mathcal{D}$ of objects and by a distance function d.

2.3.2 Metric ball regions

Access methods for vector spaces, typically partition the data set and bound elements of the partition by hyper-rectangular region. In metric space, the notion of hyper-rectangle cannot be used, since it requires the use of coordinates. However we may use the notion of ball regions. Given an object O in a metric space $\mathcal{M}$, we define a *ball region* $\mathcal{B}(O, r)$ as

$$\mathcal{B}(O, r) = \{O_i \in \mathcal{D} \mid d(O, O_i) \leq r\}$$

A ball region $\mathcal{B}(O, r)$ is determined by a center $O \in \mathcal{D}$ and a radius $r \geq 0$. It includes such objects from $\mathcal{D}$ so that their distances from O are less than or equal to r.

2.3.3 Similarity and distance functions

Computing similarity between objects can be done in several ways, depending on their specific representation. Similarity functions are generally defined as follows:

$$sim : \mathcal{D} \times \mathcal{D} \rightarrow [0, 1]$$

given two objects of the considered domain, their similarity is a real value in the interval between 0 and 1.

However, as previously observed, a suitable formalization for the similarity search problem is to represent objects in a metric space and to search for objects whose distance from the query object is small. Accordingly to this formalization, a distance function that satisfies the metric postulates should be provided instead of the similarity function. In several cases, the distance function is obtained naturally from the definition of the specific similarity search application. In other cases, where only the similarity function is available, the distance function should be obtained from it. Intuitively, the higher the similarity the smaller the distance, therefore, given a similarity function sim, a distance function d can be obtained as follows:

$$d(O_1, O_2) = 1 - sim(O_1, O_2), \tag{2.3.1}$$

where O_1 and O_2 are two objects of $\mathcal{D}$. When d satisfies the metric postulates, $\mathcal{M} = (\mathcal{D}, d)$ is a metric space.

The similarity function can be obtained from a distance function as well, provided the metric space is bounded, that is, provided that a distance between two objects never exceeds a fixed distance d_m:

$$sim(O_1, O_2) = \frac{(d_m - d(O_1, O_2))}{d_m}.$$

Relationships between similarity and distance function are discussed in [CPZ98b].

In the following we discuss some of the most important metric distance functions and similarity functions that result in metric distance functions when Equation 2.3.1 is applied.

Minkowski distances: Let us first, consider the case where the objects are real

valued vectors. Let us suppose that we have two vectors v_x and v_y defined in a vector space with *dim* dimensions. This implies that both v_x and v_y are tuples composed of *dim* real values. The traditional way of computing similarity among objects represented in a vector space is to use a function of the family of the Minkowski distances. This set of distance function is often designated as the L_p distance and it is defined for vectors v_x and v_y as

$$L_p(v_x, v_y) = \left(\sum_{j=1}^{dim} \mid v_x[j] - v_y[j] \mid^p \right)^{1/p}, p \geq 1.$$

Varying the value of p, we may enumerate the various function of this family. For instance, L_1 is known as the *city-block* or *Manhattan* distance, and it corresponds to the sum of the distances along each individual dimension of the considered vector space. L_2 is the famous and intuitive *Euclidean* distance. Another interesting function of this family is the L_∞, which corresponds to compute the limit of L_p, when p goes to ∞. L_∞ is the maximum among the distances along each individual coordinate:

$$L_\infty = (v_x, v_y) = \max_{j=1}^{dim} \{ \mid v_x[j] - v_y[j] \mid \}.$$

Figure 2.1 sketches how the shape of a ball region $\mathcal{B}(O, r)$ changes when different L_p's are used.

Quadratic form: The family of the Minkowski distances considers different vector coordinates to be independent, thus distances are strictly related to the closeness of vectors in a multi-dimensional space. However, in many cases, vector coordinates can be dependent or correlated. Good examples of such type data

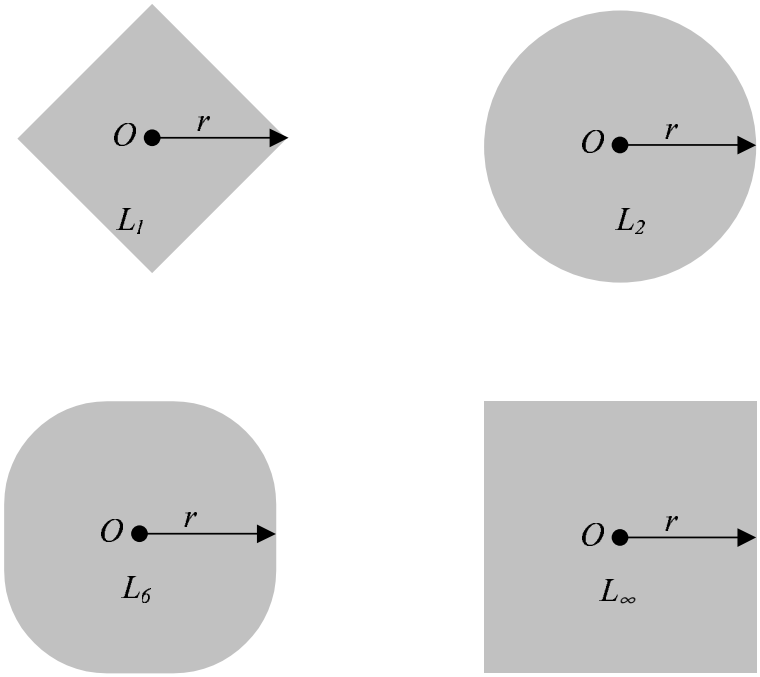


Figure 2.1: Shape of a ball region $\mathcal{B}(O, r)$, in a two dimensional vector space, when respectively L_1, L_2, L_6, L_∞, are used as distance functions.

are color histograms [Smi97] with each dimension representing a different color. Obviously, orange and pink are certainly more similar than red and blue. In order to effectively measure the distance between color histograms, this natural (though also subjective) *cross-talk* of dimensions should be taken into account properly [Fal96]. One way to handle this problem is to use the *quadratic-form* distance [FSA+95, HSE+95, NBE+93].

$$d_{qf}(v_x, v_y) = \sqrt{(v_x - v_y)^T \mathbf{A} (v_x - v_y)},$$

where $\mathbf{A} = [a_{i,j}]$ is a correlation matrix between dimensions of vectors v_x and v_y, and the superscript T indicates the vector transposition. Naturally, there is no direct correspondence between distances and positions of vectors in the space. If the correlation matrix $\mathbf{A}$ is symmetric and its diagonal is 1, that is

when $a_{i,j} = a_{j,i}$ and $a_{i,i} = 1$, then d_{qf} satisfies the metric properties.

String distance: There are several cases where objects can be effectively represented by strings of arbitrary length and retrieval is performed by searching for string that partially match a query string [HD80, Nav01]. This is quite common in computational biology [Wat95], signal processing [Lev65], text retrieval [Dam64], for instance. In all these applications distance among two strings s_1 and s_2 is defined as the minimal cost of a sequence of operations that transform s_1 into s_2. The cost is ∞ when that transformation sequence does not exist. The cost of a sequence of operations is computed as the sum of the cost of the individual operations. The most common distance functions are defined as sequences of some of the following operations:

op_1: insert a letter

op_2: delete a letter

op_3: replace a letter

op_4: swap adjacent letters

Depending on which of the previous operation are allowed, different distance functions are defined:

- *Levenshtein or Edit distance* [Lev65]: allows insertions and replacements.
- *Humming distance* [SK83]: allows only replacements.
- *Longest Common Subsequence distance* [NW70, AG87]: allows only insertions.

It is easy to show that these distance functions satisfy the metric postulates, when all individual operations is given the same cost.

Set similarity: Another interesting case is the *similarity of sets*. Given two sets $\mathcal{S}_1$ and $\mathcal{S}_2$, which contains arbitrary elements, a largely used similarity function is defined as the ratio of the number of their common elements to the number of all different elements

$$S_T(\mathcal{S}_1, \mathcal{S}_2) = \frac{\#(\mathcal{S}_1 \cap \mathcal{S}_2)}{\#(\mathcal{S}_1 \cup \mathcal{S}_2)},$$

where $\#(\mathcal{S})$ is the number of elements in set $\mathcal{S}$.

Tanimoto similarity: A generalization of the set similarity is the *Tanimoto* similarity measure [Koh84]. It is used for instance to compute the similarity between structural features of biological data [GPR97]. Let us suppose that we have vectors v_x and v_y defined in a vector space. Let us indicate with $v_x \cdot v_y$ the *scalar product* of v_x and v_y, and with $\|v_x\|$ the *Euclidean norm* of v_x. The *Tanimoto* similarity is defined as

$$S_T(v_x, v_y) = \frac{v_x \cdot v_y}{\|v_x\|^2 + \|v_y\|^2 - (v_x \cdot v_y)}.$$

When vectors contain only 0's and 1's, the similarity of sets is obtained. In this case 0 means that an element, identified by a vector position, does not belong to the set, while 1 means that it belongs to it. Notice that the distance function, obtained in correspondence of the Tanimoto similarity, satisfies the metric properties but it does not preserve the geometric closeness of vectors.

Hausdorff distance: As a final example, consider the *Hausdorff distance*, which is used to compare shapes of images [HKR93]. Here the compared objects are sets

of relevant points, for example high curvature points. Given two finite point sets $A = \{a_1, a_2, \ldots, a_p\}$ and $B = \{b_1, b_2, \ldots, b_q\}$, the Hausdorff distance is

$$H(A, B) = \max(h(A, B), h(B, A)),$$

where

$$h(A, B) = \max_{a \in A} \min_{b \in B} \| a - b \|,$$

and $\| \cdot \|$ is some underlying norm on the points of A and B, for example the L_2 or Euclidean norm. Also this distance function satisfy the metric postulates.

2.3.4 Statistical information on data

The use of statistical information on data has always played a key role for the performance optimization of database systems. Statistical information is in fact at the basis of many cost models used for the optimization of query execution. In addition, it is also used to better organize the structure of some access methods for performance improvement. There are some differences on the type of statistics that can be obtained depending on the fact that geometric information can be used, as with vector spaces and distance of the Minkowski family, or no geometric information is available, as in generic metric spaces. In the first case, statistics on the position of the objects in the space can be exploited. In the second case, only information on the distances can be used and no geometric information on data can be exploited. In the following we first introduce the probabilistic notions of density and distribution functions then we discuss how these applies to our scenario to characterize data represented in metric spaces.

Distribution and density functions

Let suppose that V is a continuous random variable [HPS71]. Mathematically it is a real-valued function, defined on a probability space, which takes real values, depending on some event whose probability is 0.

The *distribution function* F_V of a random variable V is the following probability:

$$F_V(v) = \mathtt{Pr}\{V \leq v\}$$

For instance, let us suppose that V is a continuous random variable associated with the height of people, then $F_V(v)$ is the probability that chosen a random person (the event), its height is smaller than v. Notice that, chosen a random person, the probability that its height is *exactly* v is 0 of course, given that v is a real value.

The *density function* f_V of a random variable V is a function such that

$$\int_{-\infty}^{+\infty} f_V(v)dv = 1$$

Of course, we have that

$$F_V(x) = \int_{-\infty}^{x} f_V(v)dv$$

When the random variables are more than one, for instance V_1 and V_2, then we have respectively the *joint distribution*, $F_{V_1V_2}(v_1, v_2)$, and the *joint density*, $f_{V_1V_2}(v_1, v_2)$. The joint distribution is defined as follows

$$F_{V_1V_2}(v_1, v_2) = \mathtt{Pr}\{V_1 \leq v_1 \wedge V_2 \leq v_2\}$$

The joint density $f_{V_1V_2}(v_1, v_2)$ is a function such that

$$\int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f_{V_1 V_2}(v_1, v_2) dv_1 dv_2 = 1$$

As before, we have that

$$F_{V_1 V_2}(a, b) \int_{-\infty}^{b} \int_{-\infty}^{a} f_{V_1 V_2}(v_1, v_2) dv_1 dv_2 = 1$$

This approach can be extended to an arbitrary number of random variables.

Distance distribution versus data distribution

When a data set contains objects represented in a vector space, a useful property that can be obtained by analyzing it is the *data distribution*. By using it is possible to know, for each arbitrary region of the space, what is expected amount of object contained in it or, in probabilistic terms, what is the probability that a random object belongs to the region. This can be obtained by integrating the *data density function* over the given region. For instance, Figure 2.2 shows the data density function, say $f_{XY}(x, y)$, in a two dimensional vector space, where X and Y are continuous random variables corresponding to the x and y coordinates of objects. In the Figure, dark values correspond to high values of $f_{XY}(x, y)$, while light values correspond to low values. Data density, or data distribution, has been used on vector spaces to characterize data sets for various purposes, as for instance to develop effective cost models [BKK97, FK97, KF93, PM97, TS96] for multi-dimensional access methods such as R-Trees [Gut84] or R*-Trees [BKSS90].

In this thesis we assume that data objects are represented in a generic metric space, and vector spaces are just a special case. As discussed in Section 2.3.1, metric spaces allow only to measure distance between two objects, and in particular no

coordinate systems can be used. This, among other things, has the implication that no information on data distribution can be exploited. In fact, data distribution cannot be obtained since an object in a metric space does not have an identifiable position and the only thing that can be known is its distance from other objects.

The characterization of generic metric spaces requires a novel approach. In [CPZ98a] the use of the *distance distribution* is proposed as a counterpart of data distribution used for vector spaces, since it is the "natural" way to characterize metric spaces. Given an object O_i, the distribution of distances from O_i, characterize how the other objects are distributed around O_i. For instance if we report these observations to a two dimensional vector space the result is what is sketched in Figure 2.3. That is given the distance distribution or the distance density with respect to an object O_i, we might know what is the amount of objects whose distance from O_i is smaller than a certain value or, in probabilistic terms, what is the probability that a random object has a distance from O_i smaller than a certain value. However no information on the "dense" zones of the space is available, since an object whose distance from O_i is d, may be on any point on the circumference with center O_i and radius d. If the vector space has three dimensions the circumference become a sphere and so on. In particular, in a "pure" metric space, no guesses on the position of the object can be made, since there is not position. The distribution of distances from O_i is sometime called the O_i's *viewpoint*.

In addition to the distribution of distances from a specific object, another property that can be easily computed is the *overall distance distribution*. Given a distance d the overall distance distribution says what is the probability that distances smaller than d exist, that is, what is the probability that given two random objects their

distance is smaller than d.

Managing the overall distance distribution is far more easy than managing the viewpoint of every possible object of the data set. In [CPZ98a] it is shown that the overall distance distribution can be used, instead of the viewpoints, when the data set is probabilistically *homogeneous*, that is when there is no significant *discrepancy* between the various viewpoints. In case of highly non homogeneous spaces, the viewpoint of a certain individual object is significantly different that that of a different object. However, it was found that real and "realistic" synthetic data sets are highly homogeneous so the overall distance distribution can be reliably used in practice to characterize them. These notions are more formally discussed in the following.

Let D_O be a continuous random variable corresponding to the distance $d(O, \mathbf{O_1})$, where $\mathbf{O_1}$ is a random object. The *distance density* $f_{D_O}(x)$ represents the probability[1] of distance x from object O. The corresponding *distance distribution* $F_{D_O}(x)$, that is the probability of a distance to O of being at most x, can be determined as

$$F_{D_O}(x) = \int_0^x f_{D_O}(t)\, dt \tag{2.3.2}$$

Given two different objects $O_x, O_y \in \mathcal{D}$, the corresponding viewpoints $F_{D_{O_x}}$ and $F_{D_{O_y}}$ are different functions.

In the remaining of this thesis we will simply use F_{O_i}, to indicate the distance distribution or the viewpoint with respect to the object O_i.

The overall distribution of distances over $\mathcal{D}$ can be defined as

$$F(x) = \mathtt{Pr}\{d(\mathbf{O_1}, \mathbf{O_2}) \leq x\}, \tag{2.3.3}$$

[1]We are using continuous random variables so, to be rigorous, the probability that they take a specific value is by definition 0. However, in order to simplify the explanation, we slightly abuse the terminology and use the term probability to give an intuitive idea of the behavior of the density function being defined.

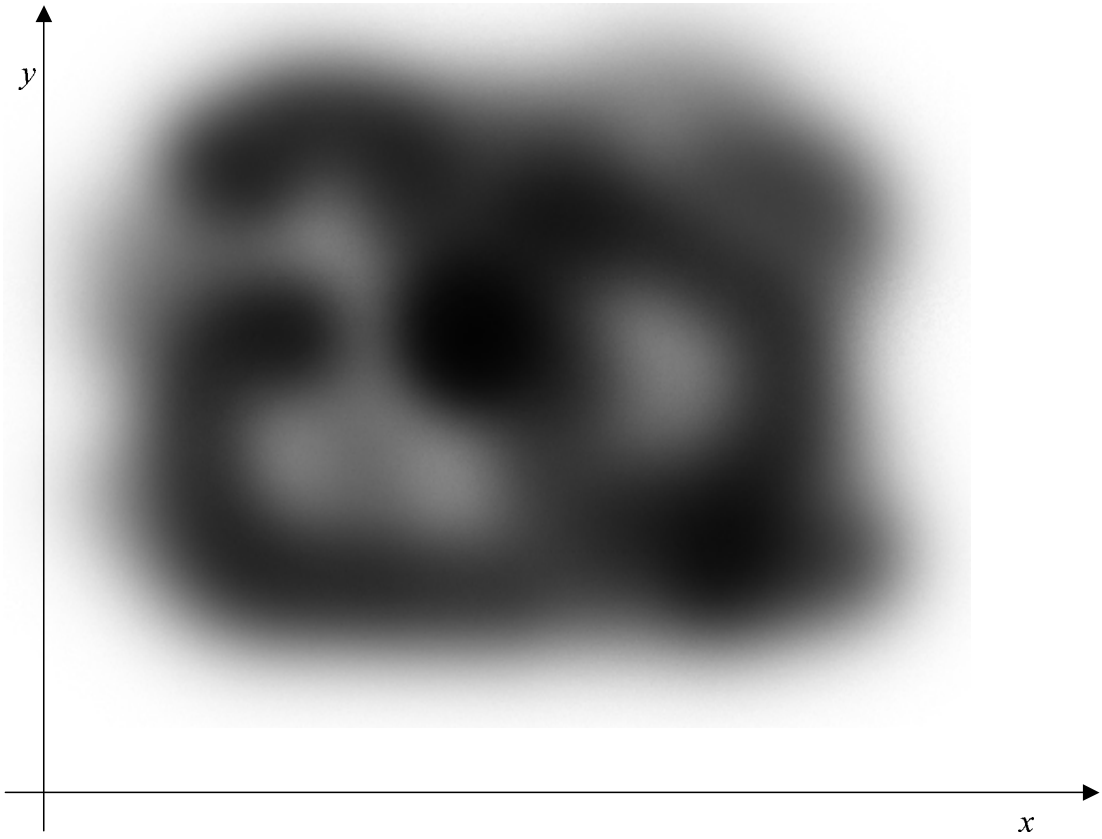


Figure 2.2: Density of data in a two dimensional vector space

where $\mathbf{O_1}$ and $\mathbf{O_2}$ are two independent random objects of $\mathcal{D}$.

Th *discrepancy* between two viewpoints is defined as

$$\delta(F_{O_i}, F_{O_j}) = Avg[\mid F_{O_i}(\mathbf{x}) - F_{O_j}(\mathbf{x}) \mid] \tag{2.3.4}$$

where $\mathbf{x}$ is a random distance in the interval $[0, d_m]$, and d_m is the maximum distance between two objects of the dataset. The discrepancy between two viewpoints is the average difference of values that they assume varying the distance.

The *index of homogeneity of viewpoints* HV of a metric space $\mathcal{M}$ is defined as

$$HV(\mathcal{M}) = 1 - Avg[\delta(F_{\mathbf{O_1}}, F_{\mathbf{O_2}})]$$

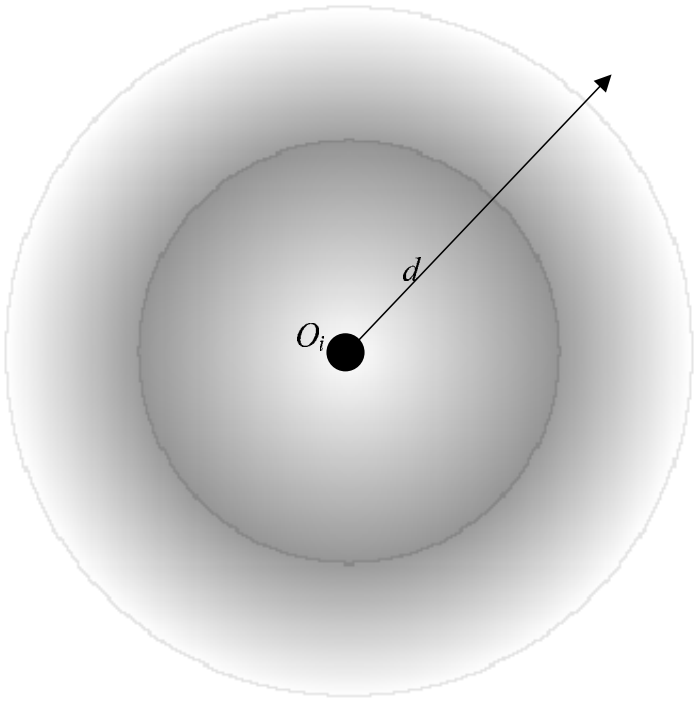


Figure 2.3: Density of distances from the object O_i

where $\mathbf{O_1}$ and $\mathbf{O_2}$ are random points of $\mathcal{U}$. The index of homogeneity of viewpoints of a metric space is 1 minus the average discrepancy of the viewpoints.

When $HV(\mathcal{M}) \approx 1$, two different relative distance distributions are likely to behave similarly and in [CPZ98a] was shown that the overall distance distribution $F(x)$ can be used as the representative instead of any F_{O_i}.

As we said before, for real and realistic synthetic data sets, including the ones that we have used in this thesis 2.5, the index of homogeneity was found to be close to 1. Accordingly, we use the overall distance distribution $F(x)$, in our experiments, for any reference object O_i.

Notice that, even if we neglect the computational complexity of a procedure that would be able to determine $F(x)$, all objects from $\mathcal{D}$ are usually not known. Therefore,

an estimation of $F(x)$ is computed, by considering distances between objects from a representative sample of $\mathcal{D}$.

2.4 Similarity queries

The most frequently used types of similarity queries are *similarity range* and the *nearest neighbors* queries. Accordingly, in this thesis we concentrate our efforts on these two, even if other forms of similarity queries exist, as for instance the *all pairs* or the *all nearest neighbors* queries.

In similarity search applications, in principle, it is not possible to distinguish the set of objects that qualify from those that do not qualify. In fact, all objects qualify to a similarity query and the only possibility is to asses the degree of similarity between each object and the query. The result is that, theoretically, all objects should be retrieved in correspondence of a similarity query. Of course this is not realistic for practical applications, and the possibility of distinguishing between qualifying and not qualifying objects should be somehow enforced. There are two basic techniques to obtain this, in correspondence of which similarity range and nearest neighbors queries are defined.

The first possibility, which results in similarity range queries, is to use a threshold on the degree of similarity with the query, or correspondingly on the distance from the query. Qualifying objects are those whose similarity with the query is above the specified threshold, or correspondingly, whose distance is below the specified threshold.

The second possibility, which results in nearest neighbors queries, is to specify the maximum number of objects to be retrieved. Let us suppose that k is the number of

object to be retrieved. Qualifying objects are the k most similar objects to the query, or correspondingly, the k objects closest to the query.

In the first case, the response set is determined by the maximal dissimilarity with respect to the query, while in the second case the response is constrained by the maximal number of closest objects. In the following a formal definition of similarity range and nearest neighbors queries is given.

Let us suppose that we have a data set $\mathcal{DS} \subseteq \mathcal{D}$ in metric space $\mathcal{M} = (\mathcal{D}, d)$ and a query object $O_q \in \mathcal{D}$. The similarity range query $\mathbf{range}(O_q, r_q)$ for query object O_q and search radius r_q is defined as

$$\mathbf{range}(O_q, r_q) = \{O_i \in \mathcal{DS} \mid d(O_q, O_i) \leq r_q\}. \tag{2.4.1}$$

Notice that a range queries is in fact defined by the ball region $\mathcal{B}(O_q, r_q)$, that we will call the *query region*.

The nearest neighbors query $\mathbf{nearest}(O_q, k)$ for query object O_q and k nearest neighbors is defined as

$$\begin{aligned} \mathbf{nearest}(O_q, k) \quad = \{O_i \in \mathcal{DS} \mid \quad & 1 \leq i \leq k \wedge \\ & \forall j < k, d(O_j, O_q) \leq d(O_{j+1}, O_q) \wedge \\ & \forall j > k, d(O_k, O_q) \leq d(O_j, O_q)\}. \end{aligned} \tag{2.4.2}$$

Similarity response sets are ordered (sorted, ranked) sets, where the position of an object in the set is determined by its distance with respect to O_q.

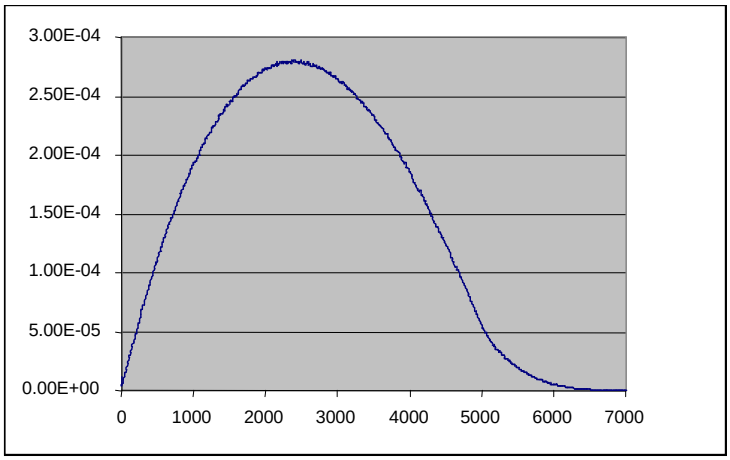
Some access methods for similarity search partition the data set into disjoint subsets which are characterized by bounding regions. Such aggregate information is used for pruning regions (subsets) that cannot contain qualifying objects. When a range query is considered, it is easy to imagine that regions having no overlap with the query region can safely be pruned.

In fact, the situation is quite similar even in case of the nearest neighbor queries. By considering the distance to the k-th nearest neighbor, we can transform the nearest neighbors query to a range query. The only problem is that this radius is not known in advance, so the query region. However, good nearest neighbor search strategies only access the regions that have an overlap with this theoretic query region. In order to achieve this behaviour, typically a priority queue of candidate regions is maintained, and the regions are accessed starting with the most promising one.

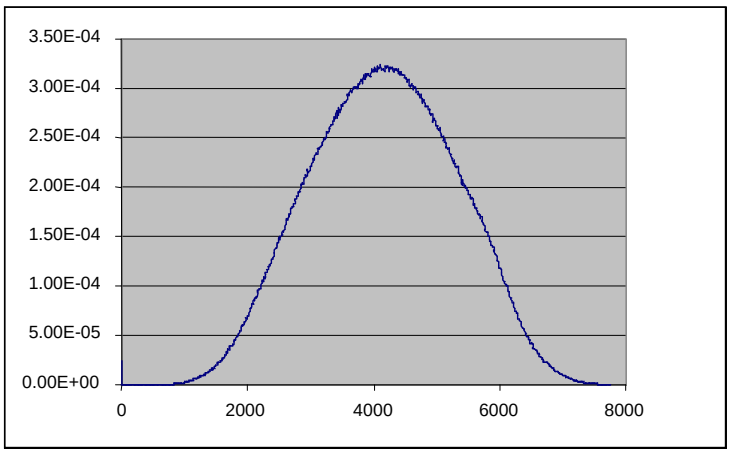
Details of similarity search algorithms that implement Equation 2.4.1 and 2.4.2 using access methods are discussed in Chapter 3.

2.5 Data sets used in this thesis

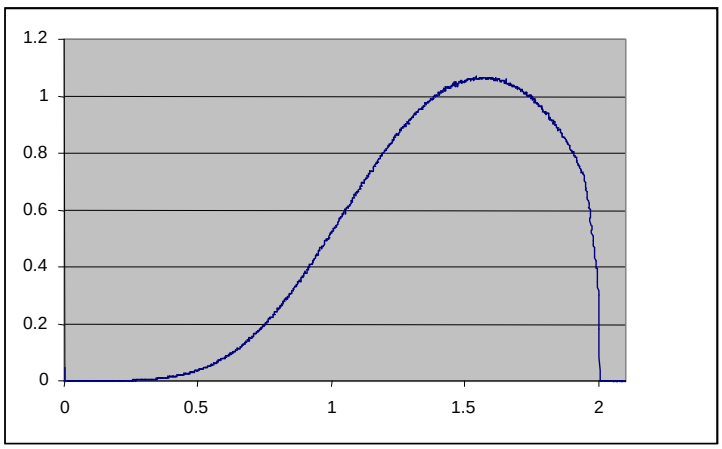
We have conducted our experiments using three data sets: one synthetic and two real-life data sets of image color features. The synthetic data set, called UV, is a set of vectors uniformly distributed in a 2-dimensional space where vectors are compared through the Euclidean (L_2) distance. The second data set, designated as HV1, represents color features of images. In this case, the features are represented as 9-dimensional vectors containing the *average*, *standard deviation*, and *skewness of pixel values* for each of the red, green, and blue channels [SO95]. An image is divided into five overlapping regions, each one represented by a 9-dimensional color feature vector. This results in a 45-dimensional vector as a descriptor of one image. The distance function to compare two feature vectors is again the Euclidean (L_2) distance. The third data set, called HV2, contains color histograms represented in 32-dimensions. This data set was obtained from the UCI Knowledge Discovery in Databases Archive, [HB99]. The color histograms were extracted from the Corel



UV Data set



HV1 Data set



HV2 Data set

Figure 2.4: Overall distance density functions of the data sets used for the experiments

image collection as follows. The HSV space is divided into 32 subspaces (colors), using 8 ranges of hue and 4 ranges of saturation. The value for each dimension of a vector is the density of each color in the entire image. The distance function used to compare two feature vectors is the histogram intersection implemented as L_1. Data sets UV and HV1 contain 10,000 objects, while HV2 contains 50,000 objects.

The range of distances and the corresponding overall distance density functions can be seen in Figure 2.4. Note the differences in densities of the individual data collections that were selected to test proximity for qualitatively different metric spaces, i.e. spaces characterized by different data (distance) distributions.

Notice that even if these data sets are defined in a vector space, we ignore this and in the remaining of the thesis we deal with them as if they where defined in a generic metric space. For instance, we never make any assumption on data distribution, and only distance distribution is used when needed.

Chapter 3

Access methods for similarity search

3.1 Introduction

Searching for data in large repository is intrinsically a difficult task. In order to perform search operations efficiently a physical level support is needed. Accordingly, several access methods were designed to optimize the execution of both exact match and similarity search operation.

In the following we introduce three classes of access methods: sequential scan techniques, hashing techniques, and tree-based techniques. Since the approaches proposed in this thesis are applied to tree-based access methods, we briefly introduce sequential scan and hashing techniques. Then, we dedicate more space to tree-based access methods. We define a general structure for tree-based access methods and we also introduce the basic similarity search algorithms. The definition of some specific tree based access methods is finally discussed distinguishing four different application areas: one dimensional data, multi dimensional data, spatial data, and metric data.

3.2 Access methods for similarity search

Literature offers several excellent surveys related to access methods for similarity search. As an example consider [GG98, BBK01, CNBYM01]. Here we describe some of the most significant approaches related to the issues discussed in this thesis.

In this chapter, for convenience, we broadly classify the access structures organizations for efficient processing of similarity search queries into three categories:

a) sequential scan

b) hashing

c) tree-based

Access structure organizations for *sequential scan* organize data in such a way that a sequential scan of the whole data base can be efficiently executed. *Hashing methods*, organize data in buckets according to some hashing functions so that only a few buckets should be accessed to process a query. *Tree-based access methods*, organize data in a tree hierarchy. Only a few nodes of the tree should be accessed to process a query.

Comparison of different access methods can be performed considering several aspects. However, as suggested in [GG98], among the important features of access methods is worth considering the followings:

Dynamicity: Access methods should allow insertion and removal of entries, without affecting their performance

Secondary storage: Since databases contains huge amount of data, data structures

used to organize entries cannot be maintained into main memory and should be designed to make an efficient use of secondary storage.

Independence of input sequence: Performance of access methods should not depend on the order in which data are inserted in the index.

Scalability: Performance of access methods should not degrade quickly as the size of the database increases.

Space efficiency: The space required to maintain the access structure should be limited.

The algorithms and the techniques presented in this thesis are mainly devoted to tree-based access method. Therefore, in the following we will briefly introduce sequential scan and hashing techniques, then we will dedicate more space to the tree-based techniques.

3.2.1 Sequential scan

The basic technique to process similarity search queries defined in Section 2.4 is to perform a single scan of the entire database. In order to perform this efficiently, data can be densely stored on contiguous blocks of a secondary storage. The *sequential scan* techniques, access sequentially all data of the database, according to the ordering used to place data on the secondary storage. This technique is faster than accessing randomly small blocks spread widely on the secondary storage. In fact, data stored contiguously can be transferred into main memory very fast, since no head seek latency occurs. On the other hand, in case of random block transfer, head seek time is is the main reason for inefficiency. Sequential scan algorithms transfer into main

memory large blocks of data, depending on the available amount of main memory. After a block is transferred, data contained in it are processed, then a new block is transferred, until all data are processed. Performance of sequential scan techniques is linearly proportional to the size of the database. Their aim is to improve performance of similarity search algorithms by minimizing the head seek overhead. However, they cannot reduce the data transfer and data process overheads, that are linear to the size of the database.

A relevant approach for data represented in vector spaces, based on the sequential scan technique, is the VA-File (Vector Approximation File) [WSB98]. It reduces the size of the data set using a coarse approximate representation of data objects. In this representation more objects may collapse in one coarse object. Sequential scan is performed on the reduced data set containing the coarse representation. The resulting coarse result set is used to lookup in the original data set to find the actual result set.

Notice that in some applications, head seek and data transfer overhead have a minor impact with respect to data process overhead. In fact, data process overhead is mainly due to distance computation between data objects and query objects and, some applications, as for instance genetic databases, require distance functions whose computation costs much more than disk accesses.

Hashing and tree-based access methods, on the other hand, use data structures that also aim at minimizing the amount of data transferred and processed.

3.2.2 Hashing

The idea behind the hashing techniques [Knu98, WFHC92] is that data objects should be allocated in a certain number of buckets in order to access just a few of them when

a query should be processed. Each bucket typically corresponds to a disk page and has a specific capacity. A hash function takes as input an object and returns the identifier of a bucket. The object is added to the obtained bucket. A typical technique used to handle the situation where adding a new object in a bucket would exceed the bucket's capacity is to store the additional entries in overflow buckets associated with the original bucket.

Hashing techniques for exact match search use arbitrary hash functions: when a query is processed, the hash function is applied to the query object. The exactly matching data object is searched in the obtained bucket and eventually in the corresponding overflow buckets.

Hashing techniques for similarity search, as for instance range queries, use hash functions that preserve the closeness of data objects. Close (similar) objects are stored in the same bucket.

A significant hashing approach for multi-dimensional data, therefore applicable to data represented in vector spaces and distances measured using functions of the Minkowski family, is the *grid-file* [NHS84]. It partitions the search space symmetrically in all dimensions. Specifically the grid-file put a d-dimensional orthogonal grid on top of the search space obtaining a set of hyper-rectangular cells, whose shape and size might be different, depending on the regularity of the grid itself. Each cell or a group of cells are associated with data buckets. Each cell is associated with one bucket but each bucket might contain several adjacent cells. When a new object (that is a vector) should be added to the data set, first the cell that contains it is identified. If the bucket containing the cell is not full, then the object is inserted. If the bucket is full, there are two possibilities. If the bucket contains several cells, one checks if

one of the existing hyperplane of the grid can be used for splitting the bucket. If there is not such an hyperplane or if the bucket is associated with only one cell, then a new hyperplane is added to the grid so that the bucket is split. Exact queries are executed by locating the cell that contains the query object and searching inside the corresponding bucket. Range queries are processed by locating cells that overlap the query region and searching in the corresponding buckets.

Another interesting hashing approach for similarity search in metric spaces is the *D-Index* [GSZ02], whose basic techniques were also proposed in [GSZ00, GSZ01]. It is a multilevel hash structure that takes advantage of the idea of the *excluded middle partitioning*, also exploited in [Yia99] for building an access method based on excluded middle vantage point. In each level of the hash structure a certain number of buckets are defined. Each level is associated with a hash function that, given an object, assigns it to either a specific bucket or no buckets in the corresponding level. Objects that are associated with no buckets in a level become candidate to be stored in the next level. It might happen that some objects are excluded to be stored in all levels. These remaining objects are stored in a separate *excluded bucket*. In this access structure, each object is stored in one bucket of the multilevel structure or in the excluded bucket. Queries can be processed, by accessing at most one bucket per level plus the excluded bucket. The upper bound on the number of disk access is limited by the number of levels, and the number of required distance computations can be significantly reduced by using some pre-computed distances. Distance computation between query objects and some accessed objects sometime is not performed since, by using pre-computed distances, it is possible to infer that they do not belong to the result set.

3.3 Tree-based access methods and similarity search algorithms

Approaches classified as tree-based access methods currently dominate the field. Their basic principle is the hierarchical decomposition of the data space. Techniques developed in this thesis suppose that tree-based access methods are used so we devote more space to their description.

All tree-based access methods have some common features, concerning their structure. All of them, for instance, hierarchically organize the space by using regions, however there are differences concerning the definition of the used regions and the way the trees are built and maintained. For instance, the Generalized search trees (GIST) [HNP95] subsumes most of the characteristics of tree-based access methods under a generic architecture, so it can be used as a reliable framework for the implementation of new access methods.

In the following we describe the general structure of tree-based access methods, we outline the generic similarity search algorithms that exploit this structure, finally we discuss some specific implementation of tree-based access methods.

3.3.1 General structure

Tree-based access methods partition objects and store elements (set of objects) of the partition in buckets, as hashing methods also do. However, while in hashing methods a bucket identifier is obtained by applying a hash function to data objects, in tree based access methods, buckets are accessed by traversing the tree structure.

The tree structure is composed of nodes that contain pointers to other nodes.

Nodes that do not refer other nodes are called *leaf nodes*. The remaining are called *internal nodes*. Each node of the tree is associated with a region that represents a subspace of the entire search space. The region associated with a node N_i is contained in the region associated with the N_i's parent node and contains all regions associated with the N_i's descendants.

A leaf node is simply a container of references to data records along with the objects that represent them. Let's suppose that N_i is a leaf node of a tree. The generic structure of N_i is the following:

$$N_i = \left((p_1^i, O_1^i), \ldots, (p_{n_i}^i, O_{n_i}^i)\right)$$

where n_i is the number of objects contained in N_i, p_j^i is a pointer to a record (for instance an image), and the object O_j^i is the corresponding representation (for instance the image color histogram). All objects $O_j^i, 1 \leq j \leq n_i$, belong to the region associated with N_i. Notice that in traditional database systems, objects O_j^i are typically called keys.

The definition of the region associated with a node is stored in its parent node, along with its reference. In fact, an internal node is a container of references to other nodes, either internal or leaf nodes, along with the definition of their associated region. Let's suppose that N_i is an internal node of the tree. The generic structure of N_i is the following:

$$N_i = \left((r_1^i, \mathcal{R}_1^i), \ldots, (r_{n_i}^i, \mathcal{R}_{n_i}^i)\right)$$

where $r_j^i, 1 \leq j \leq n_i$, is a reference to another node, say N_j, and $\mathcal{R}_j^i, 1 \leq j \leq n_i$, is the region associated with N_j. Regions are not necessarily disjoint, that is, it is possible that some regions associated with nodes referred by the same parent node overlap.

In some tree-based access methods also internal nodes, in addition to leaf nodes, may contain references to searched data and the object that represents it. The corresponding (extended) node structure is the following:

$$N_i = \left(((p_1^i, O_1^i), r_1^i, \mathcal{R}_1^i), \ldots, ((p_{n_i}^i, O_{n_i}^i), r_{n_i}^i, \mathcal{R}_{n_i}^i)\right).$$

We will generally suppose that only leaf nodes contain pointers to real data, so we will use the simplified node structure. However, when needed we will mention the other possibility.

Let's suppose that N_p is an internal node and node N_c one of its child nodes. Let $\mathcal{R}_p$ be the region associated with N_p and $\mathcal{R}_c$ the one associated with N_c. As we have previously said, region $\mathcal{R}_c$ is included in region $\mathcal{R}_p$, that is if object $O \in \mathcal{R}_c$ then $O \in \mathcal{R}_p$.

The number of elements n_i in a node N_i, either leaf or internal node, and the type of regions associated with nodes depend on the specific access method.

The entry point to the tree is the *root node*. The root node is associated with a region representing the entire universe.

Notice that even if regions associated with different nodes might overlap, a reference to a record whose representation is contained in the intersection of two regions is typically stored just in one node.

When only leaf nodes contain references to real data, leaf nodes represent a partition of the whole data set. The strategies used to partition the data set and to bound elements of the partition with regions are specific for each access method.

Some tree-based access methods are designed to be maintained into main memory and do not take into consideration secondary memory storage. These are not suitable for large databases. Other access methods are primarily designed to be maintained

into secondary storage. In this case, in order to minimize disk access overhead, size of nodes of the tree corresponds to the size of a disk page. In this way reading a node of the tree corresponds to one disk access.

Most tree-based access methods are built using strategies that maintain them height-balanced. Therefore, the lengths of the paths from the root node to all leaf nodes are identical. In these cases, the length of the path from the root to the leaf nodes is called the *height* of the tree.

3.3.2 General similarity search algorithms

The hierarchical structure of tree-based access methods allows similarity search algorithms to ignore subsets of objects where qualifying objects cannot be found. Search algorithms traverse the tree structure top down, starting from the root, discarding paths corresponding to sub-trees where no qualifying objects can be found. Leaf nodes reached by search algorithms are exhaustively searched for qualifying objects.

For the sake of simplicity we suppose that data objects and references to searched data are only contained in the leaf nodes. However, it is straightforward to modify the algorithms in order to have a correct behaviour on tree structures where also internal nodes contain references to searched data.

The algorithms described in this section solve range and nearest neighbors queries defined in Section 2.4. The intuition behind these algorithms is the following. In case of range queries, only nodes associated with regions intersecting the query region $\mathcal{B}(O_q, r_q)$, where O_q is the query object and r_q the query radius, should be accessed. If the region associated with a node does not intersect the query region, all regions associated with nodes belonging to the sub-tree rooted at this node do not intersect

the query region and leaf nodes of the sub-tree do not contain objects that are covered by the query region. Therefore the entire sub-tree can be safely pruned. In case of nearest neighbors search, the behaviour is similar, since we will see that the nearest neighbors search algorithm is in fact defined as a range search algorithm where the query radius is reduced step by step. A query region with a large radius is used at the beginning. Then the radius is reduced step by step depending on the nearest objects currently found.

Similarity search algorithms on tree-based access methods traverse multiple branches of a tree to collect objects that satisfy the query. Accordingly, both algorithms use a dynamic queue of *Pending Requests*, **PR**, to temporarily store pointers to tree's nodes, whose bounding regions potentially contain qualifying objects, to be accessed in the next iterations of the algorithms. Each node corresponds to a different branch to be searched in the trees structure.

Range search algorithm

The response $\mathbf{range}(O_q, r_q)$ to the similarity range query of query object O_q and search radius r_q can be determined by the Algorithm 3.3.1.

The algorithm takes as input values the query region, composed of the query object O_q and the query radius r_q. It returns the set of pairs $\mathbf{range}(O_q, r_q) = \{(p_1, O_1), (p_2, O_2), ..., (p_l, O_l)\}$ where p_i is a pointer to a record, O_i is the object that represents it, and l is the number of elements retrieved.

The algorithm starts with an empty result set and a queue of pending requests **PR** containing only the pointer to the root of the tree (Step 1). Then the algorithm iterates while **PR** contains some nodes to be accessed (Steps 2 and 11). At each

Algorithm 3.3.1. **Range**
Input: query object O_q; query radius r_q.
Output: response set **range**(O_q, r_q).

1. Enter pointer to the tree root into **PR**; empty **range**(O_q, r_q).
2. While **PR** $\neq \emptyset$, do:
3. Extract entry N from **PR**. Suppose that N is bounded by region $\mathcal{R}$.
4. Read N.
5. If N is a leaf node then:
6. For each $(p_j, O_j) \in N$ do:
7. If $d(O_q, O_j) \leq r_q$ then $(p_j, O_j) \rightarrow$ **range**(O_q, r_q).
8. If N is an internal node:
9. For each child node N_c of N, bounded by region $\mathcal{R}_c$ do:
10. If $Overlap(\mathcal{B}(O_q, r_q), \mathcal{R}_c)$ then insert pointer to N_c into **PR**.
11. End

iteration, a node is extracted from **PR** (Step 3) and its content is read (Step 4). Notice that in case of secondary storage access methods, reading a node corresponds to one disk access. If the extracted node is a leaf node (Step 5) the algorithm checks if the contained objects are covered by the query region. Covered objects, along with the corresponding record pointers, are inserted in the result set (Steps 6 and 7). If the extracted node is an internal node (Step 8) it refers to other nodes of the tree, so the algorithm checks if these child nodes should be accessed. Each referred node is inserted in **PR**, to be examined in the next iterations, if the region, they are associated with, overlaps the current query region (Steps 9 and 10).

Algorithm 3.3.2. **Nearest neighbors**
Input: query object O_q; number of neighbors k.
Output: response set **nearest**(O_q, k).

1. Enter pointer to the tree root into **PR**; fill **nearest**(O_q, k) with k (random) objects; determine r_q as the max. distance in **nearest**(O_q, k) from O_q.
2. While **PR** $\neq \emptyset$, do:
3. Extract the first entry N from **PR**. Suppose that N is bounded by region $\mathcal{R}$.
4. If $Overlap(\mathcal{B}(O_q, r_q), \mathcal{R})$ then read N, exit otherwise.
5. If N is a leaf node then:
6. For each $(p_j, O_j) \in N$ do:
7. If $d(O_q, O_j) < r_q$ then update **nearest**(O_q, k) by inserting (p_j, O_j) and removing the most distant pair from O_q; set r_q as the max. distance of objects in **nearest**(O_q, k) from O_q.
8. If N is an internal node:
9. For each child node N_c of N bounded by region $\mathcal{R}_c$ do:
10. If $Overlap(\mathcal{B}(O_q, r_q), \mathcal{R}_c)$ then insert pointer to N_c into **PR**.
11. Sort entries in **PR** with increasing distance to O_q.
12. End

Nearest neighbors search algorithm

The response **nearest**(O_q, k) to the nearest neighbors query for query object O_q and k nearest neighbors, according to [HS95], can be executed by the Algorithm 3.3.2 in the optimum way, i.e. with the minimum number of accessed nodes.

There is a strong similarity between Algorithms 3.3.1 and 3.3.2. However we can highlight two main differences. First, the nearest neighbors algorithm maintains a dynamically shrinking query radius while the search radius (specified as input parameter) is constant for range queries. Second, the queue of pending requests **PR** does not assume any ordering for the range queries, while increasing distance to the query object is used for ordering in the approximate nearest neighbors algorithm – a

node close to the query object should be accessed first.

The algorithm for approximate nearest neighbors search takes as input values the query object O_q, the number of neighbors required k. It returns the set of pairs $\mathbf{nearest}(O_q, k) = \{(p_1, O_1), (p_2, O_2), ..., (p_k, O_k)\}$ where p_i is a pointer to a record, O_i is the object that represents it, and $d(O_q, O_i) \leq d(O_q, O_j), i < j$.

The algorithm starts with a result set initialized with k random objects from the data set, the current query radius r_q set to the maximum distance between the query object and the objects in the result set, and the queue of pending requests **PR** containing only the pointer to the root of the tree (Step 1). Then the algorithm iterates while **PR** contains some nodes to be accessed (Steps 2 and 12). At each iteration, a node is extracted from **PR** (Step 3). The content of the node is read only in case the region associated with it overlaps the query region (Step 4) elsewhere the algorithm exits since no other region in **PR** may overlap the query region, given the ordering used in **PR** (Step 11). If the extracted node is a leaf node (Step 5) the algorithm checks if the contained objects are nearer to the query objects than those in the current result set. In this case, these nearer objects are inserted in the result set, along with the corresponding record pointer, elements exceeding the k-th position of the updated result set are removed, and the current query radius r_q is set to the distance between the query object and the object that represent the k-th element in the result set (Steps 6 and 7). If the extracted node is an internal node (Step 8) the algorithm checks if pointed nodes should be considered or not. Each pointed node is inserted in **PR**, to be examined in the next iterations, if the region they are associated with overlaps the current query region (Steps 9 and 10). The entries in **PR** are sorted according to their distance to the current query object (Step

11); the distance of an object from a region is the nearest possible distance of any point inside the region from the object. Notice that given the dynamic behaviour of the query radius r_q, a node that is inserted in **PR** at Step 10 can be pruned at Step 4 since, after its insertion and before its extraction, the query radius might be reduced.

3.4 Specific tree-based access methods

In this section we describe some specific tree-based access methods. We have considered four different application areas and for each of them we have described the most popular tree-based access methods. The first application area is that of *one dimensional data*, where an absolute ordering among data can be conveniently exploited. This is a typical case in traditional database systems. B-Trees are the most popular tree-based access methods in this case. The second application area is that of *point or multi-dimensional data*, where data are represented by points in a multi-dimensional vector space. The point access methods described here are k-d-Trees and quad-Trees. The third application area is that of *spatial data*, where spatial objects like polygon, or curves, in addition to points, are managed. R-Trees are typically used as access methods in this case. Finally, the fourth application area is that of *metric or distance only data*, where no coordinate information can be exploited since only distances among data objects can be computed. M-Trees are a significant access method for this kind of data.

As previously stated, techniques developed in this thesis are defined for metric data. Specifically they were tested by using M-Trees as access method.

3.4.1 One dimensional access methods: B-Trees

B-Trees [BM72], and their extensions [Com79], especially B^+-Trees, are the most popular tree-based access methods. They can be used for one dimensional data, that is in those case where an absolute ordering among data can be exploited, and are widely used for high performance primary key accesses, in traditional databases. B-Trees are secondary storage structures, therefore, node's size corresponds to the size of a disk page. Each node, either leaf or internal node, may have several pointers to descendant (either nodes or data records).

The structure of a leaf node of a B-Tree is defined exactly as in Section 3.3.1. That is, it just contains a certain number of objects (keys) along with the corresponding references to real data (records).

Regions associated with nodes are disjoint adjacent intervals of the one dimensional space. The region $\mathcal{R}_j^i$ associated with the j-th child of the i-th node is

$$\mathcal{R}_j^i = [a_j^i, b_j^i)$$

Notice that, since intervals are adjacent,

$$b_j^i = a_{j+1}^i$$

so the structure of the node can be conveniently simplified by only storing the left bounds of the intervals.

Internal nodes of B-Trees, in addition to references to child nodes and associated regions, also contain references to data records, as in the extended internal node representation given in Section 3.3.1. Let us call p_j^i the j-th reference contained in node N_i. p_j^i points to the record identified by the left bound b_j^i of the j-th interval, that is, $O_i = b_j^i$ is the key of the pointed record.

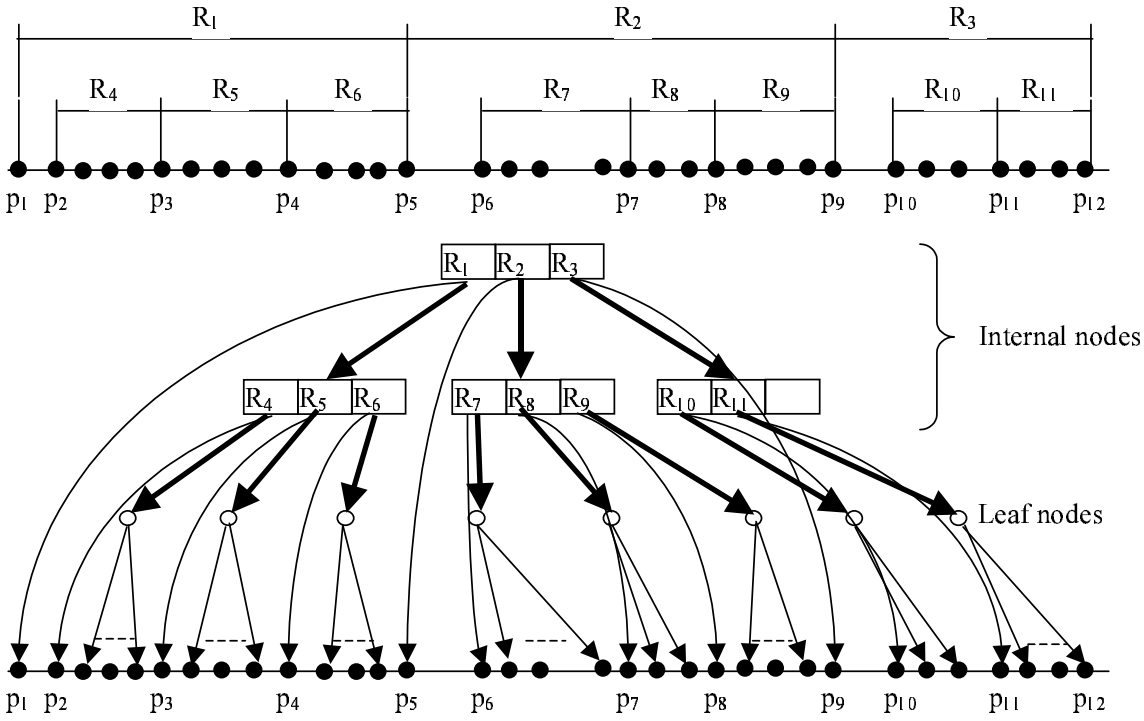


Figure 3.1: B-Trees structure example, supposing that the maximum number of entries in a node is three

Figure 3.1 shows the structure of a B-Tree and how intervals are organized.

B^+-Trees, simplify the B-Tree structure. In fact, in B^+-Trees only leaf nodes refer to data records, while internal nodes only refer to other nodes.

B-Trees and B^+-Trees are always height-balanced. The technique to guarantee this is to have the tree growing from the leaves to the root. Insertion first locates the leaf node where the new entry should be inserted. If the node is full, it is split in two. As a consequence of this, the entry referring the old node should be modified and a new entry, corresponding to the new node should be added in the parent node. If the parent node is also full, it is also split in two and splits may propagate up to the root of the tree. When also the root is split, a new level (a new root) is added to the tree, guaranteeing the height-balancing.

3.4.2 Point access methods: k-d-Trees and quad-Trees

Access methods described in this section are able to deal with points in a vector space, even if some of them were extended to deal also with spatial data as for instances polygons. We describe two different tree-based point access methods, namely k-d-Trees and quad-Trees, and discuss some of their extensions.

k-d-Trees

K-d-Trees [Ben75, Ben79] are binary search trees that exploit a recursive subdivision of the space. Let us suppose that our objects are vectors with dim dimensions. In every level of the tree a value (key) is used as discriminator for branching decision of a specific dimension of the corresponding vector space. The root node (level 0) discriminates for the first dimension. Nodes pointed by the root (level one) discriminate for the second dimension. And so on, up to the level $dim - 1$. If more than dim levels are needed, the process starts again from the first dimension. Space subdivision is obtained by $(dim - 1)$-dimensional hyperplanes. Each splitting hyperplane has to contain at least one data object. See Figure 3.2 for an example.

Leaf nodes just contain a reference to a record and the object (vector) that represents it, or they are empty in case that no objects are contained in the corresponding region. Of course empty leaf nodes can be materialized when needed, saving a lot of unused memory space.

The structure of internal nodes is slightly different than that described in Section 3.3.1:

$$N_i = \left((p_i, O_i), (r_l^i, \mathcal{R}_l^i), (r_r^i, \mathcal{R}_r^i)\right)$$

An internal node contains a single pointer to a record, along with its representation,

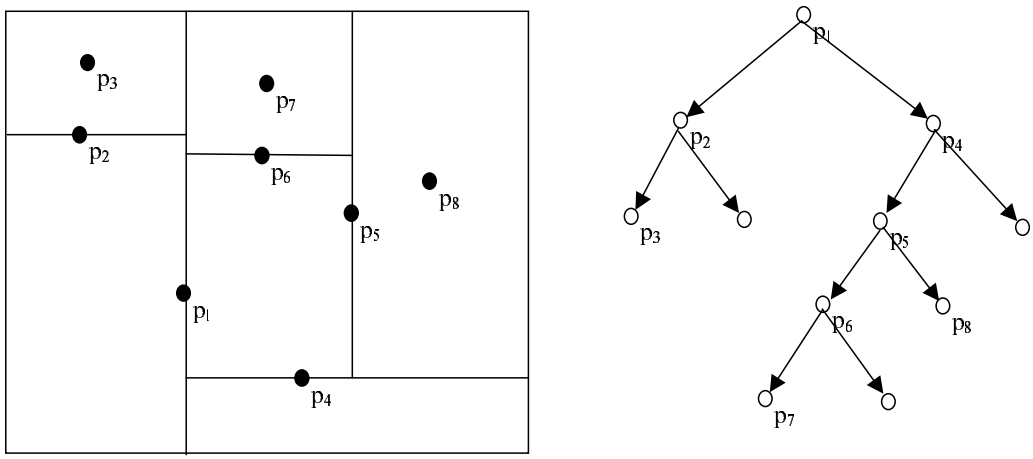


Figure 3.2: k-d-Trees structure example

and two entries. The left entry points to the left child node and contains a description of the region associate with it. The right entry, symmetrically points to the right child node and contains a description of its bounding region. Modifications of search algorithms for this modified structure are straightforward.

Let us suppose that N is a node at level lev of the tree, and object $O = (o_0, \ldots, o_{dim-1})$ is the object contained in it. Let $X = (x_0, \ldots, x_{dim-1})$ be the axis of our dim-dimensional vector space. The left region $\mathcal{R}_l$, associated with the left child node, is obtained by the intersection of the semi-space $x_{(lev \mod dim)} \leq o_{(lev \mod dim)}$ and the region $\mathcal{R}$ associated with the node N itself. Symmetrically, the right region $\mathcal{R}_r$, associated with the right child node, is obtained by intersecting the semi-space $x_{(lev \mod dim)} > o_{(lev \mod dim)}$ and the region $\mathcal{R}$. Notice that the hyperplane $x_{(lev \mod dim)} = o_{(lev \mod dim)}$ contains the object O.

It is straightforward to observe that the structure of a node can be conveniently simplified by only storing the value $o_{(lev \mod dim)}$, corresponding to the $(lev \mod dim)$ dimension, and the pointers r_l and r_r respectively to the left and right nodes.

The name k-d-Trees stands for k-dimension tree, where k is used as the dimensionality of the indexed space.

k-d-Trees have been designed as main memory access structures, but the extended k-d-Trees [CF80] has been modified for secondary storage environments. This scheme uses the k-d structure as a directory to data stored in pages, but only the leaves of the directory contain these page pointers.

A successful attempt to generalize the B-Trees and k-d-Tree ideas in one is presented in [Rob81] and called the k-d-B-Trees. In geometric terms, the data space is divided into a set of *dim*-dimensional boxes, which are also divided in boxes, etc., down to the depth of the tree. Instead of a set of one-dimensional points, each branch node in a k-d-B-Tree contains a set of regions. Each region is represented by the set of $2 \cdot dim$ coordinates which define its boundaries. The union of all regions in a branch node is the region which encloses them. The lowest level nodes, representing the "innermost" regions, contain pointers to sets of tuples pertinent to these regions. Alternatively, the leaf nodes may contain coordinates of their content points, plus another level of indirection to the actual tuples corresponding to these points.

quad-Trees

As previously discussed, k-d-Trees are binary trees where each node discriminate values of a single dimension of the vector space. The problem is that this data structure degrades its performance when the number of dimensions is high. An alternative possibility is to build a data structure where each node provides 2^{dim} descendants in order to ensure discrimination for all dimensions at the same time. This multi-way trees, extensively discussed in [Sam95], are called *quad-Trees* even if usually this term refers

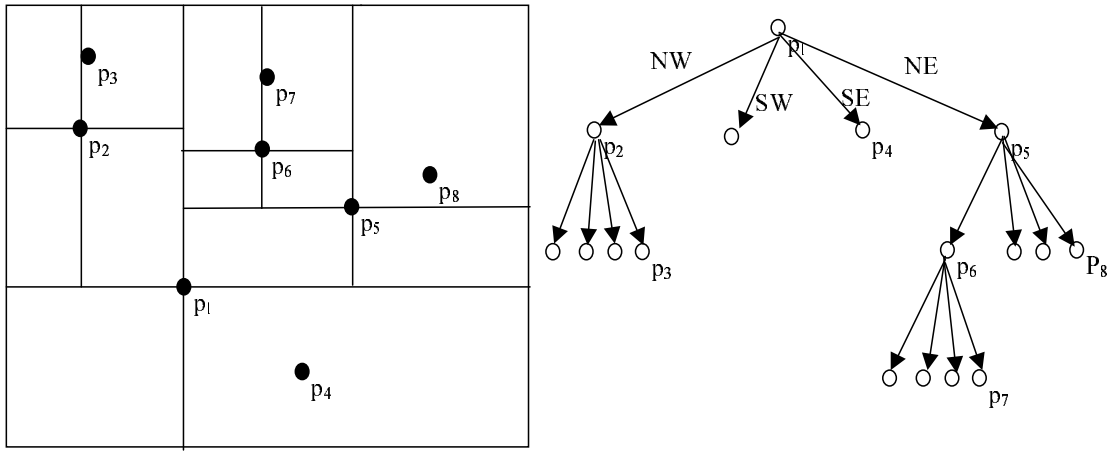


Figure 3.3: Point quad-Trees structure example

to the two-dimensional variant, where each node has four descendants. Quad-trees as k-d-Trees were also originally designed as main memory data structures.

In the two dimensional case, regions associated with nodes are rectangles. Let us suppose that N_i is an internal node. Its four descendants are associated with the four rectangles obtained by cutting the rectangle associated with N_i with an horizontal and a vertical straight lines. The four obtained rectangles are typically referred as the NW, NE, SW and SE (northwest, etc.) quadrants.

There are several variants of the quad-Tree structure. A very important one is the *point quad-Tree* [FB74]. In point quad-Trees, internal nodes have the following structure:

$$N_i = \left((p_i, O_i), (r_1^i, \mathcal{R}_1^i), \ldots, (r_{2^{dim}}^i, \mathcal{R}_{2^{dim}}^i)\right)$$

Similarly to k-d-Trees, each internal node contain a single reference to record along with the object that represents it. However, 2^{dim} entries, are included instead of just two. Each entry refers to a child node and contains a description of the region (hyper-rectangle) associated with the child node. See Figure 3.3 for an example.

As in k-d-trees leaf nodes either just contain one object, or they can be empty, in case no objects are contained in the corresponding region.

In a point quad-tree, objects (points in the *dim*-dimensional vector space) can be inserted consecutively. When a new object is inserted, first a point search is performed to check if the object already exist. If the object is not found, then it is inserted in the leaf node where the search stopped. If the leaf node already contains another object, then the associated region is divided into 2^{dim} subspaces, by using *dim* hyperplanes all containing the object contained in the leaf node. The former leaf node become an internal node containing 2^{dim} regions and referring 2^{dim} empty leaf nodes. The new object is inserted in the new leaf node referred by the entry containing the unique region that includes it.

Another popular variant of quad-Trees are the *region quad-Trees* [Sam95]. In this variant, regions are always partitioned regularly. That is, the 2^{dim} subspaces resulting from a partition are always of equal size. The hyperplanes used to cut regions do not necessarily contain the associated data object, as in point quad-Trees. For instance in a 2 dimensional vector space, they are such that they divide each side of the rectangular region in two segments of equals length. Some complex versions of region quad-Trees, as for instance the PM quad-Trees [SW85] or those described in [Sam88], are able to deal with spatial data in addition to points.

3.4.3 Spatial access methods: R-Trees

Spatial access methods are able to deal with spatial data such as polygons, curves, regions, etc. in addition to points. The most significant proposal of spatial access methods are the R-Trees, discussed in the following. Other spatial access methods

mainly refine the original design of R-Trees.

R-Trees were originally proposed in [Gut84]. They are height-balanced trees similar to B^+-Trees [BM72] (see also Section 3.4.1). Spatial objects are represented by their minimum bounding box. Therefore, objects are defined as follows:

$$O_i = (I_0, \ldots, I_{dim-1})$$

where I_i is an interval $[a, b]$ of the i-th dimension of the vector space, describing the extent of the spatial object along the i-th dimension. Of course, other representations for the bounding boxes are also possible.

Leaf nodes contain sets of objects (bounding boxes of spatial objects), along with references to real spatial objects.

Internal nodes are defined exactly as in Section 3.3.1. The region $\mathcal{R}_c$ associated with a child node N_c is the minimum bounding box including all objects contained in the leaf nodes of the sub-tree whose root is N_c. Regions associated with nodes at the same level of the tree may overlap, however each object is stored only in one leaf node even if it is covered by several regions. See Figure 3.4 for an example.

When a new object is inserted, the tree is traversed by choosing at each level the child node whose region needs the minimum enlargement to contain the object. If several children satisfy the same criterion, the one associated with the smallest region is chosen. Once arrived to a leaf node, if the corresponding region needs to be enlarged, it is adjusted appropriately, propagating the changes upward. If the leaf node is full, it is split and entries are distributed in the new and old node creating two new regions. These updates are also propagated upwards with eventually other splits in the internal nodes. Various strategies were proposed to minimize the overlap during insertion. Some were proposed in the original R-Trees paper [Gut84], other

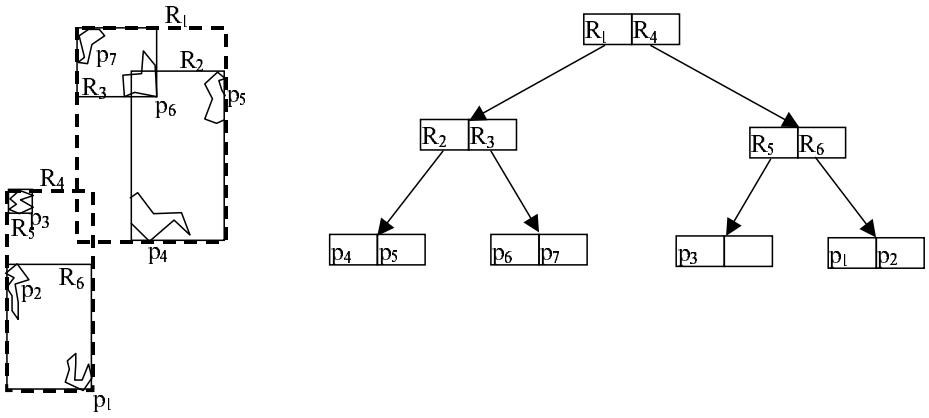


Figure 3.4: R-Trees structure example, supposing that the maximum number of entries in a node is two

were proposed later by other authors as for instance the *packed R-Tree* [RL85], or a packing techniques based on the use of the Hilbert Curve [KF93].

Zero overlap among regions is in general not obtainable. When a new data object (bounding box) that overlaps two regions is inserted, since we want to store it in just one leaf node, one of the two regions has to be enlarged, to include the object, and the consequence is that the enlarged region will overlap the other region. However, if we allow a data object, which overlaps more regions, to be stored in all overlapped leaf nodes, we can obtain non overlapping regions at the price of storing objects in several nodes. This is the main idea behind the R^+-tree [SRF87]. Another variation of the R-tree, called R^*-tree [BKSS90], does not change the properties of the basic structure, but rather has a more complex insertion algorithm that build the tree improving the overall performance.

3.4.4 Metric access methods: M-Trees

M-Trees [CPZ97] are secondary storage height balanced access methods. They organize data objects by creating partitions where set of objects are bounded by ball regions (see Section 2.3.2).

Data objects and references to real data records are stored in leaf nodes, whose structure is defined exactly as described in Section 3.3.1.

Internal nodes contain references to child nodes along with their bounding regions. In M-Trees bounding regions are ball regions defined as follows:

$$\mathcal{R}_i = \mathcal{B}(O_i, r_i)$$

where O_i is a reference objects, or routing object, used as center of the region and r_i is the radius of the ball region.

The ball region $\mathcal{B}(O_i, r_i)$ associated with the node N_i covers all regions associated with the nodes belonging to the sub-tree whose root is N_i and all objects contained in the leaf nodes of the sub-tree. That is, if $\mathcal{R} = \mathcal{B}(O, r)$ is a region associated with any node of the sub-tree, then $d(O_i, O) + r \leq r_i$. If O is any object belonging to a leaf nodes of the sub-tree, then $d(O_i, O) \leq r_i$. See Figure 3.5 for an example.

When a new object O should be inserted, the tree is traversed choosing in each level the child node that needs the minimum expansion (possibly no expansion) to contain the new object. In case of ties, the node associated with a region with the closest center is chosen. Finally the node is added to a leaf node. If it is full, it is split in two. Two new objects are chosen as centers of the two new regions, associated with the old and new node. Objects are distributed in the two nodes and their regions are defined by using the previously chosen centers and the distance from the centers to the farthest object in each node as corresponding radii. There are several strategies

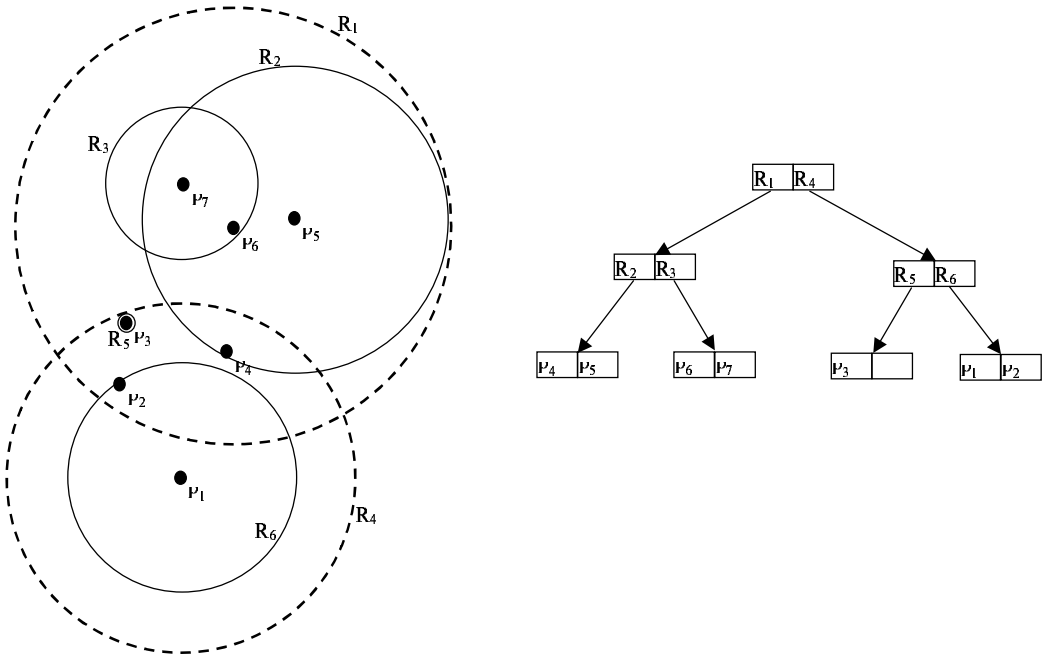


Figure 3.5: M-Trees structure example, supposing that the maximum number of entries in a node is two

that can be used to chose the two new representatives and to distribute objects in the two nodes. From various experiments, the authors say that the best strategy is trying a split that minimizes the maximum of the covering radii obtained. Notice that a split in a leaf node needs an insertion in the parent node. In case also the parent node is full then it has to be split too and the split can be propagated up to the root of the tree, as in B-Trees (see Section 3.4.1) and R-Trees (see Section 3.4.3).

As in R-Trees, regions of M-Trees may overlap and objects are stored in only one leaf node. A variation of the original design of the M-Trees that tries to minimize the overlap between regions are the *Slim-Trees* [TTSF00]. The basic structure of Slim-Trees is the same of the M-Trees. However, they use a new splitting algorithm based on Minimal Spanning Tree, a new algorithm to chose the most appropriate child node during insertion that leads to higher storage utilization, and a "Slim-down" algorithm

to be used in a post-processing step to reduce the sizes of the ball regions thus their overlaps.

Chapter 4

Proximity of ball regions in metric spaces

4.1 Introduction

The approximate similarity search algorithm, whose details are presented in Section 6.9 of next chapter, is based on an estimation of the quantity of objects shared by the query region and data regions: data regions that are judged to share few objects with the query region are discarded. To this aim, this chapter deals with the problem of estimating, given two arbitrary ball regions defined in a metric space, the amount of objects contained in their intersection. Large overlap between regions not always implies that several objects are shared by them. In fact, the number of objects contained in the intersections depends on the distribution of objects in the space. There may be regions with a large intersection and few objects in common, but also regions with a small intersection and many objects in common, which happens when the intersection covers a dense area of the data space. In this chapter, this phenomenon, called the *proximity of regions*, is analyzed and techniques for its quantification are proposed.

The problem of region proximity in vector spaces was studied in [KF92] to decluster nodes of R-trees [Gut84] for parallelism. In this chapter proximity measures are developed for general metric spaces, which naturally subsumes the case of vector spaces. Techniques that satisfy the following criteria are proposed: (1) the proximity is measured with sufficient precision; (2) the computational cost is low; (3) it can be applied to different metrics and data sets; (4) storage overheads are moderate.

The problem of the proximity of regions is analyzed using a probabilistic approach. After a discussion about the computational difficulties of the proximity measurement, heuristics to compute it in an effective and efficient way are proposed. An extensive validation was performed to prove the quality of the proposed approaches.

4.2 Formal definition of proximity

In some existing applications, such as [CPZ97, TTSF00] where proximity was used to obtain a better organization of access methods' structure, a simplified measurement of the proximity between two ball regions was used. We refer to this simplified version as the *trivial proximity*. Specifically, the trivial proximity is computed through a function, linearly proportional to the overlap of the regions, which can be generalized as follows.

$$X^{trivial}(\mathcal{B}(O_x, r_x), \mathcal{B}(O_y, r_y)) = \begin{cases} 0 & \text{if } r_x + r_y < d(O_x, O_y) \\ \frac{r_x + r_y - d(O_x, O_y)}{2 \cdot d_m - d(O_x, O_y)} & \text{if } \max(r_x, r_y) \leq \min(r_x, r_y) + d(O_x, O_y) \\ \frac{2 \cdot \min(r_x, r_y)}{2 \cdot d_m - d(O_x, O_y)} & \text{otherwise} \end{cases} \tag{4.2.1}$$

Equation 4.2.1 sets the proximity to 0 when two ball regions do not overlap. Otherwise, the proximity is proportional to the regions' intersection. The values are

normalized to obtain proximity values in the range [0,1]. The proximity is 1 when both regions include all objects, that is when their radii are equal to the maximum distance d_m.

Although the trivial proximity measure is simple to compute, it is not accurate because it does not take into account the distribution of objects in the space.

The issue of proximity is far more complex. Intuitively, the proximity of two ball regions should be a value proportional to the amount of objects that simultaneously occur in both of the regions. Accordingly, using a probabilistic approach, we may define the proximity $X(\mathcal{B}(O_x, r_x), \mathcal{B}(O_y, r_y))$ of ball regions $\mathcal{B}(O_x, r_x), \mathcal{B}(O_y, r_y)$ as the probability that a randomly chosen object $\mathbf{O} \in \mathcal{D}$ appears in both regions, i.e.

$$X(\mathcal{B}(O_x, r_x), \mathcal{B}(O_y, r_y)) = \mathtt{Pr}\{d(\mathbf{O}, O_x) \leq r_x \wedge d(\mathbf{O}, O_y) \leq r_y\} \tag{4.2.2}$$

Note that the proximity cannot be quantified by the amount of space covered by the regions' intersection. Due to the lack of space coordinates in general metric spaces, such a quantity cannot be determined.

Our aim is to use proximity to design an approximate similarity search algorithm that discard data regions with small probability of sharing objects with the query region. As discussed and proved in Section 6.9, high accuracy of proximity measurement is fundamental for high precision of the approximate similarity search algorithm. In fact our experiments give evidence that results obtained using the trivial proximity are far less accurate than results obtained using the proximity measurements developed in this thesis.

4.3 Application considerations

Several practical applications may benefit from the measurement of the proximity between two regions, for example:

Region splitting Static regions are not typical and the evolution process in storage structures is regulated by specific *split* strategies. When a region $\mathcal{R}$ splits, two new regions, say $\mathcal{R}_1$ and $\mathcal{R}_2$, are created. One way of splitting a region may be more advantageous than another – the content of a region can typically be split in several ways. When a large number of objects is contained in the intersection of two new regions (i.e. their proximity is high), the probability of accessing both regions during query execution is also high, assuming query objects have the same distance distribution as the data objects. For example, consider Figure 4.1a where regions $\mathcal{R}_1$ and $\mathcal{R}_2$, resulting from a split, cover the shared area of a cluster of objects. Queries Q_1 and Q_2 access only one region, while Q_3 access both regions. However, respecting the assumption that query objects have the same distance distribution as the data objects, queries like Q_3 are far more frequent than Q_1 or Q_2, so this partitioning is not very beneficial. Proximity can be used to detect such situations and to determine a good split.

Allocation When a new region is created, it must be placed in the storage system. In such a situation, region proximity measures can be used to determine the most suitable storage bucket for the new region. The strategy is different for single and multiple (independent) disk systems, where parallel processing can be supported. If parallel disks are available, buckets with a high probability of simultaneous access (that is buckets whose corresponding regions have a high

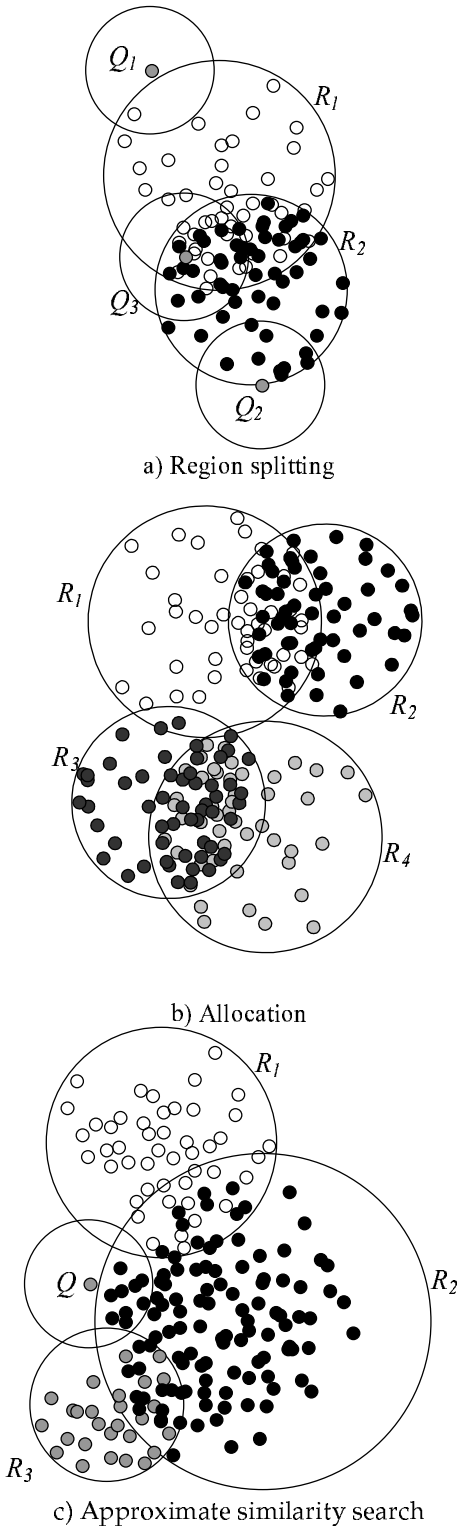


Figure 4.1: Use of the proximity measure for region splitting (a), the allocation of objects on disks (b), and approximate similarity search (c)

proximity) should not be put on the same disk, i.e. they should be *declustered* for parallel access. In a single disk environment, buckets with a high probability of being accessed together should be placed as close as possible, i.e. *clustered*. Consider the example in Figure 4.1b, where regions $\mathcal{R}_1$ and $\mathcal{R}_2$ ($\mathcal{R}_3$ and $\mathcal{R}_4$) have a high proximity. On the other hand, proximity between $\mathcal{R}_1$ and $\mathcal{R}_3$ ($\mathcal{R}_2$ and $\mathcal{R}_4$) is low. The consequence is that $\mathcal{R}_1$ and $\mathcal{R}_2$ ($\mathcal{R}_3$ and $\mathcal{R}_4$) should either be put on different disks, in the case of multiple disks, or they should be placed as close as possible, on a single disk system.

Approximate similarity search The proximity measure can also be useful for approximate similarity search algorithms to prune regions with a small probability of containing qualifying objects, that is data regions with a small proximity to the query region. Exact similarity search algorithms access all buckets whose bounding regions overlap the query region. However, even if the query region overlaps a data region, no or few data objects may appear in the intersection, i.e. the proximity between the query region and the data region is small. The result is that even though all data regions are accessed, few of them actually contain qualifying data objects (few regions have a non empty intersection with the query region). Many accesses are thus actually void and could be saved if the proximity is used as a condition for pruning. In Figure 4.1c, it can be seen that, although the query region Q intersects regions $\mathcal{R}_1$, $\mathcal{R}_2$ and $\mathcal{R}_3$, the intersection with $\mathcal{R}_1$ and $\mathcal{R}_3$ is empty and thus it is not necessary to access these regions.

One of the approximate similarity search methods developed in this thesis implements this idea. Results obtained with this method out-performs those obtained

with the other three methods. Full details of approximate similarity search by using proximity are given in Section 6.9.

4.4 Computational issues

To precisely compute proximity according to Definition 4.2.2, the knowledge of distance distributions with respect to the regions' centers is required. Since any object from $\mathcal{M}$ can become a region's center, such knowledge is not likely to be obtained. However, as discussed in Section 2.3.4 we can assume that, when the index of homogeneity of viewpoints is close to one, the distance distribution is (practically) independent of the centers themselves and depends on the distance between the regions' centers instead. This also implies that all pairs of regions with the same radii and constant distance between centers have on average the same proximity, no matter what their actual centers are. Consequently, we can approximate the proximity of two regions, whose distance between centers is d_{xy}, with the *overall proximity* $X_{d_{xy}}(r_x, r_y)$ of any pairs of regions having radii r_x and r_y, and whose distance between centers is d_{xy}. Specifically we define the overall proximity as the following conditional probability [HPS71]:

$$X_{d_{xy}}(r_x, r_y) = \mathtt{Pr}\{d(\mathbf{O}, \mathbf{O_x}) \leq r_x \wedge d(\mathbf{O}, \mathbf{O_y}) \leq r_y \mid d(\mathbf{O_x}, \mathbf{O_y}) = d_{xy}\}, \qquad (4.4.1)$$

where $\mathbf{O_x}$, $\mathbf{O_y}$, and $\mathbf{O}$ are random objects.

The overall proximity $X_{d_{xy}}(r_x, r_y)$ is defined as the probability that a random object $\mathbf{O}$ belongs to the regions with random centers $\mathbf{O_x}$ and $\mathbf{O_y}$, and radii r_x and r_y, given that the distance between centers $\mathbf{O_x}$ and $\mathbf{O_y}$ is d_{xy}.

The proximity of two ball regions $\mathcal{B}(O_x, r_x)$ and $\mathcal{B}(O_y, r_y)$ such that $d_{xy} = d(O_x, O_y)$

is therefore approximated as follows:

$$X(\mathcal{B}(O_x, r_x), \mathcal{B}(O_y, r_y)) \approx X_{d_{xy}}(r_x, r_y).$$

Now, let us consider the way how $X_{d_{xy}}(r_x, r_y)$ can be computed. Let X, Y and D_{XY} be continuous random variables corresponding, respectively, to distances $d(\mathbf{O}, \mathbf{O_x})$, $d(\mathbf{O}, \mathbf{O_y})$, and $d(\mathbf{O_x}, \mathbf{O_y})$. The *joint conditional density* $f_{X,Y|D_{XY}}(x, y|d_{xy})$ is the probability[1] that distances $d(\mathbf{O}, \mathbf{O_x})$ and $d(\mathbf{O}, \mathbf{O_y})$ are, respectively, x and y, given that $d(\mathbf{O_x}, \mathbf{O_y}) = d_{xy}$. Then, $X_{d_{xy}}(r_x, r_y)$ can be computed as

$$X_{d_{xy}}(r_x, r_y) = \int_0^{r_x} \int_0^{r_y} f_{X,Y|D_{XY}}(x, y|d_{xy}) dy dx \tag{4.4.2}$$

Unfortunately, an explicit form of $f_{X,Y|D_{XY}}(x, y|d_{xy})$ is unknown. In addition, computing and maintaining joint conditional densities as discrete functions would result in a very high number of values. The function depends on three arguments so that the required storage space is $O(n^3)$, provided n is the number of samples used for each argument. This makes the approach totally unacceptable.

We propose to compute the proximity measure by using an approximation of $f_{X,Y|D_{XY}}(x, y|d_{xy})$, designated as $f^{appr}_{X,Y|D_{XY}}(x, y|d_{xy})$, that is expressed in terms of the joint density $f_{XY}(x, y)$. Note that $f_{XY}(x, y)$ is simpler to determine than $f_{X,Y|D_{XY}}(x, y|d_{xy})$. We can observe that X and Y are independent – if we know the distance between $\mathbf{O}$ and $\mathbf{O_x}$, this does not affect the distance between $\mathbf{O}$ and $\mathbf{O_y}$, unless we add some additional information as for instance the distance between $\mathbf{O_x}$ and $\mathbf{O_y}$. Therefore, by definition, we have that

$$f_{XY}(x, y) = f_X(x) \cdot f_Y(y).$$

[1] As also stated in Section 2.3.4, we are using continuous random variables so, to be rigorous, the probability that they take a specific value is by definition 0. However, in order to simplify the explanation, we slightly abuse the terminology and use the term probability to give an intuitive idea of the behavior of the density function being defined.

Given the definition of the random variables X and Y, it is also easy to show that $f_X(d) = f_Y(d)$, so we can omit the name of the random variable and substitute them with the overall distance density $f(d)$ (see Section 2.3.4). Therefore the joint density is defined as:

$$f_{XY}(x, y) = f(x) \cdot f(y).$$

From a storage point of view, such an approach is feasible. The problem remains how to define an accurate transformation that produces the joint conditional density from the joint density. To achieve this we propose some heuristics and we prove through experimentation that they are very accurate. This is the topic of next sections.

4.5 Heuristics for an accurate measurement of the proximity

Given a metric space $\mathcal{M} = (\mathcal{D}, d)$ and two objects O_x and O_y of $\mathcal{D}$ with $d(O_x, O_y) = d_{xy}$, the space of possible distances $x = d(O, O_x)$ and $y = d(O, O_y)$, measured from an object $O \in \mathcal{D}$, is constrained by the triangular inequality, i.e. $x + y \geq d_{xy}$, $x + d_{xy} \geq y$, and $y + d_{xy} \geq x$ (see Section 2.3.1). Figure 4.2 helps to visually identify these constraints. In the gray area, called the *bounded area*, the triangular inequality is satisfied, while in the white area, called the *external area*, the triangular inequality is not satisfied, so an object O with such distances to O_x and O_y does not exist in $\mathcal{D}$.

In general, $f_{X,Y|D_{XY}}(x, y|d_{xy}) \neq f_{XY}(x, y)$, because the *joint density* $f_{XY}(x, y)$ gives the probability that the distances $d(\mathbf{O}, \mathbf{O_x})$ and $d(\mathbf{O}, \mathbf{O_y})$ are x and y, no matter what the distance is between $\mathbf{O_x}$ and $\mathbf{O_y}$. The difference between the two densities

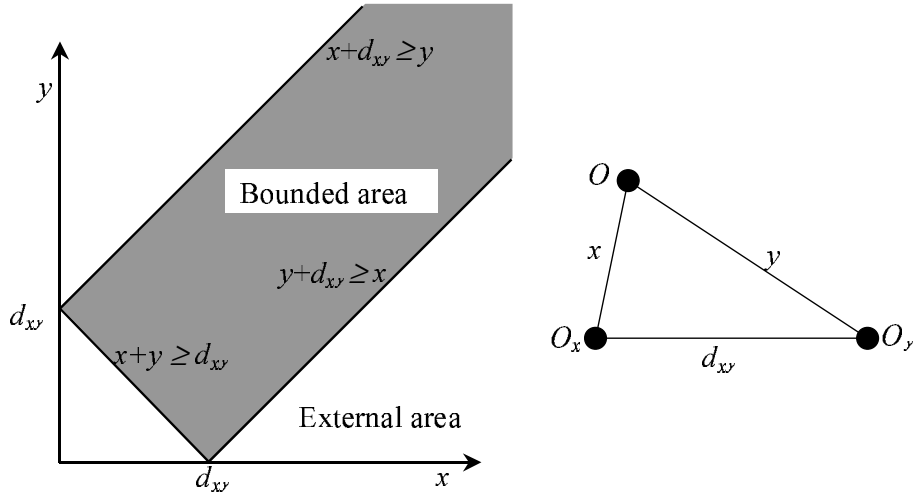
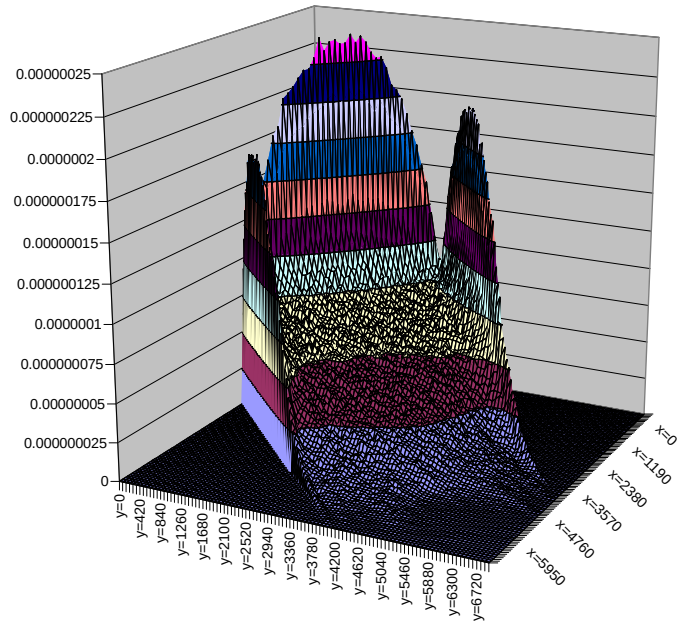


Figure 4.2: Area bounded by the triangular inequality

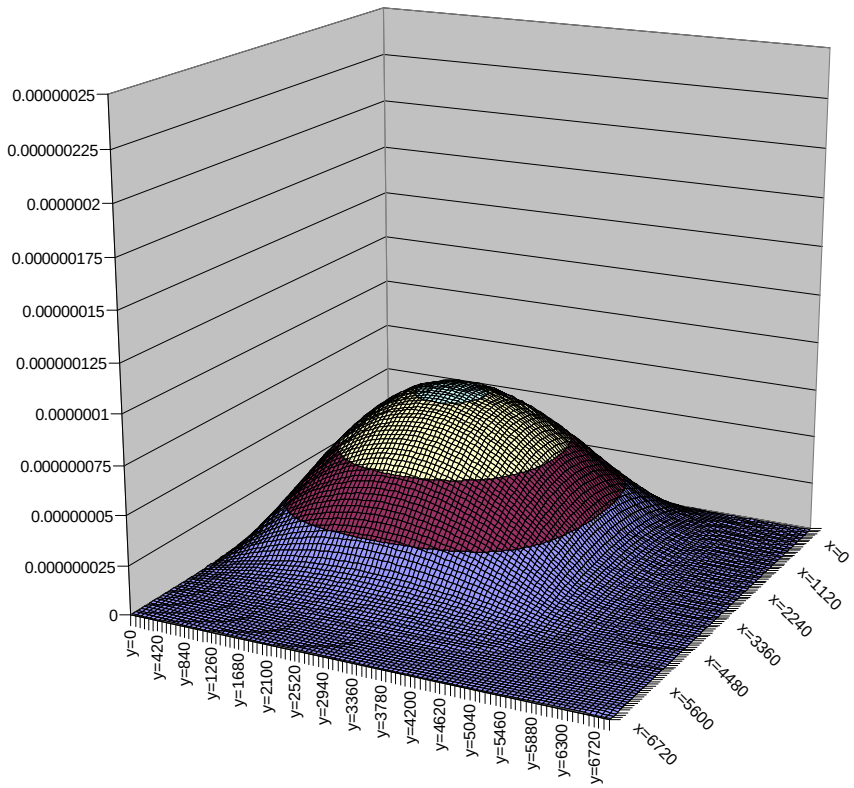
is immediately obvious when we consider the metric space postulates. Accordingly, $f_{X,Y|D_{XY}}(x, y|d_{xy})$ is 0 if x, y, and d_{xy} do not satisfy the triangular inequality, because such distances cannot simply exist. However, $f_{XY}(x, y)$ is not restricted by such a constraint, and any pair of distances $\leq d_m$ is possible. To illustrate this, Figure 4.3 shows the joint conditional density $f_{X,Y|D_{XY}}(x, y|d_{xy})$ for a fixed d_{xy} and the joint density $f_{XY}(x, y)$. Note that the graph of the joint conditional density has values greater than zero only in the bounded area, and that quite high values are located near the edges, while the joint density has values greater than zero also outside the bounded area. The graphs in Figure 4.3 are obtained using the two dimensional uniformly distributed data set UV described in Section 2.5.

4.5.1 Definition of the heuristics

Previous observations form the basis for the heuristics we propose to obtain the approximate joint conditional density $f^{appr}_{XY|D_{XY}}(x, y|d_{xy})$ by means of the joint density.



Joint conditional density



Joint density

Figure 4.3: Comparison between $f_{X,Y|D_{XY}}(x, y|d_{xy})$ and $f_{XY}(x, y)$

The intuitive idea can be outlined as follows:

> *Given d_{xy}, collect values of $f_{XY}(x, y)$ for x and y from the external area and put them inside the bounded area.*

When distances x, y, and d_{xy} satisfy the triangular inequality, the value of $f^{appr}_{XY|D_{XY}}(x, y|d_{xy})$ depends on the specific strategy used to implement the previous idea, otherwise, $f^{appr}_{XY|D_{XY}}(x, y|d_{xy}) = 0$. In this way, the integral over the bounded area is 1. This is the basic assumption of any probabilistic model that would be violated if the joint densities were simply trimmed out by the triangle inequality constraints.

We have tried four different implementations of this heuristic, varying the strategy applied to move density values. Figure 4.4 provides a visual representation of the methods, where the circles represent the joint density function, while the arrows indicate directions in which the necessary quantities are moved from the external area to the bounded area. The strategies can be briefly characterized as follows.

Orthogonal approximation Collect values of $f_{XY}(x, y)$ outside the bounded area and accumulate them on top of the corresponding constraint following a direction that is orthogonal to the constraint.

Parallel approximation Collect values of $f_{XY}(x, y)$ outside the bounded area and accumulate them on top of the corresponding constraint following a direction that is parallel to the axis.

Diagonal approximation Collect values of $f_{XY}(x, y)$ outside the bounded area and accumulate them on top of the corresponding constraint following a direction that always passes through d_m.

Normalized approximation Collect values of $f_{XY}(x,y)$ outside the bounded area to obtain a linear coefficient that modifies (increases) densities inside the bounded area.

Note that proximity, measured by using the $f^{appr}_{XY|D_{XY}}(x,y|d_{xy})$ defined according to the first three methods, Orthogonal, Parallel, and Diagonal, can also be obtained directly from the joint density $f_{X,Y}(x,y)$. In fact, instead of computing $f^{appr}_{XY|D_{XY}}(x,y|d_{xy})$ and integrating it, the same result can be obtained by integrating directly $f_{X,Y}(x,y)$ in the gray marked area, as illustrated in Figure 4.4: to simulate the gathering of values of $f_{XY}(x,y)$ outside the bounded area, required to obtain $f^{appr}_{XY|D_{XY}}(x,y|d_{xy})$, the integration is performed in the external area, covered by the gray marked areas, as well. Specifically we have the following,

$$X^{appr}_{d_{xy}}(r_x,r_y) = \int_0^{r_x}\int_0^{r_y} f^{appr}_{X,Y|D_{XY}}(x,y|d_{xy})dydx =$$

$$= \int_0^{b_x(d_{xy},r_x,r_y)} \int_{b_y^1(x,d_{xy},r_x,r_y)}^{b_y^2(x,d_{xy},r_x,r_y)} f_{X,Y}(x,y)dydx \qquad (4.5.1)$$

In the following, we simplify the terminology by omitting the d_{xy}, r_x, r_y parameters in the integration bounds and use only the symbols $b_x()$, $b_y^1(x)$ and $b_y^2(x)$. The integration bounds $b_x()$, $b_y^1(x)$, and $b_y^2(x)$ are functions that are specific for each approximation method. In particular, $b_x()$ gives the integration range along the x axis, while $b_y^1(x)$ and $b_y^2(x)$ form the lower and upper bounds of the gray area along the y axis for a specific x. A detailed definition of these integration bounds is given in next subsection.

The Normalized technique is even more simple because we only integrate in the bounded area restricted by the region radii (see the gray marked area in Figure 4.4) and we multiply the result by the normalization coefficient

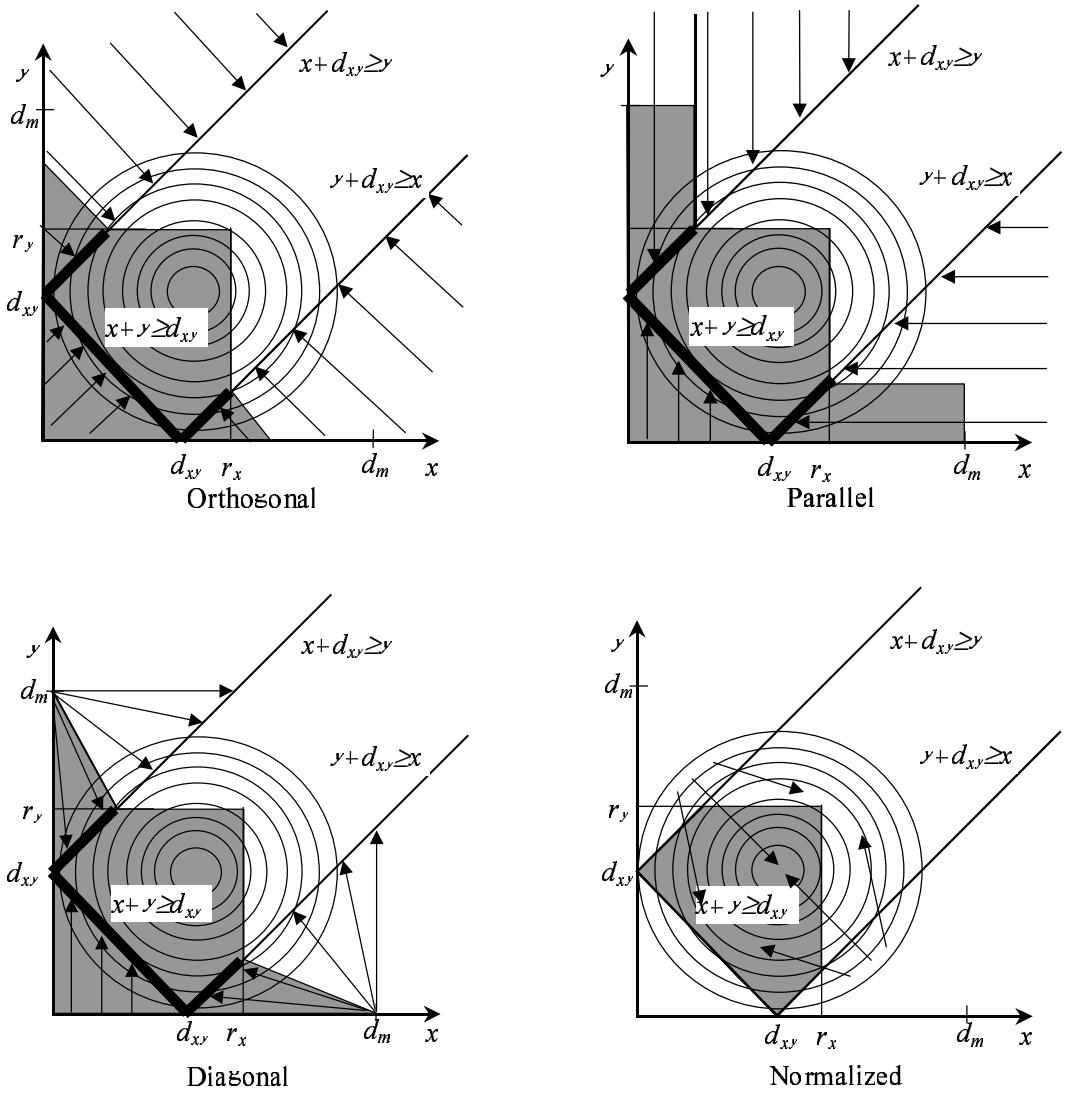


Figure 4.4: The four heuristics proposed to compute region proximity

$$NC(d_{xy}) = 1/(1 - E(d_{xy})), \tag{4.5.2}$$

where $E(d_{xy})$ is the integral of $f_{XY}(x, y)$ over the external area.

Integration bounds

In this section we formally define the bounding functions b_x, $b_y^1(x)$ and $b_y^2(x)$ of the four approximation methods described above. Even though the graphical representation of the integration areas seems to be easy and clear, its formalization is not straightforward, because several special cases should be taken into account to obtain the correct behaviour. Notice that in our simplified formalisation of the problem, the function $f_{XY}(x, y)$ can assume arguments outside the range $[0, d_m]$. In those cases we suppose that the returned value is 0.

We decompose the problem in subcases that can be considered separately – see Figure 4.5 as a convenient graphical reference. First, we distinguish two different cases: (i) $r_y < d_{xy}$ and (ii) $r_y \geq d_{xy}$. In these two situations, we identify some intervals along the x axis.

In case (i), we identify these three intervals:

1. $I'_1 = [0, d_{xy} - r_y)$,

2. $I'_2 = [d_{xy} - r_y, min(d_{xy} + r_y, r_x))$, and

3. $I'_3 = [min(d_{xy} + r_y, r_x), d_m]$.

In case (ii), we identify other three intervals:

1. $I''_1 = [0, min(r_y - d_{xy}, r_x))$,

2. $I_2'' = [min(r_y - d_{xy}, r_x), min(d_{xy} + r_y, r_x))$, and

3. $I_3'' = [min(d_{xy} + r_y, r_x), d_m]$.

Using the intervals defined above, we define the upper bound $b_y^2(x)$. First consider case (i) $r_y < d_{xy}$. When $x \in I_1'$, since the regions do not intersect (that is $d_{xy} - r_x > r_y$), the proximity is 0 (see Figure 4.5-a_3) so the upper bound $b_y^2(x)$ is 0 too. Otherwise, the upper bound is the straight line, which is specific for a method used, passing by point A shown in Figure 4.5-(a_1 and a_2). We call $a(x)$ that straight line. When $x \in I_2'$, the upper bound is always equal to r_y. When $x \in I_3'$, the upper bound is the minimum between the two, method specific, straight lines passing respectively, by point B_1 and B_2. We call $b(x)$ the straight line passing by B_1 and $c(x)$ the one passing by B_2. Figures 4.5-(a_1 and a_2) show two different situations. In the first case, the minimum is $b(x)$, in the other case, the minimum is $c(x)$.

Now consider case (ii) $r_y \geq d_{xy}$, that is there is always an intersection between the two regions. When $x \in I_1''$, the upper bound is the minimum between the two, method dependent, straight lines passing, respectively, by point A_1 and A_2. We call $d(x)$ the line passing by A_1 and $e(x)$ the one passing by A_2. Figures 4.5-(b_2 and b_3) show two different situations. In the first case the minimum is $e(x)$, while in the other case the minimum is $d(x)$. When $x \in I_2''$, the upper bound is always equal to r_y. Since I_3'' is defined exactly as I_3', the same arguments apply as can be seen in Figures 4.5-(b_1 and b_2).

More formally, $b_y^2(x)$ can be defined as follows:

$$b_y^2(x) = \begin{cases} \begin{cases} \begin{cases} a(x) & \text{if } d_{xy} - r_x \leq r_y \\ 0 & \text{elsewhere} \end{cases} & \text{if } x \in I_1' \\ r_y & \text{if } x \in I_2' \\ min(b(x), c(x)) & \text{if } x \in I_3' \end{cases} & \text{if } r_y < d_{x,y} \\ \begin{cases} min(d(x), e(x)) & \text{if } x \in I_1'' \\ r_y & \text{if } x \in I_2'' \\ min(b(x), c(x)) & \text{if } x \in I_3'' \end{cases} & \text{elsewhere} \end{cases} \tag{4.5.3}$$

where $a(x)$, $b(x)$, $c(x)$, $d(x)$, and $e(x)$ are defined for the individual approximation methods as follows.

Let us now specifically define the integration bounds $b_x()$, $b_y^1(x)$, and $b_y^2(x)$ for each specific approximation method, using previous observations.

In the Orthogonal method, to define $b_y^2(x)$, that is the upper bound of the integration area, the required straight lines $a(x)$, $b(x)$, $c(x)$, $d(x)$, and $e(x)$ are the following:

$$\begin{aligned} a(x) &= 2 \cdot r_y - d_{xy} + x \\ b(x) &= 2 \cdot r_y + d_{xy} - x \\ c(x) &= 2 \cdot r_x - d_{xy} - x \\ d(x) &= 2 \cdot r_x + d_{xy} - x \\ e(x) &= 2 \cdot r_y - d_{xy} - x \end{aligned}$$

The lower bound $b_y^1(x)$ of the integration area is defined as

$$b_y^1(x) = \begin{cases} \begin{cases} d_{xy} - 2 \cdot r_x + x & \text{if } d_{x,y} - r_x \leq r_y \\ 0 & \text{elsewhere} \end{cases} & \text{if } r_x < d_{x,y} \\ 0 & \text{elsewhere} \end{cases}$$

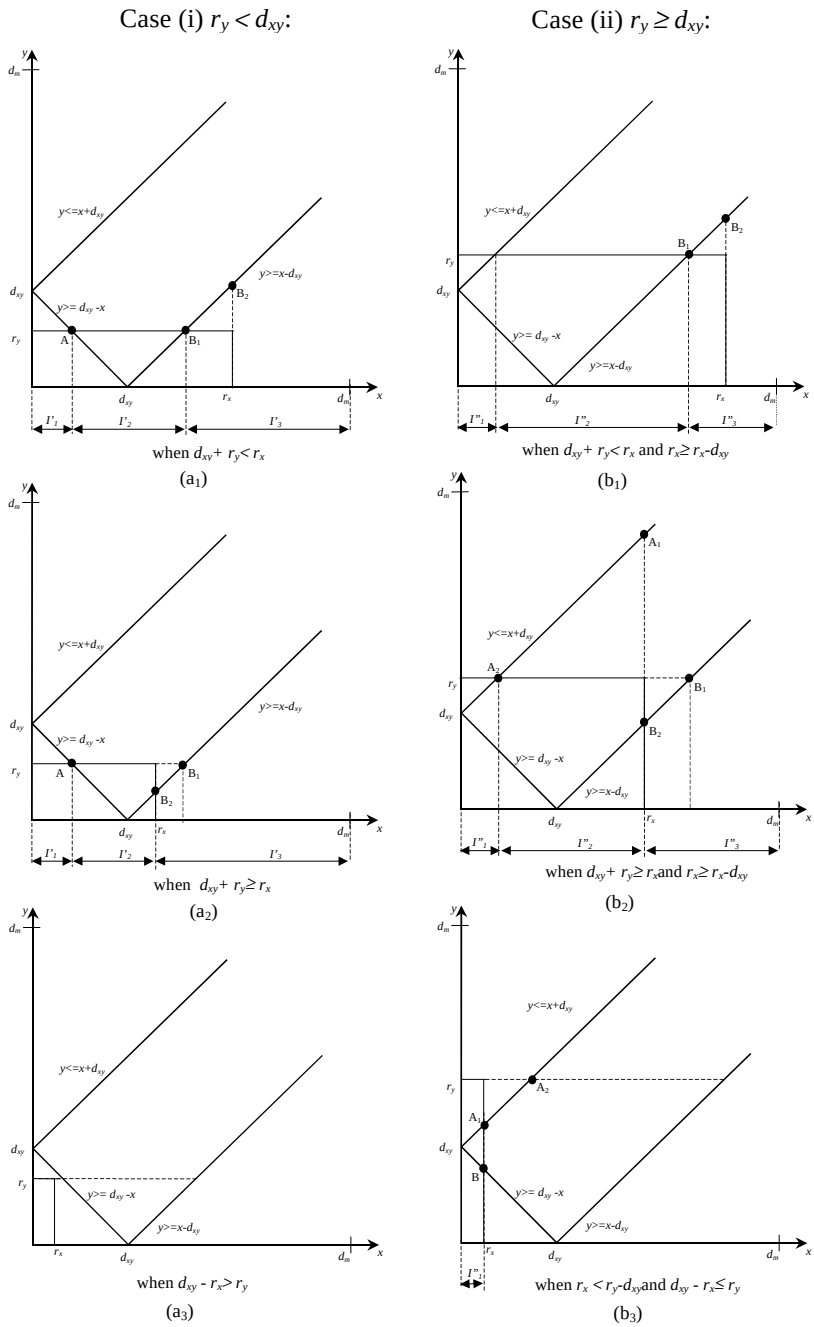


Figure 4.5: Cases to be taken into account when defining bounding functions

That is, when r_x is smaller than d_{xy}, and r_x is such that $d_{xy} - r_x \leq r_y$, the lower bound is the straight line $y = x + d_{xy} - 2 \cdot r_x$ passing by point B – see Figure 4.5-b$_3$. In all other cases the lower bound is 0.

Last, we need to define b_x, that is the range of the integration. If r_x is smaller than d_{xy} the integration is made in the interval $[0, r_x]$, otherwise in the interval $[0, min(r_1, r_2)]$, where r_1 and r_2 are such that, respectively, $b(r_1) = 0$ and $c(r_2) = 0$. We can make it explicit as follows:

$$b_x = \begin{cases} r_x & \text{if } r_x < d_{x,y} \\ min(2 \cdot r_y + d_{xy}, 2 \cdot r_x - d_{xy}) & \text{if } r_x \geq d_{x,y} \end{cases}$$

To define $b_y^2(x)$ for the Parallel method, we use again the outline of equation 4.5.3 and define $a(x)$, $b(x)$, $c(x)$, $d(x)$ and $e(x)$ as follows:

$$\begin{aligned} a(x) = b(x) &= r_y \\ c(x) &= r_x - d_{xy} \\ d(x) = e(x) &= d_m \end{aligned}$$

$b_y^1(x)$ is always 0 so:

$$b_y^1(x) = 0$$

Last, b_x is defined as follows:

$$b_x = \begin{cases} r_x & \text{if } r_x < d_{x,y} \\ d_m & \text{if } r_x \geq d_{x,y} \end{cases}$$

That is, if r_x is smaller than d_{xy}, then the integral is made in the interval $[0, r_x]$, otherwise in the interval $[0, d_m]$.

To define $b_y^2(x)$ for the Diagonal method, we use again the layout of equation 4.5.3 and define $a(x)$, $b(x)$, $c(x)$, $d(x)$ and $e(x)$ as follows:

$$\begin{aligned}
a(x) &= r_y \\
b(x) &= -\frac{r_y}{d_m - r_y - d_{xy}} \cdot x + \frac{r_y}{d_m - r_y - d_{xy}} \cdot d_m \\
c(x) &= -\frac{r_x - d_{xy}}{d_m - r_x} \cdot x + \frac{r_x - d_{xy}}{d_m - r_x} \cdot d_m \\
d(x) &= -\frac{d_m - d_{xy} - r_x}{r_x} \cdot x + d_m \\
e(x) &= -\frac{d_m - r_y}{r_y - d_{xy}} \cdot x + d_m
\end{aligned}$$

Bounding functions $b_y^1(x)$ and b_x for the Diagonal method are defined exactly the same as for the Parallel method so we make no further discussion on them.

Last, we have to consider the Normalized method. Because of the different nature of this method, also the definitions of the bounding functions have a different layout. In particular, there is not need for extending the integration area, so just the intersection between the original constraints ($x \leq r_x$ and $y \leq r_y$) of the integration area and the triangular inequality constraint are needed. The result is the following:

$$b_x = r_x$$

$$b_y^1(x) = |x - d_{xy}|$$

$$b_y^2(x) = x + d_{xy}$$

4.5.2 Computational complexity of the heuristics

The computational cost of Equation 4.5.1 is clearly $O(n^2)$, where n is the number of samples needed for one integration. Since one of our major objectives is efficiency, such a cost is still high. However, we can transform the formula as follows:

$$\int_0^{b_x()} \int_{b_y^1(x)}^{b_y^2(x)} f_{XY}(x,y)dydx = \int_0^{b_x()} \int_{b_y^1(x)}^{b_y^2(x)} f(x)f(y)dydx = \\ = \int_0^{b_x()} f(x) \cdot (F(b_y^2(x)) - F(b_y^1(x)))dx \qquad (4.5.4)$$

Provided that density $f(d)$ and distribution $F(d)$ functions are explicitly maintained in the main memory, Equation 4.5.4 can be computed with complexity $O(n)$. This assumption is realistic even for quite high values on n, so the computational complexities of the Orthogonal, Parallel and Diagonal methods are linear. As far as the Normalized method is concerned, we can see that the normalization coefficient, defined by Equation 4.5.2, is not restricted by specific region radii, thus it only depends on d_{xy}. Such information can also be maintained in the main memory. Consequently, the computational complexity of the Normalized method is also $O(n)$.

4.6 Validating the approaches to the proximity measure

In this section, we investigate the accuracy of the proposed approaches to computing the proximity. Before presenting the simulation results, we first describe the evaluation process and define comparison measures.

4.6.1 Experiments and comparison measures

We computed the *actual proximity* $X_{d_{xy}}^{actual}(r_x, r_y)$ for all data sets described in Section 2.5 as follows. We uniformly chose 100 values of d_{xy}, r_x, and r_y, in the range of possible distances. The proximity $X_{d_{xy}}^{actual}(r_x, r_y)$ was computed for all possible combinations of the chosen values. To accomplish this task, we found for each d_{xy} 400 pairs of objects (O_x, O_y), i.e. the centers of the balls, such that $|d(O_x, O_y) - d_{xy}| \approx 0$. For each pair of objects, we used the predefined values of r_x and r_y to generate ball regions. Then, we only considered pairs of intersecting regions, because non intersecting balls have 0 proximity and no verification is needed. For each pair of balls, we counted the number of objects in their intersection. The actual proximity was finally obtained by computing the average number of objects in the intersection for each generated configuration of d_{xy}, r_x, and r_y and by normalizing (dividing by the total number of objects in the data set) such values to obtain the probability.

We did not consider distances d_{xy} of very low densities, because it was not possible to compute the actual proximity with sufficient precision – the data sets contained only very few objects at such distances.

Once the actual proximity was determined, we computed the approximate proximity for the same values of d_{xy}, r_x, and r_y. The comparison between the actual and the approximate proximity was quantified for each possible configuration as the *absolute error*

$$\epsilon(r_x, r_y, d_{xy}) = |X_{d_{xy}}^{actual}(r_x, r_y) - X_{d_{xy}}^{appr}(r_x, r_y)|$$

An alternative way to evaluate our approaches would be to use the *relative error*, defined as the ratio of the absolute error and the actual proximity. However, our choice for the absolute error can be justified as follows. Suppose that the actual

proximity is almost 0 (e.g. 10^{-5}), while the approximate proximity is exactly 0. In this case, the relative error is 1, i.e. we have a high error. Consider now the opposite case where the actual proximity is zero, and our approximation is 10^{-5}. In this case, the relative error is ∞. However, given the meaning of proximity (see Section 4.1 and Section 4.3), and considering previous examples, we can say that such an approximation is good, because it almost produces the correct results. For example, when it is applied for approximate similarity search, as discussed in section 6.9, regions can be safely pruned if the proximity is 10^{-5} because, statistically, only one object in 100,000 can be lost. This means that the absolute error is a more objective measure, i.e. more suitable for our purposes.

Given the large number of results, we summarized them by computing the average error $\epsilon'_\mu(d_{xy})$ for all pairs of radii at a given distance between the centers, and the average error $\epsilon''_\mu(r_x, r_y)$ for all distances between the ball centers at a given pair of radii, specifically:

$$\epsilon'_\mu(d_{xy}) = Avg_{\mathbf{r_x r_y}}(\epsilon(\mathbf{r_x}, \mathbf{r_y}, d_{xy}))$$

and

$$\epsilon''_\mu(r_x, r_y) = Avg_{\mathbf{d_{xy}}}(\epsilon(r_x, r_y, \mathbf{d_{xy}})).$$

In a similar way, we computed the variance of the error for a given distance d_{xy}:

$$\epsilon'_\sigma(d_{xy}) = Var_{\mathbf{r_x r_y}}(\epsilon(\mathbf{r_x}, \mathbf{r_y}, d_{xy}))$$

The evaluation of ϵ'_μ is used to measure the average error of approximations for specific distances between the ball centers. However, ϵ'_μ alone is not sufficient to correctly judge the quality of the approximation. In fact, it is obtained as the average error for all possible values of r_x and r_y so that some peculiar behaviors may remain hidden.

In this respect, the stability of the error must also be considered. For this purpose, we computed the variance ϵ'_σ. Note that high average errors and small variances may also provide good approximations. To illustrate this, suppose that we want to use the proximity to order (rank) a set of regions with respect to a reference region. The ranking results obtained through the actual and approximate proximity may turn out to be identical even though ϵ'_μ is quite high. In fact, when the variance of error is very small, it means that the error is almost constant, and the approximation somehow follows the behavior of the actual proximity. In this case, it is highly probable that the approximated proximity increases (or decreases) according to the trend of the actual one, thus guaranteeing the correct ordering.

On the other hand, ϵ''_μ represents the average error from a different point of view and complements ϵ'_μ. It is determined for a given pair of radii (r_x, r_y) by varying d_{xy}. This measure offers a finer grained view on the error behavior, since the average is only computed varying the distance d_{xy}.

4.6.2 Discussion on the experimental results

For all data sets, the actual proximity was compared with our techniques and the trivial proximity (defined by Equation 4.2.1). Figures 4.6, 4.7, and 4.8 presents the average error ϵ'_μ and its variance ϵ'_σ. Note that all the approximation methods outperform the trivial one, and the error of the trivial method may even be one order of magnitude higher. The same holds for the variance of the errors. For all the proposed techniques, ϵ'_σ is one order of magnitude smaller than the value obtained with the trivial technique. This implies that the trivial proximity may provide results that significantly differ from the actual proximity. In addition, the proposed methods

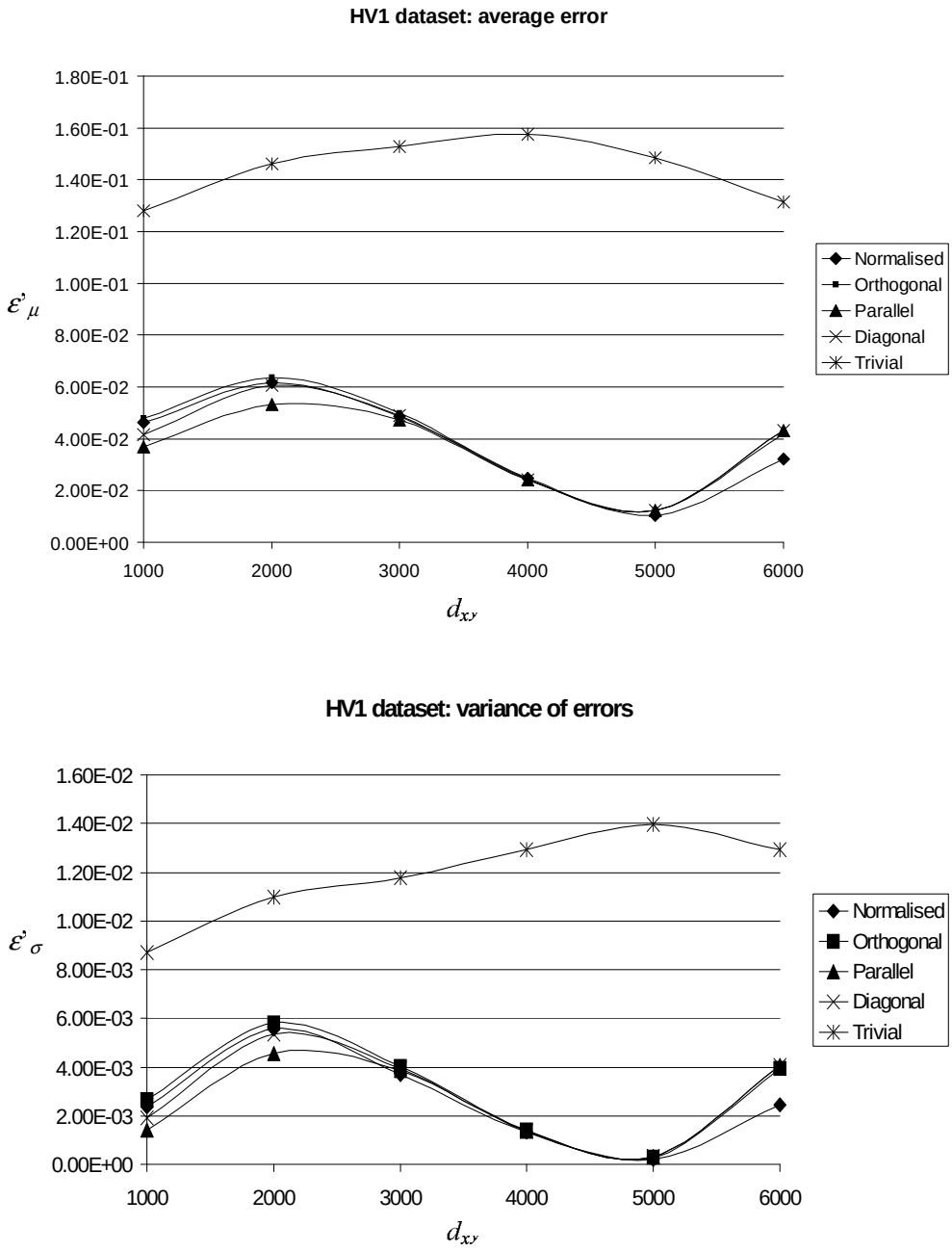


Figure 4.6: Average and variance of errors given d_{xy} in HV1

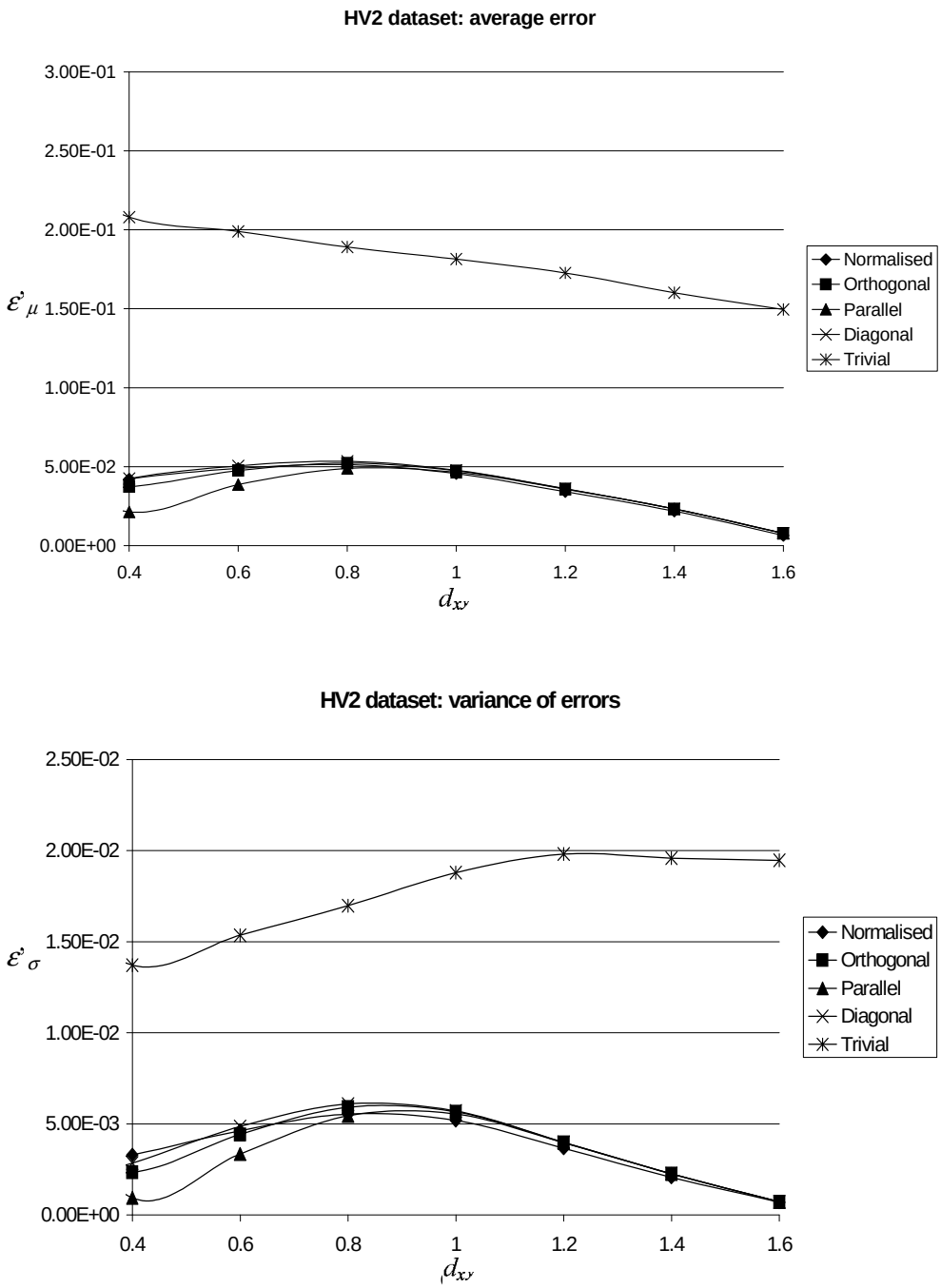


Figure 4.7: Average and variance of errors given d_{xy} in HV2

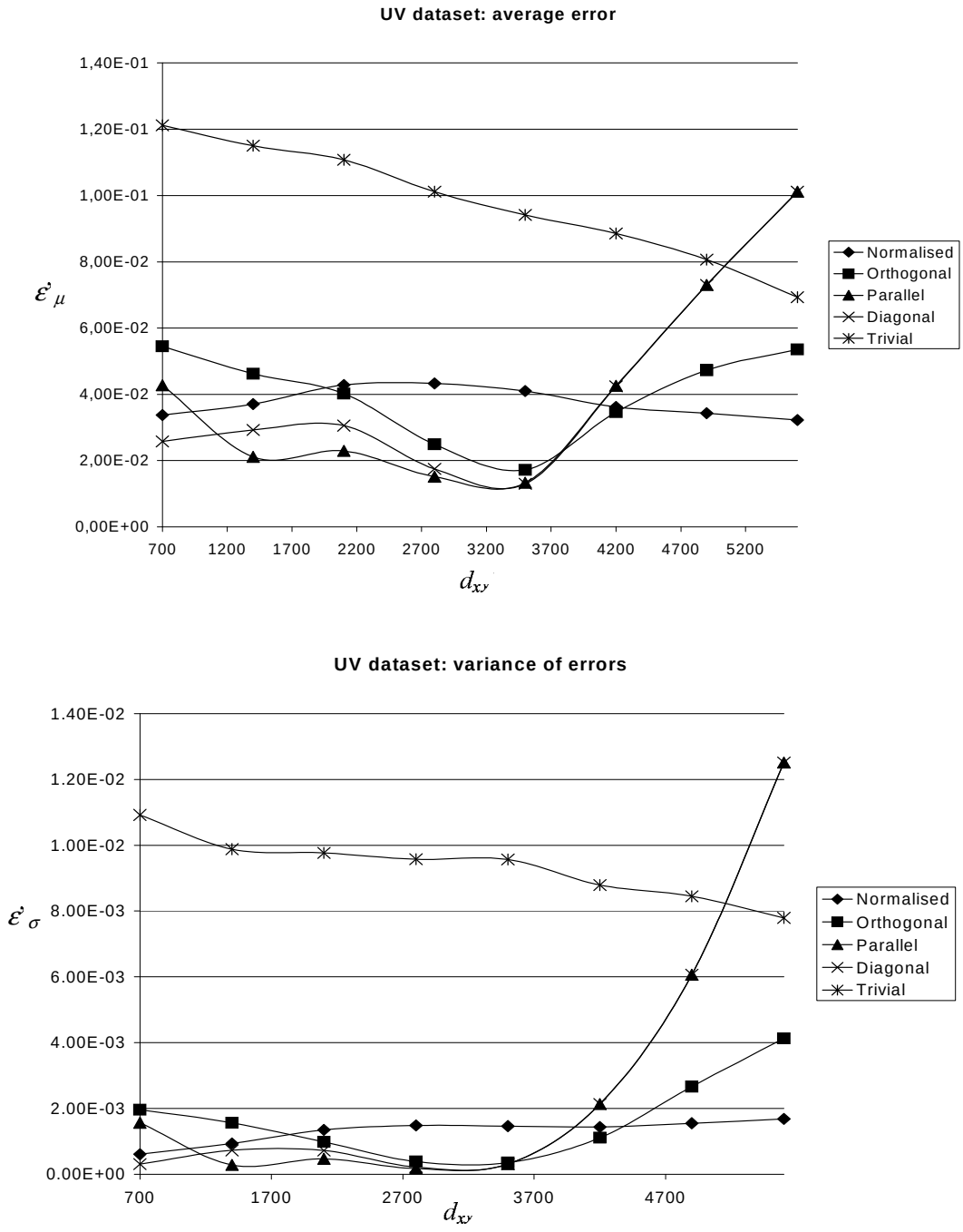


Figure 4.8: Average and variance of errors given d_{xy} in UV

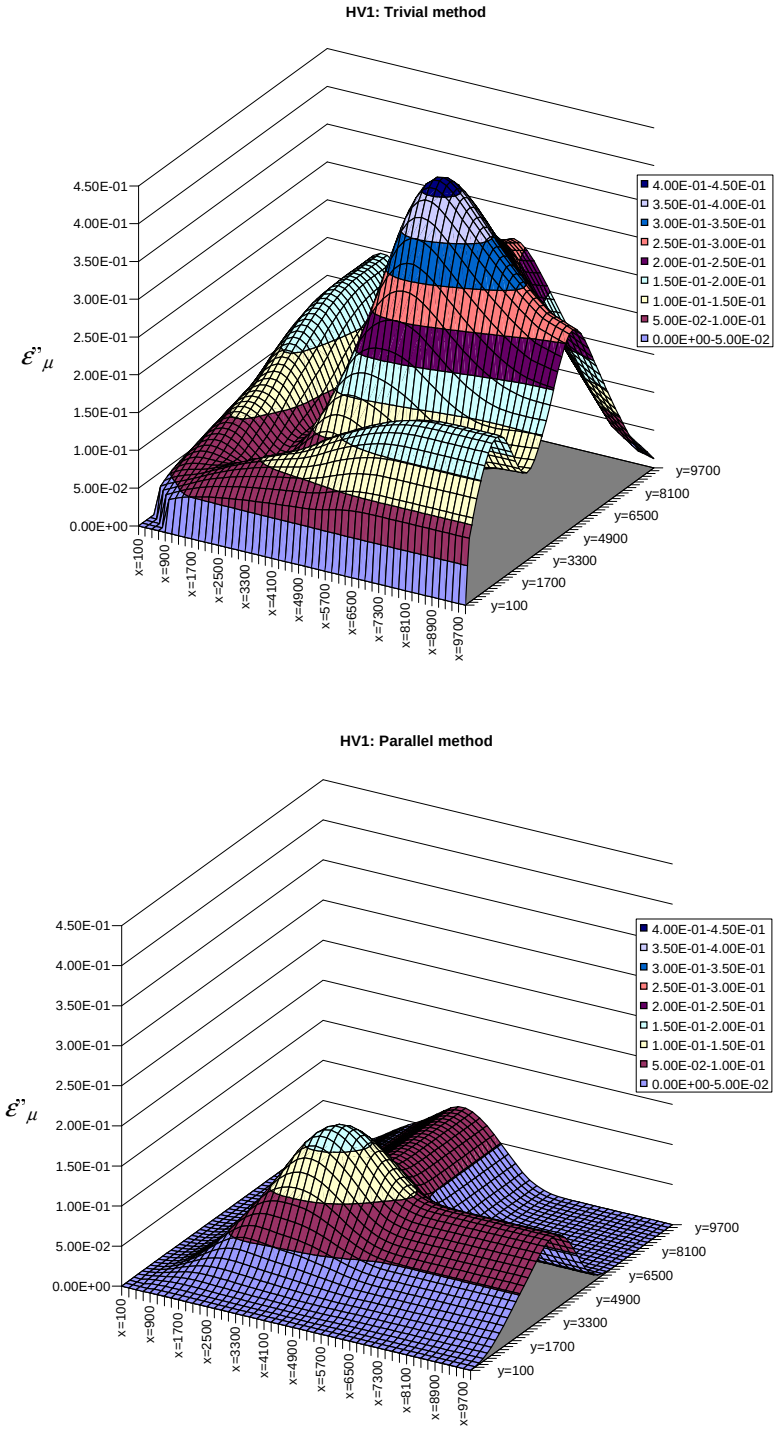


Figure 4.9: Comparison between the errors of the trivial method and the parallel method given r_x and r_y in HV1

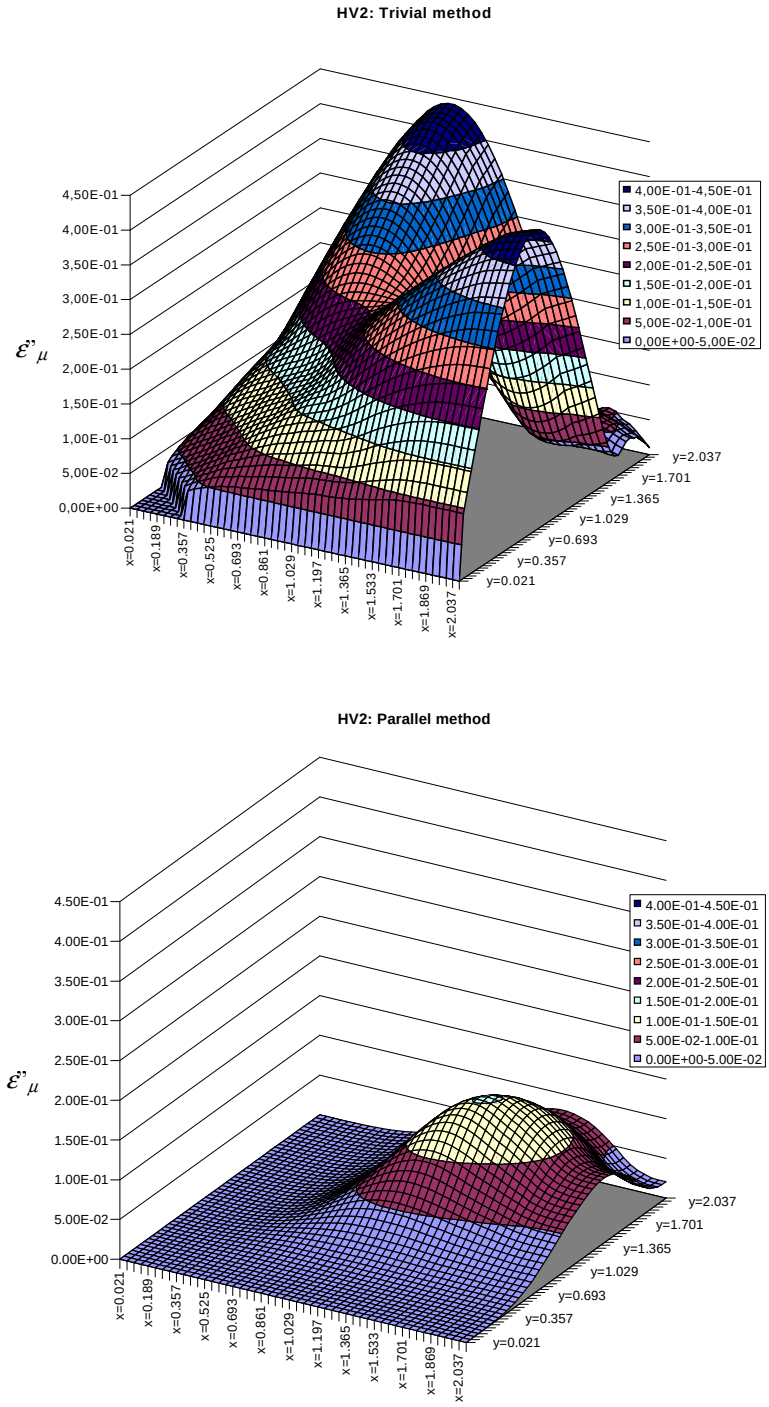


Figure 4.10: Comparison between the errors of the trivial method and the parallel method given r_x and r_y in HV2

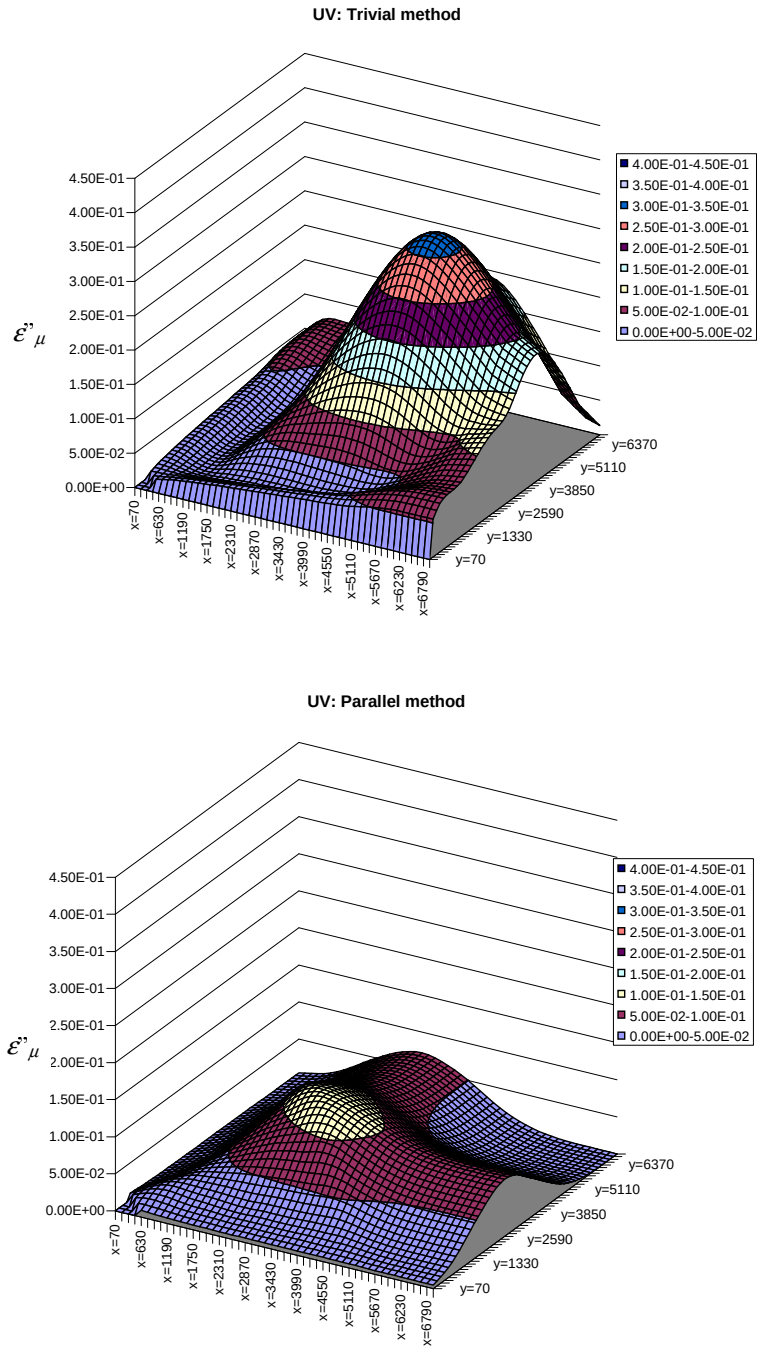


Figure 4.11: Comparison between the errors of the trivial method and the parallel method given r_x and r_y in UV

provide very good and stable results. They have a small variance as well as small errors, so that they can be reliably used in practice.

Although there is not a clear winner among the proposed methods, the Parallel methods gives the best results in the most frequently used range of distances. If we compare the proposed methods for the UV data set, we can see that the Parallel method provides good and stable results. The quality of this method deteriorates, both in terms of ϵ'_μ and ϵ'_σ, for high values of d_{xy}, which are not likely to occur in practice. Here the best results are obtained through the Normalized method. In the HV1 and HV2 data sets we can see again that the Parallel method provides the best performance, though the differences with respect to the other techniques described are even less significant.

Consider now the average error for a given pair of radii ϵ''_μ. For the sake of simplicity, we only compare ϵ''_μ for the Parallel and the trivial method. The results are shown in Figures 4.9, 4.10, and 4.11. As an additional confirmation of the observation that we have made for ϵ'_μ and ϵ'_σ, the error ϵ''_μ for our approximations is again significantly smaller than the one measured for the trivial method. In particular, the error of the trivial method is always quite high, while for a substantial range of r_x and r_y values, the error of the Parallel method is close to 0.

Chapter 5

Approximate similarity search

5.1 Introduction

Similarity searching has become fundamental in a variety of application areas, including multimedia information retrieval, data mining, pattern recognition, machine learning, computer vision, computational biology, data compression, and statistical data analysis. Though a lot of work has been done to develop structures able to perform similarity search fast, results are still not satisfactory, and much more investigation is needed. Accordingly, approximate similarity search has emerged as a relevant research topic. The idea of this promising approach is that a high improvement of efficiency can be obtained at the price of some controlled imprecision in the results of a query. In the following first an introduction to the issue of approximate similarity search is given, then some of the most promising approaches proposed so far are described.

5.2 Approximate similarity search issues

As discussed in Chapter 3, in order to increase efficiency, tree-based access methods create a partition of searched data set, and bound elements (set of objects) of such partition in regions [Gut84, BKK96, BÖ97, CPZ97, BÖ99, TTSF00]. Each node of the tree corresponds to a set of objects. Consider the example in Figure 5.1. The partition contains three subsets, distinguished respectively by white, black, and gray points. The subset corresponding to white points is bounded by region $\mathcal{R}_1$, the subset corresponding to black points is bounded by region $\mathcal{R}_2$, and the subset corresponding to gray points is bounded by region $\mathcal{R}_3$. When a similarity query should be processed, only nodes bounded by regions overlapping the query region should be accessed, saving a lot of disk accesses and distance computations.

However, access methods typically suffer from the so called *dimensionality curse* problem. It has been observed that, when the number of dimensions of a data set is greater than 10-15, performance of access methods decreases and a linear scan over the whole data set would perform better [BGRS99, WSB98]. One consequence of the dimensionality curse is that the probability of overlaps between the query and data regions is very high and the execution of a similarity query may require to access many of the data regions loosing the advantage of any indexing structure what so ever. Indeed all data regions that overlap the query region must be accessed. For instance, in Figure 5.1, the query region overlaps regions $\mathcal{R}_1$, $\mathcal{R}_2$ and $\mathcal{R}_3$ so all of them should be accessed to answer to the query.

Given this inefficiency problem other techniques are being investigated. Here we discuss the *approximate similarity search* approach [AMN+98, PAL99, CP00], that has recentely emerged as important research issue. The idea behind approximate

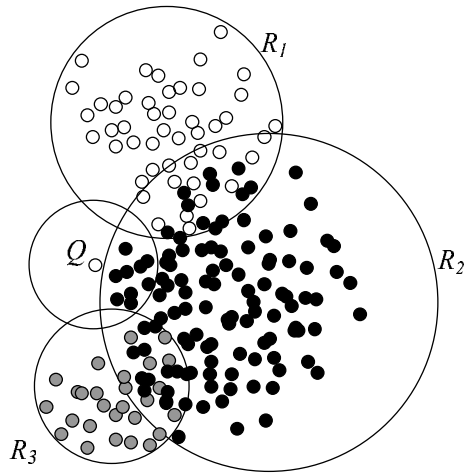


Figure 5.1: Partitions, data regions, and query regions

similarity search is that queries are processed faster at the price of some imprecision in the results. The approximation is obtained by relaxing some constraint at the basis of "exact" similarity search algorithms.

Approximate similarity search is also encouraged by other observations. It can be noticed that the nature of the similarity based search process is intrinsically interactive. Users, typically, issue several similarity queries, to the search system, eventually reusing results of current queries to express new ones. For instance, the user may start using an initial image, to search for similar ones, then uses images returned by the query to issue new similarity search queries. In this case, efficient execution of elementary queries is even more important and users may accept some imprecision, in the temporary results, at the price of faster responses. Furthermore, some controlled imprecision in the result of a similarity search query may not be noticed by users or will be accepted when the increase in performance obtained is high. In fact, similarity is an intuitive and subjective measurement. People, for instance, judge the similarity between two images differently. However, this subjectivity is lost when similarity is

defined by a mathematical formula.

5.3 Survey of existing approaches to approximate similarity search

In the following, we present a short survey of the most important approaches to the approximate similarity search in order to highlight our contribution. Approaches to approximate similarity search can be classified into two broad categories [FTAA01]: (1) approaches able to reduce the size of data objects, and (2) approaches able to reduce the data set that needs to be examined. In the following we discuss these two categories. Specifically, approaches belonging to the second one will be treated more deeply since the techniques that we propose in this thesis belong to it.

5.3.1 First category: approaches able to reduce the size of data objects

The first category is mainly based on dimensionality reduction. Typically linear-algebraic methods such as Karhunen-Loeve Transformation (KLT) [Fuk90], Discrete Fourier Transform (DFT) [OS89], Discrete Cosine Transform (DCT) [Kai85], or Discrete Wavelet Transform (DWT) [Cas96] are used. All these methods assume that a few dimensions are enough to retain the most important information regarding the represented data objects so the other dimensions can be simply ignored.

Another approach in this category is the VA-file [WSB98], also introduced in Section 3.2.1. It reduces the size of multidimensional vectors by quantizing the original data objects. In this approach, nearest neighbor search is performed in two steps. In

the first step, the approximated vectors are scanned identifying candidate vectors. In the second step, the found candidates are visited in order to find the actual nearest neighbors. A modification to VA-files was proposed in [FTAA00], where the creation of a VA-file is improved by first transforming the data using KLT in a more suitable domain. Though the improvement of such techniques is significant, they only work for vector spaces.

An interesting approach also falling in this category, which can also be applied to non vector data, is the FASTMAP [Fal96]. It supposes to have n objects and an $N \times N$ distance matrix. FASTMAP tries to project these objects in a dim dimensional vector space, by only using the information given by the distance matrix, in such a way that distances are preserved. Of course, Euclidean distance between points in the vector space approximates the real distance between objects defined in the matrix. The quality of the approximation depends on the number of dimensions of the target vector space and on the specific distance matrix. The idea behind FASTMAP is to project objects on a specific line passing by two pivot objects in a dim dimensional vector space. A special heuristic is used to carefully select the two pivot objects.

5.3.2 Second category: approaches able to reduce the data set that needs to be examined

Algorithms of the second category use strategies to reduce the data set that needs to be examined. These can be further classified depending on the specific strategy used to achieve their goal:

Relaxed branching strategies This strategies can be used with access methods based on hierarchical decomposition of the space. Their aim is not to access

regions when they are not likely to contain desired results, or when access would only marginally improve the existing results. In this case, an approximate *pruning condition* is used to decide if a region should be accessed or not.

Early termination strategies In this case, search algorithms are prematurely stopped when current result is judged to be satisfying the approximation requirements. This strategies uses a *stop condition* to decide if it is time to stop the algorithm – the search terminates as soon as the chances for obtaining significantly better results become low. Here the hypothesis is that after some steps of search iteration, good approximation is obtained while further improvement is of minor importance and consume most of the total search costs.

Our algorithms for approximate similarity search belongs to this category since they aim at reducing the portion of the data set needed to be examined in order to find the result. Two of them use relaxed branching strategies, the other two adopt early termination strategy. Accordingly, they relay on an accurate definition of either a pruning condition or a stop condition respectively.

In the following we discuss more in details some of the most authoritative approaches belonging to this category.

5.3.3 Approximate nearest neighbors searching using BBD trees

Suppose to have a set of points $\mathcal{DS}$ in a *dim* dimensional vector space and a query object O_q. Let O_N be the nearest neighbor of O_q, and O_A some other object in the searched collection. Given $\epsilon > 0$, provided that $0 < d(O_N, O_q) \leq d(O_A, O_q)$, O_A is an

(1+ϵ)-nearest neighbor of O_q if

$$\frac{d(O_A, O_q)}{d(O_N, O_q)} \leq 1 + \epsilon$$

That is, O_A is within relative error ϵ of the true nearest neighbor. This idea can be generalized to the case of the j-th nearest neighbor of O_q, for $1 \leq j \leq n$, where n is the size of the database. Using respectively O_A^j and O_N^j to designate, the j-th approximate and nearest neighbor, the constraint should be modified as follows

$$\frac{d(O_A^j, O_q)}{d(O_N^j, O_q)} \leq 1 + \epsilon.$$

If this constraint is satisfied, O_A^j is called the *(1+ϵ)-j-approximate nearest neighbor* of O_q.

The algorithm proposed in [AMN+98], given a data set $\mathcal{DS}$ represented in a vector space with distances measured using functions of the Minkowski family, and given a query object O_q, guarantees to find an (1+ϵ)-approximate nearest neighbor of O_q in $O(\log n)$ time. Alternatively, when k-nearest neighbors search is considered, the algorithm guarantees to find k (1+ϵ)-k-approximate nearest neighbors[1] of O_q in $O(k \log n)$ time. The parameter ϵ can be used to control the tradeoff between efficiency and quality of the approximation. The higher ϵ, the higher the performance, the higher the error.

This algorithm uses as underlying indexing structure a so called *balanced box-decomposition (BBD) tree* that is a variant of a quad-Tree [Sam95] and is similar to other balanced structures based on box-decomposition [BET95, Bes95, CK95]. Specifically, this structure is based on a hierarchical decomposition of the space where

[1]Notice that also one of the techniques proposed in this thesis is designed in such a way that it returns k (1+ϵ)-k-approximate nearest neighbors, see Section 6.6 and [ZSAR98].

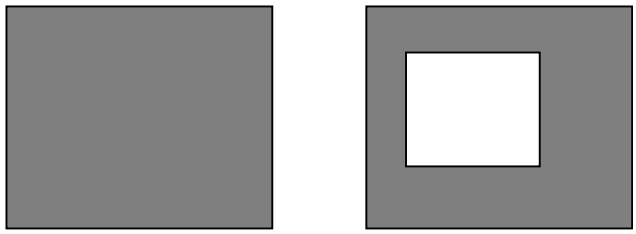


Figure 5.2: Possible regions in a BBD tree: a) *dim*-dimensional rectangle and b) set theoretic difference of two rectangles.

regions are represented by nodes of a tree that has $O(\log n)$ height. Each node of the tree is associated with a region and points to other nodes. Regions associated with nodes pointed by the same parent node do not overlap each other. There are two types regions in a BBD tree. A region can be a *dim*-dimensional rectangle or the set theoretic difference of two rectangles, one enclosed within the other, as depicted in Figure 5.2 respectively by picture a) and b). These rectangles are *fat*, in the sense that the ratio between the longest and the shortest sides is bounded. The tree has $O(n)$ nodes and it can be built in $O(dim \cdot n \cdot \log n)$ time, where n is the size of the data set. Each region associated with a leaf node contains a single object. The set of leaf regions defines a partition of the space.

The nearest neighbor algorithm on this data structure is intuitively defined as follows. Given a query object O_q, the tree is traversed and the leaf node associated with the region containing the query is found. Given the properties of the BBD trees, the leaf node is found in $O(\log n)$ and there is only one leaf node that contains it. At this point, a *priority search* is performed by enumerating leaf regions in increasing order of distance from the query object. The distance from an object O to a region is computed as the distance of O to the closest point that can be contained in the

region. When a leaf region is visited, the distance of the associated object from O_q is measured and the closest point seen so far is recorded. Let us call O_A such a current closest point. The algorithm stops when current leaf region is such that its distance is larger than $d(O_q, O_A)$, that is, the current region cannot contain objects whose distance from the query object is shorter than that of O_A. Since all remaining leaf regions are farther than current region, it means that O_A is the nearest neighbor to O_q.

The approximate nearest neighbor algorithm uses a stop condition to prematurely stop the search algorithm. Specifically, the algorithm stops as soon a the distance to the current leaf region exceeds $d(O_q, O_A)/(1+\epsilon)$. It is easy to show that in these circumstances O_A is a $(1+\epsilon)$-approximate nearest neighbor. To clarify the behavior of the exact and approximate nearest neighbor search algorithms consider Figure 5.3. Data objects are represented by black spots. Each object is included in a rectangular region associated with a leaf node. Each region is identified by a number assigned incrementally according to the distance of the region from the query object O_q. Thus, region 1 is the closest to O_q, in fact it contains O_q, while region 10 is the farthest. The search algorithm starts from region 1 to collect the potential nearest object to O_q. In the figure, we suppose that region 3 was accessed and object O_A was found as the current closest object. The circumference in the figure has in fact radius equal to $d(O_q, O_A)$. The exact algorithm would continue accessing regions that overlap the circumference and it would stop after accessing region 10, which contains the exact nearest neighbor. The approximate algorithm, on the other hand, would access only regions that overlap the dotted circumference, which has a radius equal to $d(O_q, O_A)/(1+\epsilon)$. Therefore it would stop after accessing region 8, missing the

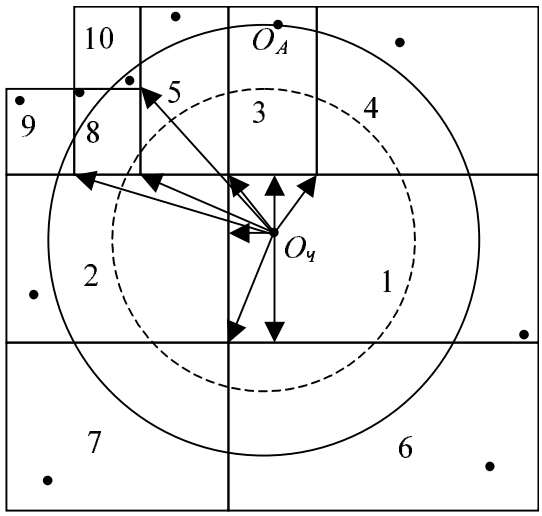


Figure 5.3: Overview of the approximate nearest neighbors search algorithm using BBD trees

exact nearest neighbor.

The priority search can be performed in $O(\log n)$ times the number of regions that are visited, by using an auxiliary heap. Let m be the number of regions visited, m has an upper bound depending only on dim and ϵ, for any Minkowski metric, defined as $\lceil 1+6dim/\epsilon \rceil^{dim}$. This bound arises from the maximum number of disjoint dim-cubes of diameter $\epsilon/3$ that can intersect a dim-ball region of radius 1. Since the upper bound can be considered a constant, provided that dim and ϵ are fixed, the algorithm finds the $(1+\epsilon)$-approximate nearest neighbor in $O(\log n)$ time.

The algorithm can be easily extended to the case of k nearest neighbors. In fact, in this case the current k closest objects are recorded and the algorithm stops when the distance of the current k-th closest object O_A^k is such that $d(O_q, O_A^k)/(1+\epsilon)$. In this case O_A^k is a $(1+\epsilon)$-k-approximate nearest neighbor. In this algorithm, the upper bound for the number of accessed regions m is $2k + \lceil 1 + 6dim/\epsilon \rceil^{dim}$. Therefore, provided that dim and ϵ are fixed, the algorithm finds the $(1+\epsilon)$-k-approximate

nearest neighbor in $O(k \log n)$ time.

Notice that upper bound m does not depend on the size of the data set n. However, it depends exponentially on *dim* so this algorithm can be practically used only on vector spaces with a small number of dimensions, e.g. in the range from 2 to 20. This algorithm can only be used in case of data represented in a vector space with distances measured by using Minkowski metrics. In particular, it cannot be used in case of data represented in a generic metric space.

5.3.4 Approximate range searching using BBD trees

The use of BBD trees, briefly described in previous section, was also exploited to design an approximate range search algorithm. This approach was proposed in [AM95]. Let us suppose to have a data set $\mathcal{DS}$ of objects defined in a *dim* dimensional vector space, and to use a function of the Minkowski family to measure distances among objects. Let us also suppose to have a range query $\mathbf{range}(O_q, r_q)$ defined by the ball region $\mathcal{B}(O_q, r_q)$. Typically, the goal of a range query is to retrieve the set of objects of $\mathcal{DS}$ included in $\mathcal{B}(O_q, r_q)$. Alternatively, in some cases one may just need to count the number of objects that qualify for the range query or, more generally, we may suppose that objects of $\mathcal{DS}$ have been assigned a weight, and the goal is to compute the accumulated weight $weight(\mathbf{range}(O_q, r_q))$ of the objects that qualify for the query. Retrieving the set of objects included in a range query has in general a cost higher than counting them, however in this work only the counting version of the problem was considered.

The exact range search algorithm returns the exact weight of the result set, while the approximate range search algorithm returns a weight that approximates the real

weight of the actual result set. The idea of this approximation technique is to consider range queries as *fuzzy* range queries. In a fuzzy range query objects that are "close" to the boundary of the query may or may not be included in the count.

Given a range query defined by the ball region $\mathcal{B}(O_q, r_q)$, and given a real value $\epsilon > 0$, regions $\mathcal{B}^-(O_q, r_q/(1+\epsilon))$ and $\mathcal{B}^+(O_q, r_q \cdot (1+\epsilon))$ can be defined. Regions $\mathcal{B}^-$ and $\mathcal{B}^+$ have the same center of $\mathcal{B}$, however, their radius is respectively reduced or increased of a factor $1+\epsilon$. In the following, for sake of simplicity, by using $\mathcal{B}^-$, $\mathcal{B}$, and $\mathcal{B}^+$ we also refer to the set of objects of $\mathcal{DS}$ respectively included by these regions.

The value $weight(\mathcal{R}^A)$, where $\mathcal{R}^A \subseteq \mathcal{DS}$, is a *legal answer* to an *(1+ϵ)-approximate range query* when

$$\mathcal{B}^- \subseteq \mathcal{R}^A \subseteq \mathcal{B}^+.$$

That is, all objects of $\mathcal{B}^-$ should be included, all objects of $\mathcal{DS} \setminus \mathcal{B}^+$ should not be included, and objects of $\mathcal{B}^+ \setminus \mathcal{B}^-$ may or may not be included.

Notice that this approach allows for false dismissals and false hits. In fact, when some objects of $\mathcal{B} \setminus \mathcal{B}^-$ are not included false dismissals occur, while when some objects of $\mathcal{B}^+ \setminus \mathcal{B}$ are included false hits occur.

Range queries can be answered by using search algorithms on BBD trees. Let us first consider the exact range search algorithm. We suppose that, given a node N of a BBD tree, the value $weight(N)$, defined as the sum of the weights of all objects included in the region associated with N, is also registered in the node N, so no access to children is required to compute it. In the following, by using N we also refer to the set of objects contained in node N. Given a range query $\mathbf{range}(O_q, r_q)$, the exact range search algorithm traverse the BBD tree counting the weight of objects inside $\mathcal{B}(O_q, r_q)$ as follows. The algorithm starts from the root node of the tree and

initialize a global variable *count* to 0. Given a node N_i of the tree the algorithm does the following:

(a) if $N_i \subset \mathcal{B}$ add $weight(N)$ to *count*; Stop.

(b) if $N_i \cap \mathcal{B} = \emptyset$ do nothing; Stop.

(c) if N_i is a leaf node check if the associated object is included in $\mathcal{B}$, and in that case add its weight to *count*; Stop.

(d) if N_i is an internal node, recursively consider all its children and add the weights respectively obtained.

The behaviour of this algorithm is sketched in Figure 5.4a). If the region associated with the current node is included in $\mathcal{B}$, its weight is immediately considered, since it is stored in the node, without accessing its children. Therefore, in the figure, the weight of $\mathcal{R}_1$ is immediately considered. If the region associated with the current node is outside the query region, as $\mathcal{R}_3$ in the figure, it is immediately discarded. In the other cases, the specific children are accessed. Therefore, children of the node corresponding to region $\mathcal{R}_2$ are accessed.

This algorithm can be slightly modified to answer to an $(1+\epsilon)$-approximate range query using an approximate pruning condition as follows:

(a) if $N_i \subset \mathcal{B}^+$ add $weight(N)$ to *count*: Stop.

(b) if $N_i \cap \mathcal{B}^- = \emptyset$ do nothing; Stop.

(c) if N_i is a leaf node check if the associated object is included in $\mathcal{B}$, and in that case add its weight to *count*; Stop.

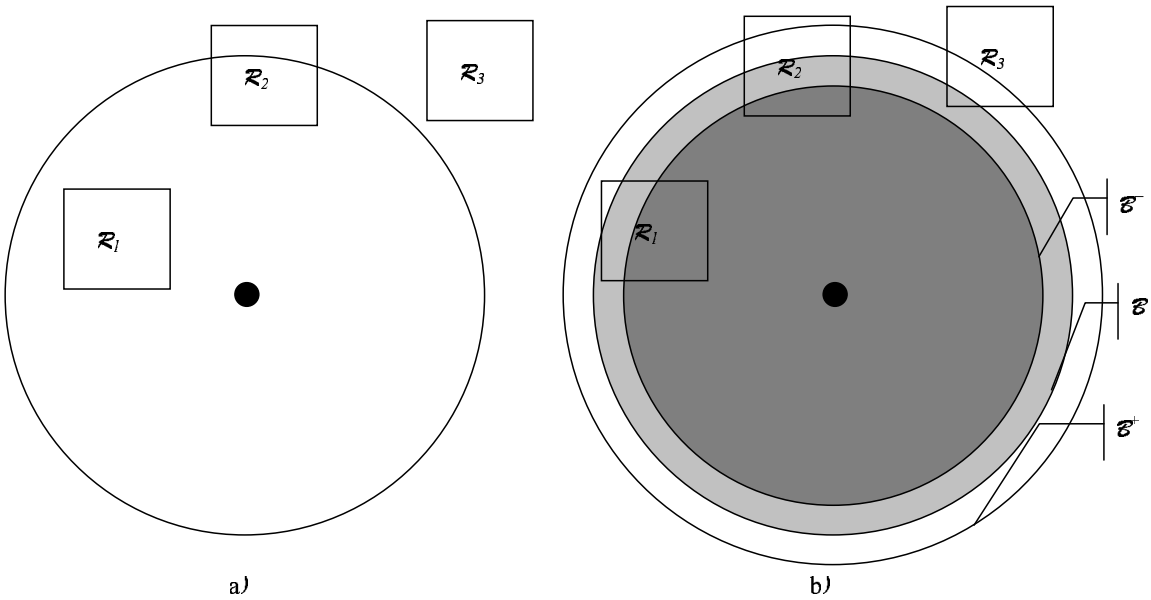


Figure 5.4: Range queries using BBD tree: a) exact behaviour and b) approximate behaviour

(d) if N_i is an internal node, recursively consider all its children and add the weights respectively obtained.

The behaviour of the approximate algorithm is sketched in Figure 5.4b). If the region associated with the current node is included in $\mathcal{B}^+$, its weight is immediately considered without accessing its children – false hits may occur, since objects outside the query region $\mathcal{B}$ may be counted. In the figure, the weight of $\mathcal{R}_1$ is consequently immediately considered. If the region associated with the current node is outside the region $\mathcal{B}^-$ and not included in $\mathcal{B}^+$, as $\mathcal{R}_3$ in the figure, it is immediately discarded – false dismissals may occur, since objects inside $\mathcal{B}$ may be missed when they are outside $\mathcal{B}^-$. In the remaining cases, children of the corresponding node are accessed. Thus, in the figure, children of the node corresponding to region $\mathcal{R}_2$ are accessed.

The complexity of the search algorithm is proven to be $O(\log n + (1/\epsilon)^{dim})$ and the

lower bound of the maximum number of nodes visited is $\log n + (1/\epsilon)^{dim-1}$. Notice that for values of ϵ smaller that 1, the complexity increases exponentially with the number of dimensions.

The algorithm as it is, can only be used to count (compute weight of) objects that qualify for an approximate range query. Specifically, it cannot be used to retrieve the set of objects qualifying for the approximate range query. In fact, in order to obtain that, leaf nodes should always be accessed, since leaf nodes are the only nodes containing pointers to real objects.

5.3.5 Approximate nearest neighbors searching using angle property

In [PAL99, PL99] an original technique for reducing the number of node accessed during nearest neighbors searching is proposed. The main novelty of this technique consists in the use of the angle between objects contained in a ball region, defined in a vector space, and a query object, with respect to the center of the ball region as shown in Figure 5.5. Exploiting this angle, some heuristics to decide whether a region should be accessed or not are proposed. This technique was applied to SS-Trees [WJ96], however it can be applied to all access methods for vector spaces that partition the space, bound elements of the partition with ball regions, and organize regions hierarchically.

The proposed heuristics are justified by the following three properties of data sets represented in high dimensional vector spaces:

Pr_1: As dimensionality rises, the points that are bounded by a ball region become almost equidistant from the center of the ball region.

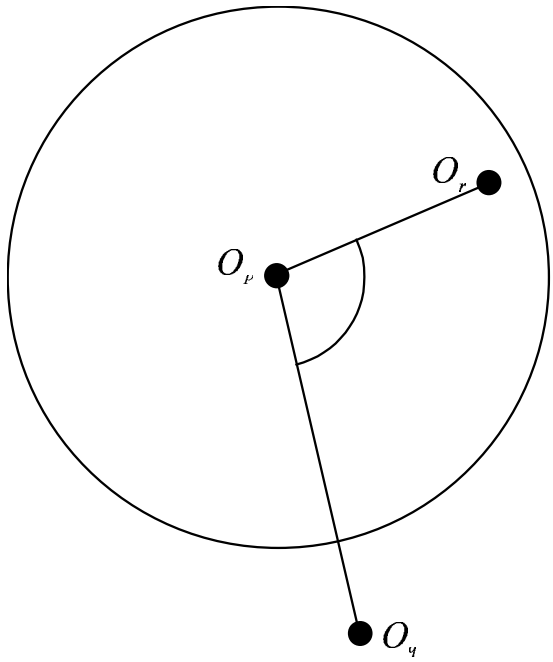


Figure 5.5: Angle between objects contained in a ball region and a query object with respect to the center of the ball region

Pr_2: As dimensionality increases, the radii of the, smaller, child ball regions grow nearly as large as the radius of the larger parent ball region, thus also their centers tend to be close each other.

Pr_3: Given a set of points within a bounding ball region and some query point in the vector space, the angle between the query point and each point in the ball region falls inside a decreasing interval of angles around $\pi/2$.

Let us discuss in detail the proposed heuristics. Exact nearest neighbors search algorithms access all regions that overlap the current query region. The proposed heuristics uses an approximate pruning condition to decide whether a region should be accessed or not.

In [PAL99] they propose to access a region if one of the following conditions is true:

C_1: The corresponding node is an internal node.

C_2: The current query region includes the center of the region's parent

C_3: The center of the region resides in the half of the parent ball region closer to the query object (that is the angle between the center of the region and the query object, with respect to the center of the parent's ball region is smaller than $\pi/2$).

Condition C_1 forces all internal node to be examined. This is justified because in SS-Trees much of the performance degradation in nearest neighbor searching is due to the accesses to the leaf nodes. Properties Pr_1, Pr_2 and Pr_3 suggest that in high dimensional vector spaces, objects reside close to the border of their bounding region, and in particular since child regions tend to be as large as their parent, objects are close to the border of the parent region. In addition these objects fall close to a 90 degree angle with a given query object. Therefore as suggested by Figure 5.6 regions whose center are in the half ball region closer to O_q, as required by condition C_3, are more likely to contain qualifying objects. Finally C_2 was proposed to avoid cases where applying C_3 can be harmful since several qualifying objects may be missed.

In order to check C_3 the evaluation of the angle between the query region and the region's center, with respect to its parent's center is needed. If the angle is acute then the region's center is in the half of the parent region closer to the query region. Deciding if the angle is acute, obtuse, or right can be done by using the *dot product*. Let us suppose that the query region O_q is $(v_1^q, \ldots, v_{dim}^q)$, the region's center O_r is

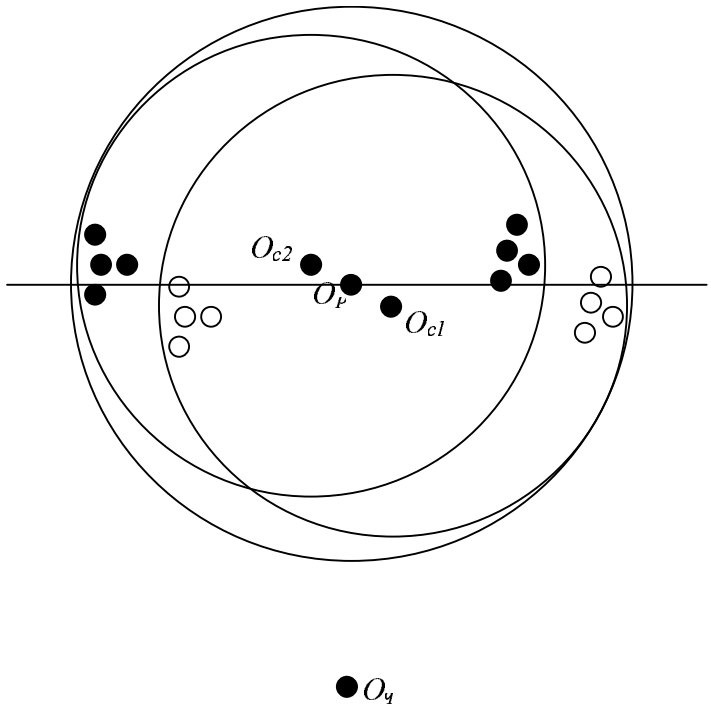


Figure 5.6: Objects belonging to children whose center is in the closest half of the parent node are more likely to contain nearest neighbors

$(v_1^r, \ldots, v_{dim}^r)$, and its parent's center O_p is $(v_1^p, \ldots, v_{dim}^p)$. Then, the dot product $[O_q \cdot O_r]_{O_p}$ of O_q and O_r, with respect to O_p is computed as follows:

$$[O_q \cdot O_r]_{O_p} = \sum_{1}^{dim} (v_i^q - v_i^p)\,(v_i^r - v_i^p)$$

When $[O_q \cdot O_r]_{O_p} > 0$ the angle is acute. If $[O_q \cdot O_r]_{O_p} < 0$ the angle is obtuse. Finally, if $[O_q \cdot O_r]_{O_p} = 0$ the angle is right.

In [PL99] the previous pruning conditions were improved. In fact, the zone where is more likely to find qualifying objects according to properties Pr_1, Pr_2, and Pr_3, is close to the border of the data region, and forms an angle of about 90 degree with the query object, with respect to the region's center. Let suppose to indicate by θ

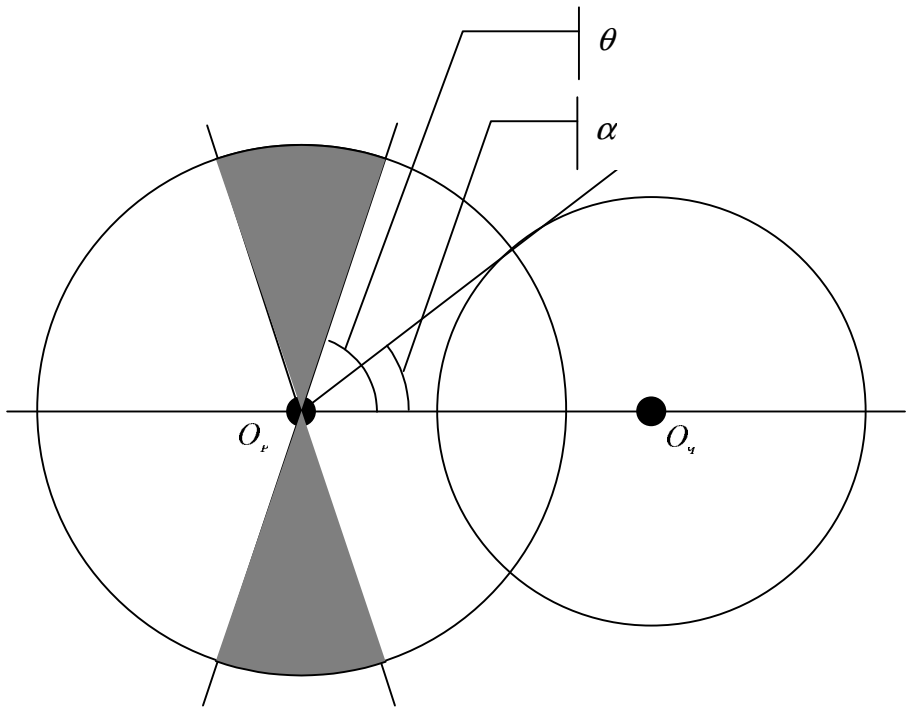


Figure 5.7: If the query region does not intersect promising portions of the data region, this is discarded.

the expected angle of the promising zone. If the angle α obtained considered the intersection between the two regions and the line passing from their centers, with respect to the data region's center, is higher than θ then the region is accessed, elsewhere it is discarded. Figure 5.7 sketches this situation. Notice that when θ is set to 0 all regions overlapping the query region are accessed.

These methods can only be applied to nearest neighbors search in vector spaces. The first version of the proposed approach cannot be tuned to trade performance with quality of results, while the second allows one to use the angle θ as a threshold to decide when regions can be discarded.

5.3.6 PAC nearest neighbor searching

In [CP00] an approach for *Probably Approximately Correct* nearest neighbor search in metric spaces is proposed. The idea is to bound the error on the distance of the approximate nearest neighbor so that an $(1+\epsilon)$-approximate nearest neighbor is found, similarly to the technique using BBD trees, see [AMN+98] and Section 5.3.3. In addition, the proposed algorithm may prematurely stop when the probability that the current approximate nearest neighbor is not an $(1+\epsilon)$-approximate nearest neighbor is below a user defined threshold δ. In this proposal the approximation is controlled by two parameters. The ϵ parameter is used to specify the upper bound on the desired relative error on the distance of the approximate nearest neighbor. The δ parameter is used to specify the degree of confidence that the ϵ upper bound is not exceeded. Notice that if δ is set to zero, the proposed algorithm stops when the resulting object is guaranteed to be a $(1+\epsilon)$-approximate nearest neighbor. Values of δ greater than zero may return an object that is not an $(1+\epsilon)$-approximate nearest neighbor. On the other hand, when ϵ is set to zero, δ controls the probability that the retrieved object is not the real nearest neighbor. Of course, when both ϵ and δ are set to zero, the exact nearest neighbor is always found.

More formally, let O_q be the query object, O_N the exact nearest neighbor, and O_N^A the approximate nearest neighbor found. Let ϵ_{act} the actual error on distances, that is

$$\epsilon_{act} = \frac{d(O_q, O_N^A)}{d(O_q, O_N)} - 1.$$

The proposed approximate nearest neighbor algorithm retrieve a $(1+\epsilon)$-approximate nearest neighbor with confidence δ. That is, the retrieved object is such that

$$\Pr\{\epsilon_{act} > \epsilon\} \leq \delta.$$

These ideas were applied to the algorithm for searching the nearest neighbor to a query object in M-Trees [CPZ97]. Therefore this method is not limited to the case of data represented in vector spaces, but data represented in a generic metric space can be searched using this approach.

The authors observe that a similarity search process can be conceptually split in two phases.

Locating In the first phase the search algorithm locates the object that will be returned

Stopping The second phase, the longest one, is used to determine that the object retrieved is in fact the correct one

Accordingly the two approximation parameters ϵ and δ are used respectively to control how the object should be located, and when the algorithm should stop.

Let us see how this two actions are accomplished. The nearest neighbor search algorithm for M-Trees recursively access data regions (nodes) of the tree starting from the root nodes. When the query region, defined by the query object and the distance of the current nearest object found, and a data region overlap, the data region is accessed. The overlap test here is relaxed so that $(1+\epsilon)$-approximate nearest neighbor are considered. This relaxed overlap test was inspired by the technique proposed in this thesis, see Section 6.6 and [ZSAR98]. The exact algorithm stops when no more overlapping regions are found. The proposed algorithm, on the other hand, stops when the probability that the effective error does not exceed ϵ is smaller than δ. This stop condition is obtained using $G_{O_q}(x)$, the *distribution of the nearest neighbors* of

O_q with respect to a specific data set $\mathcal{DS}$ of size n, defined as follows [CPZ98a]:

$$G_{O_q}(x) = \mathtt{Pr}\left\{\exists O \in \mathcal{DS} : d(O_q, O) \leq x\right\} = 1 - (1 - F_{O_q}(x))^n.$$

As previously stated, the algorithm should stop when $\mathtt{Pr}\left\{\epsilon_{act} > \epsilon\right\} \leq \delta$, where ϵ_{act} is the actual relative error on distances. That is when

$$\begin{aligned} &\mathtt{Pr}\left\{\exists O \in \mathcal{DS} : d(O_q, O_N^A)/d(O_q, O) - 1 \geq \epsilon\right\} = \\ &= \mathtt{Pr}\left\{\exists O \in \mathcal{DS} : d(O_q, O) \leq d(O_q, O_N^A)/(1+\epsilon)\right\} \leq \delta \end{aligned}$$

and this correspond to check if

$$G_{O_q}(d(O_q, O_N^A)/(1+\epsilon)) \leq \delta.$$

This algorithm can be used for generic metric spaces and its performance is good, however it is limited to the case of just a single nearest neighbor search.

Chapter 6

Four new techniques for approximate similarity search in metric spaces

6.1 Introduction

In this chapter we discuss and propose new techniques for approximate similarity search. These approaches offer a high improvement of efficiency at the price of inprecision in the results. Our investigation focuses on approximate similarity search of data represented in metric spaces. As previously stated, this also includes the more specific case of vector spaces, thus our techniques can also be applied to them. We suppose that tree-based access methods for metric spaces, as for examples M-Trees [CPZ97] or Slim-Trees [TTSF00], are used to improve performance of similarity search queries. The techniques that we propose can be easily applied to these access methods by modifying their (exact) similarity search algorithms. Our results show that inprecision can be effectively controlled still guaranteeing high performance.

6.2 Overview of the approaches

We have investigated four new techniques for approximate similarity search. They are based on specific definitions for the approximate pruning or stop conditions (see Section 5.3.2). Specifically, two of these approximation techniques work for nearest neighbor queries and use a stop condition as approximation strategy. The other two techniques work both for nearest neighbor and range queries and use a pruning condition as approximation strategy.

First method The first technique of approximate similarity search uses a user defined parameter as an upper bound on the relative error between the distances of the query object from the objects retrieved by the exact algorithm and the distances of the query object from the objects retrieved by the approximate algorithm. The approximate similarity search algorithm decides which nodes of the tree can be pruned, even if their bounding regions overlap the query region, guaranteeing that the obtained relative error on distances does not go above the specified upper bound. This method was defined both for nearest neighbor and range queries and its details are discussed in Section 6.6.

Second method The second technique of approximation retrieves k approximate nearest neighbors of a query object by returning k objects that statistically belong to the set of the n ($n \geq k$) exact nearest neighbors of the query object. The value n is specified by the user as a fraction of the whole data set. The approximate similarity search algorithm stops when it determines, by using the overall distance distribution (see Section 2.3.4), that the current k retrieved objects belong to the specified fraction of the objects nearest to the query. This

method was defined for nearest neighbor queries only and it is discussed in Section 6.7.

Third method The third type of approximation is based on the pragmatic observation that similarity search algorithms on tree structures are defined as an iterative process where in each iteration the result, that is the set of objects retrieved, is improved until no further improvement is possible. In case of k nearest nearest neighbor queries, at each iteration the current k retrieved objects are nearer than those of the previous iteration. This can be made explicit measuring the distance between the current k-th object and the query objects. This distance decrease at each iteration and it can be noticed that it decreases rapidly in the first iterations then it slows down and remains almost stable for several iterations before the similarity search algorithm stops. The approximate similarity search algorithm exploits this behaviour and stops the search algorithm when the distance decreasing of the current k-th object seems to be stopping. In this case a user defined value is used as a threshold to decide when the distance decreasing can be considered to be stopping. This method was defined for nearest neighbor queries only and its description is given in Section 6.8.

Fourth method The fourth technique uses the proximity measure, discussed in Chapter 4, to decide which nodes of the tree can be pruned, even if their bounding regions overlap the query region, guaranteeing a low probability to loose qualifying objects. In fact, when the proximity of a region, bounding a tree's node, and the query region is small, the probability that qualifying objects are found in their intersection is small. A user specified parameter is used as a

Method	Range queries	NN queries	Approx. strategy
First	yes	yes	pruning condition
Second	no	yes	stop condition
Third	no	yes	stop condition
Fourth	yes	yes	pruning condition

Table 6.1: Characteristics of the approximation methods

threshold to decide if a node should be accessed or not. When the proximity is below the specified threshold the node is not accessed. This method was defined both for nearest neighbor and range queries and we discuss it in details in Section 6.9.

A summary of the main features of the proposed approximation methods is given in Table 6.1.

All four techniques of approximation were experimentally tested on real and synthetic data sets, and their efficiency was compared. The results obtained are encouraging, and efficiency improvement of even two orders of magnitude has been achieved. In other words, whereas a precise similarity search may take several minutes, in an approximated search this can be reduced to seconds, while the precision of approximation typically remains quite high. Consider as a preliminary illustrative example Figure 6.1. It presents search results for 10 nearest neighbors of query objects O_{q1} and O_{q2}, separately for the exact (NN) and the proximity based approximate algorithms (ANN) a proximity threshold of 0.01 in HV1 (see Section 2.5) data set. For each retrieved object, the object identifier (OID) and its distance from the query are reported. Objects that are simultaneously found by the exact and approximate algorithms are typed in bold. The last column reports the costs needed to execute the queries as the number of tree node reads. If we consider the first three objects

		OID Dist	OID Dist	OID Dist	OID Dist	OID Dist	OID Dist	OID Dist	OID Dist	OID Dist	OID Dist	Cost
O_{q1}	NN	7083	1561	7079	**2843**	8849	7077	6755	**7035**	2830	**257**	*1465*
		1298.2	1312.4	1329.35	1359.44	1362.76	1366.64	1376.8	1390.41	1404.95	1409.77	
	ANN	**2843**	**7035**	**257**	522	1206	839	567	1590	205	1069	*11*
		1359.44	1390.41	1409.77	1453.15	1517.73	1521.83	1549.45	1591.93	1631.66	1675.41	
O_{q2}	NN	3493	8623	1139	**3494**	4902	3954	**3504**	4010	8608	7255	*1503*
		1012.21	1032.24	1062.61	1063.53	1118	1131.31	1220.31	1233.08	1238.64	1250.13	
	ANN	**3494**	**3504**	636	1046	902	520	696	480	1286	1227	*15*
		1063.53	1220.31	1389.44	1389.88	1441.54	1495.74	1508.13	1597.27	1611.6	1615.37	

Figure 6.1: Comparison between the 10 nearest neighbors obtained by the precise and the proximity based approximate algorithms for two specific queries, using 0.01 as proximity threshold in the HV1 data set.

in the approximate response to O_{q1}, we can see that these objects are in the precise response on positions four, eight, and ten. However, the cost of the approximate algorithm is only 11 while that of the exact one is 1465. In a similar way, the first two approximate results of query O_{q2} correspond, respectively, to objects in positions four and seven in the exact response. The cost of the approximate algorithm is 15 while the exact search needs 1503 tree node reads, i.e. disk accesses.

In the following, we first outline the generic approximate similarity search algorithms for range and nearest neighbor queries used to implement the four approximation techniques (Section 6.3), we define the performance measures for objectively comparing the proposed techniques (Section 6.4), we describe the testing environment (Section 6.5), we give the details of the four approximation techniques and we discuss the obtained results (Sections 6.6, 6.7, 6.8, 6.9).

6.3 Generic approx. similarity search algorithms

In this thesis we propose four techniques for approximate similarity search. This section discusses the generic approximate range and nearest neighbors algorithms that can be used to support them. These algorithms are obtained by modifying Algorithms 3.3.1 and 3.3.2 for exact similarity search on tree-based access methods discussed in Section 3.3.2. Since we define techniques that can be used in generic metric spaces, we suppose that nodes, of the tree structure, are associated with ball regions (see Section 2.3.2), however the algorithms are still simplified and generic, not strictly related to any specific implementation. The pseudo-code of the algorithms for approximate similarity range queries and approximate nearest neighbors queries are respectively presented in Algorithms 6.3.1 and 6.3.2.

The difference between the four approximation techniques that we propose rely on the specific definition of the pruning or stop condition. Accordingly the approximation can be controlled by two different approximation parameters. The approximation parameter x_s is used by the stop condition strategy, while the approximation parameter x_p is used by the pruning condition strategy. Of course, the specific meaning of these two parameters and their use strictly depend on the specific techniques used to implement the stop or the pruning condition. The generic pruning condition $Prune(\mathcal{R}_q, \mathcal{R}_d, x_p)$ takes as arguments the query region $\mathcal{R}_q$, a data region $\mathcal{R}_d$ and the approximation parameter x_p. It returns true when the pruning strategy determines that the node covered by the data region can be pruned according to the approximation parameter x_p. The generic stop condition $Stop(RS_c, x_s)$ takes as arguments the current result set RS_c (the set of qualifying objects found up to the current iteration)

and the approximation parameter x_s. It returns true when the stop strategy determines that the current result set satisfies the approximation requirements, according to the approximation parameter x_s.

When the *Prune* function is defined as an overlap test and the *Stop* function is defined to be always false, the algorithms have the behaviour of the exact similarity search algorithms. In the other cases, it may happen that some nodes, which contain qualifying results, are pruned or that the algorithm is stopped before all results were found. Therefore, false dismissal can occur.

The thresholds x_s and x_p are used to tune the trade-off between the efficiency and effectiveness. Values corresponding to high performance give a less effective approximation, because more qualifying objects may be dismissed. Values that give very good approximations correspond to more expansive query execution since few node's accesses are discarded.

In all approximation techniques proposed, when x_s and x_p are set to zero, exact similarity searches are performed.

6.3.1 Generic approximate range search algorithm

Our approximate range search algorithms use only the pruning condition strategy, so they only depend on the x_p approximation parameter. The generic definition of the approximate range search algorithm is given in Algorithm 6.3.1. The algorithm takes as input values the query region, composed of the query object O_q and the query radius r_q, and the approximation parameter x_p. It returns the set of objects[1]

[1]The range search algorithm presented in Section 3.3.2 returns a set of pairs (p_j, O_j). Here for simplicity we omit the pointers p_j to the records.

Algorithm 6.3.1. **Range**
Input: query object O_q; query radius r_q; approximation parameter x_p.
Output: response set $\mathbf{range}^{x_p}(O_q, r_q)$.

1. Enter pointer to the root node into **PR**; empty $\mathbf{range}^{x_p}(O_q, r_q)$.
2. While $\mathbf{PR} \neq \emptyset$, do:
3. Extract entry N from **PR**.
4. Read N.
5. If N is a leaf node then:
6. For each $O_j \in N$ do:
7. If $d(O_q, O_j) \leq r_q$ then $O_j \rightarrow \mathbf{range}^{x}(O_q, r_q)$.
8. If N is an internal node:
9. For each child node N_c of N, bounded by region $\mathcal{B}_c(O_j, r_j)$ do:
10. If $\neg Prune(\mathcal{B}(O_q, r_q), \mathcal{B}_c(O_j, r_j), x_p)$ then insert pointer to N_c into **PR**.
11. End

$\mathbf{range}^{x_p}(O_q, r_q) = \{O_1^A, O_2^A, ..., O_{l_A}^A\}$ where O_i^A are the objects retrieved by the algorithm and l_A is the number of objects retrieved.

The main differences of the approximate range search algorithm with respect to the exact range search algorithm (see Algorithm 3.3.1) can be seen at Steps 8, 9 and 10. The exact range search algorithm, in fact, simply checks if the data region overlaps the query region. On the other hand, the approximate range search algorithm, if the extracted node is an internal node (Step 8), checks the approximate pruning condition. Each pointed node is inserted in **PR**, to be examined in the next iterations, if the pruning condition is false (Steps 9 and 10). Therefore, the approximate range search algorithm, depending on the definition of the approximate pruning condition, may discard some nodes even if they contain qualifying objects. Of course, a good pruning condition should suggest to discard nodes only when the chances that they contain

qualifying objects are very low.

6.3.2 Generic approximate nearest neighbors search algorithm

The approximate nearest neighbors algorithms may use both stop and pruning condition strategies so it depends on both x_s and x_p approximation parameters. The generic algorithm for approximate nearest neighbors search takes as input values the query object O_q, the number of neighbors required k, and the approximation parameters x_p and x_s. It returns the set of objects[2] $\mathbf{nearest}^{x_p,x_s}(O_q,k) = \{O_1^A, O_2^A, ..., O_k^A\}$ where O_i^A are the objects retrieved by the algorithm. The generic approximate nearest neighbors search algorithm is defined by Algorithm 6.3.2.

The main differences of the approximate nearest neighbors search algorithm with respect to the exact nearest neighbors search algorithm (see Algorithm 3.3.2) can be seen at Steps 4, 8 and 11 of Algorithm 6.3.2. At Steps 4 and 11 the approximate pruning condition is used instead of the simple overlap test, as discussed for the approximate similarity range search algorithm. In addition, at Step 8, the approximate stop condition is used: if the stop strategy determines that the approximate result set satisfies the approximation requirements, according to the x_s parameter, then the algorithm stops before the natural end, eventually missing other qualifying objects. The approximate stop condition, is accurate if it is able to stop the algorithm when the current result set is a good approximation of the exact result set.

[2] As we also said for the approximate range search algorithm, differently from the algorithms presented in Section 3.3.2, for simplicity we omit the pointers p_j to the real data.

Algorithm 6.3.2. **Nearest neighbors**
Input: query object O_q; number of neighbors k; approximation thresholds x_p and x_s.
Output: response set $\mathbf{nearest}^{x_p,x_s}(O_q,k)$.

1. Enter pointer to the root node into **PR**; fill $\mathbf{nearest}^{x_p,x_s}(O_q,k)$ with k (random) objects; determine r_q as the max. distance in $\mathbf{nearest}^{x_p,x_s}(O_q,k)$ from O_q.
2. While $\mathbf{PR} \neq \emptyset$, do:
3. Extract the first entry N from **PR**. Suppose that N is bounded by region $\mathcal{B}(O_i,r_i)$.
4. If $\neg Prune(\mathcal{B}(O_q,r_q),\mathcal{B}(O_i,r_i),x_p)$ then read N, go to 2 otherwise.
5. If N is a leaf node then:
6. For each $O_j \in N$ do:
7. If $d(O_q,O_j) < r_q$ then update $\mathbf{nearest}^{x_p,x_s}(O_q,k)$ by inserting O_j and removing the most distant from O_q; set r_q as the max. distance of objects in $\mathbf{nearest}^{x_p,x_s}(O_q,k)$ from O_q.
8. If $Stop(\mathbf{nearest}^{x_p,x_s}(O_q,k),x_s)$ exit.
9. If N is an internal node:
10. For each child node N_c of N bounded by region $\mathcal{B}_c(O_j,r_j)$ do:
11. If $\neg Prune(\mathcal{B}(O_q,r_q),\mathcal{B}_c(O_j,r_j),x_p)$ then insert pointer to N_c into **PR**.
12. Sort entries in **PR** with increasing distance to O_q.
13. End

6.4 Efficiency and accuracy measures

Assessing the quality of an approximate similarity search algorithm involves the measurement of the improvement of efficiency achieved and the accuracy of the approximate results.

Of course, the improvement of efficiency alone is not sufficient to assess the performance of the approximate similarity search algorithms, because of the natural tradeoff between the efficiency and effectiveness. In fact, high improvement of efficiency typically results in low accuracy of the approximate results. The performance

improvement by using approximate algorithms is obtained at the cost of degrading the quality of retrieval, because some of the objects of the exact query response set may not be included in the approximate one. A good approximate algorithm should be able to provide reasonable accuracy with a high improvement of efficiency with respect to the exact algorithm. Therefore, we also need some measurements to evaluate the quality of the approximate similarity search algorithms.

In the following we discuss these two aspects and we define the corresponding measures.

6.4.1 Efficiency

Approximate similarity search algorithms were introduced since some applications need responses to similarity query faster than what offered by exact similarity search algorithms. The *Improvement of Efficiency*, IE measures how much faster an approximate similarity search algorithm is than the corresponding exact similarity search algorithm. The improvement of efficiency is defined as

$$IE = \frac{cost(\mathbf{oper})}{cost(\mathbf{oper}^A)} \tag{6.4.1}$$

where *cost* is a function that gives the number of tree nodes accessed during query execution, **oper** is either $\mathbf{range}(O_q, r_q)$ or $\mathbf{nearest}(O_q, k)$, and $\mathbf{oper}^A$ corresponds to the approximate versions. Alternatively, *costs* could be defined as the number of distance computations performed during the query execution, but experiments demonstrate that the number of nodes accessed and the number of distance computations are linearly correlated.

6.4.2 Accuracy

We believe that, depending on the two types of similarity queries, nearest neighbor and range search, the quality of approximation should be evaluated from two different points of view:

Nearest neighbor queries: in the case of nearest neighbor queries, a useful information is the difference between the position of an object in the approximate result set and the position of the same object in the exact result set with respect to the query. This helps to judge how many objects were dismissed because of the approximation. For instance, when the first approximate nearest neighbor is the 20-th object in the exact ordering, it means that 19 objects were lost, by the approximation.

Range queries: in the approximate range search algorithms that we propose, false hits never occur so the approximate result set is always a subset of the exact result set. Therefore, it is useful to consider the percentage of objects of the exact result set that are also present in the approximate set. For instance, in the case when the exact result set contains 200 objects, while the approximated one contains 150, the percentage is $150/200 = 0.75$, that is 75% of the exact result set was retrieved.

According to the previous observations, we define and use in our experiments two measurements to assess the quality of approximation. The first one, called *error on the position*, EP, is used to determine the discrepancy between the position occupied by objects in the approximate and exact sets of nearest neighbors. The second measurement, called *number of exact results*, NE, is used to determine the percentage of

objects shared by the exact and approximate result sets of range queries.

In order to provide a more formal definition of these measurements, we need the following notation. Given a metric space $\mathcal{M} = (\mathcal{D}, d)$, we intend to support similarity search (range and nearest neighbor) queries on a set $\mathcal{DS} \subseteq \mathcal{D}$ of n elements. The exact range search retrieves the set $\mathbf{range}(O_q, r_q) = \{O_1, O_2, \ldots O_l\}$ with $l \leq n$, and $d(O_q, O_i) \leq d(O_q, O_j)$, for $i < j$. The approximate range search search algorithm retrieves the set $\mathbf{range}^{x_p}(O_q, r_q) = \{O_1^A, O_2^A, \ldots O_{l_A}^A\}$, where $l_A \leq l$ and $d(O_q, O_i^A) \geq d(O_q, O_i)$. In our approximate range search we have that $\mathbf{range}^{x_p}(O_q, r_q) \subseteq \mathbf{range}(O_q, r_q)$, since false hits never occur. The nearest neighbors search retrieves k objects $\mathbf{nearest}(O_q, k) = \{O_1, O_2, \ldots O_k\}$ with $d(O_q, O_i) \leq d(O_q, O_j)$, for $i < j$. The approximate nearest neighbors search algorithm also retrieves k objects, $\mathbf{nearest}^{x_p, x_s}(O_q, k) = \{O_1^A, O_2^A, \ldots O_k^A\}$, where $d(O_q, O_i^A) \geq d(O_q, O_i)$. We can now provide a definition of EP and NE.

EP: To compute EP we first consider EP_i as the position error of the i-th object O_i^A of the approximate set. It is computed as the difference between the position of the object in the exact result set and the approximated one, divided by the cardinality of the whole data set (otherwise, this measurement would depend on the size of the data set considered). It is easy to show that the position of object O_i^A in the exact result set $\mathbf{nearest}(O_q, k)$ is $\#(\mathbf{range}(O_q, d(O_q, O_i^A)))$. Therefore, the position error for the i-th approximate object is

$$EP_i = \frac{\#(\mathbf{range}(O_q, d(O_q, O_i^A))) - i}{\#(\mathcal{DS})}$$

The position error EP is obtained as the average of the position errors of all

objects in $\mathbf{nearest}^{x_p,x_s}(O_q, k)$

$$EP = \frac{\sum_{i=1}^{k} EP_i}{k} \tag{6.4.2}$$

NE: To compute NE we have to obtain the percentage of objects of the exact result set that are included in the approximate result set. This is simply the number of objects contained in $\mathbf{range}^{x_p}(O_q, r_q)$ divided by the number of objects contained in $\mathbf{range}(O_q, r_q)$

$$NE = \frac{\#(\mathbf{range}^{x_p}(O_q, r_q))}{\#(\mathbf{range}(O_q, r_q))} \tag{6.4.3}$$

It is worth noting that in other works [AMN+98] the measurement used was the *relative error on distances*. This gives the percentage of error on the distance from the query object of the approximate result with respect to the exact result. For the nearest neighbor search it is defined as

$$\epsilon = \frac{d(O_A, O_Q) - d(O_N, O_Q)}{d(O_N, O_Q)} = \frac{d(O_A, O_Q)}{d(O_N, O_Q)} - 1$$

where O_A is the approximate nearest neighbor, O_N is the real nearest neighbor, and O_Q is the query object. This measurement can be easily generalized to the case of k-nearest neighbors search and range queries.

However we believe this measurement not to be suitable to objectively asses the quality of an approximate similarity search algorithm for three reasons:

i) It does not give an idea of the number of object missed.

ii) In case of high dimensional vector spaces it tends to give good results in any case.

iii) Results obtained using different data sets might not be directly compared.

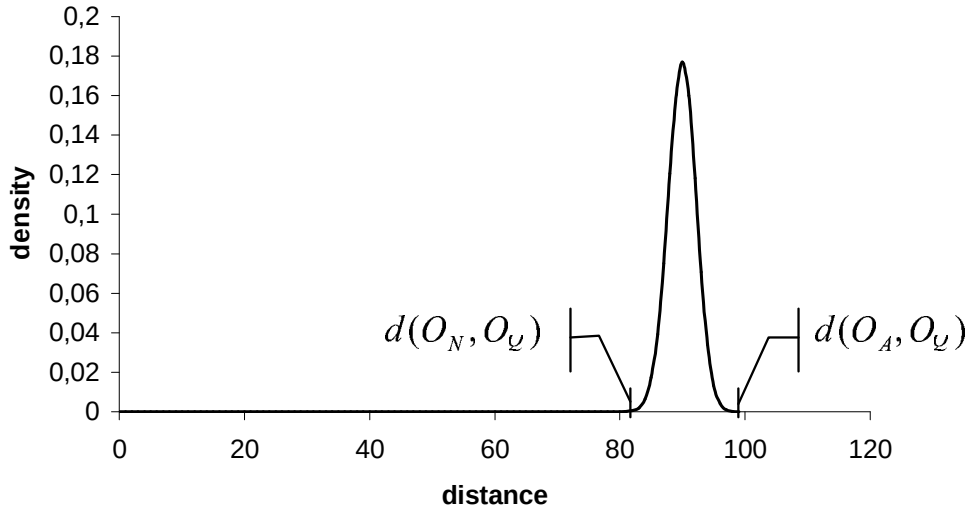


Figure 6.2: The relative distance error is not a reliable measure of the approximation accuracy. Even though the relative distance error is small, almost all objects are missed by the approximate search algorithm.

Let us discuss these three points. i) the relative error on distances gives the percentage of error on the distance of the approximate result with respect to the exact result. However, it does not give any suggestion on the amount of objects missed by the approximate algorithm, since it does not take into account distribution of distances. Let suppose that the distance between the first and the second nearest neighbor is large. Let also suppose that the approximate search algorithm misses the first nearest neighbor and returns the second. In this case the relative error on distances is high even if just one object was missed. Vice versa, suppose that relative error on distance is small, but distances of the approximate nearest neighbor O_A and the exact nearest neigbor O_N from O_Q are such that almost all remaining objects are included in between, as depicted in Figure 6.2. In this case the small relative error is misleading since all objects are in fact missed. ii) in high dimensional

vector spaces all objects tend to have similar distances. It is easy to show that as a consequence of the central limit theorem [HPS71], the peak of the distance density tends to move toward high values of distances as the number of dimensions increases. For illustration consider again Figure 6.2. The consequence is that the relative error on distances tend to be smaller anyway, since the farthest object is always relatively close to the nearest object. iii) the relative error on distances does not take into account the range of distances of the specific data set considered and its distance distribution. A particular relative error, considered to be large in a certain data set, might be considered negligible in a data set where the range of distances is much larger, or distances are distributed differently.

6.5 Experimentation settings

We have implemented the proposed approximate similarity search algorithms in the M-tree [CPZ97] framework. The approximate similarity search algorithms replaced the original search algorithms of an existing C++ implementation of M-Tree – the original M-tree code is available from http://www-db.deis.unibo.it/research/Mtree/.

Data sets UV HV1 and HV2, described in Section 2.5, were used as the test-bed. We executed similarity range queries with radii such that the smallest and the largest radii retrieved approximately between 1% and 20% of objects in the data sets. Nearest neighbors queries were executed varying k between 1 and 50.

For every obtained configuration, various values of x_s and x_p were tested. The used values depend on each specific approximation strategy and are specified later, when the strategies themselves are defined. In order to reduce statistical errors, each approximation parameter was tested for 50 different query objects, using the same

radius in case of range queries, or the same k in case of nearest neighbor queries. Query objects were not part of the data set, but followed the same data distribution.

Average values were computed for NE, EP, and the improvement of efficiency IE. For range queries we provide graphs where we show how IE depends on the approximation parameters, how IE depends on NE and how NE depends on the approximation parameters. For nearest neighbor queries we provide graphs where we show how IE depends on the approximation parameters, how IE depends on EP and how EP depends on the approximation parameters.

6.6 Method 1: Approximate similarity search using relative error of distances

To improve efficiency of similarity search algorithms, access methods for metric spaces partition the searched data and bound such elements of the partition by ball regions. Regions are then hierarchically stored in nodes of a tree. Search algorithms find qualifying objects by only accessing nodes corresponding to regions that overlap the query region. In fact, regions that overlap the query region may potentially contain objects that are also covered by the query regions, while regions that do not overlap the query region can be discarded.

Given a query region $\mathcal{B}(O_q, r_q)$ and a data region $\mathcal{B}(O_i, r_i)$, the simplest overlap test, to decide if the data region (the corresponding node of the tree) should be accessed, is to check if the distance between the centers of the regions is smaller or equal than the sum of their radii as follows

$$d(O_q, O_i) \leq r_q + r_i. \tag{6.6.1}$$

In M-trees [CPZ97], as discussed in Section 3.4.4, this test is further improved. Let $\mathcal{B}(O_p, r_p)$ be the parent region of $\mathcal{B}(O_i, r_i)$, so that the region $\mathcal{B}(O_p, r_p)$ covers $\mathcal{B}(O_i, r_i)$, and the node corresponding to $\mathcal{B}(O_p, r_p)$ is accessed before the node corresponding to $\mathcal{B}(O_i, r_i)$. The overlap test in Equation 6.6.1 is not needed to be performed, and the data region can be immediately pruned, when the difference between the distance of O_q from O_p and the distance of O_i from O_p is greater than the sum of the radii r_q and r_i, that is when

$$|d(O_q, O_p) - d(O_j, O_p)| > r_q + r_i. \tag{6.6.2}$$

Note that, in M-trees it is possible to store in the node the distance $d(O_j, O_p)$ of the object O_p from each entry (from the centers O_i of the corresponding covered regions). Moreover, the distance $d(O_q, O_p)$ between the parent object O_p and the query object O_q is computed when the node is accessed, before accessing its entries. Therefore, previous test in Equation 6.6.2 allows the search algorithm, in some cases, to discard a region without even computing the distance between its center and the query object $d(O_q, O_i)$.

These two pruning tests can be conveniently modified to obtain approximate similarity search algorithms, based on an approximation in the pruning condition, where the result set is constrained by a user-defined relative error of distances ϵ. The idea to obtain this is the following.

Let O_N be the nearest neighbor of O_q, and O_A some other object in the searched

collection. Obviously, provided $0 < d(O_N, O_q) \leq d(O_A, O_q)$, the equation

$$\frac{d(O_A, O_q)}{d(O_N, O_q)} = 1 + \epsilon$$

defines that the distance from O_q to O_A is $(1 + \epsilon)$ times the distance from O_q to O_N. Now assume that O_A is the *approximated* nearest neighbor of O_q. In such case, ϵ represents the *relative error* of the distance approximation, obtained considering O_A as the nearest neighbor of O_q instead of O_N.

Using the relative error of the distance approximation ϵ as an upper bound we can define an object O_A to be an *(1+ϵ)-approximate-nearest-neighbor* [AMN+98] of object O_q, if its distance from O_q is within a factor of $(1 + \epsilon)$ of the distance of the true nearest neighbor O_N, that is when

$$\frac{d(O_A, O_q)}{d(O_N, O_q)} \leq 1 + \epsilon.$$

This idea can be generalized to the case of the j-th nearest neighbor of O_q, for $1 \leq j \leq n$, where n is the size of the database. Using respectively O_A^j and O_N^j to designate, the j-th approximate and the nearest neighbors, the constraint should be modified as follows

$$\frac{d(O_A^j, O_q)}{d(O_N^j, O_q)} \leq 1 + \epsilon.$$

If this constraint is satisfied, O_A^j is called the *(1+ϵ)-j-approximate nearest neighbor* of O_q. However, even if $d(O_q, O_N^j)$ is unique for a given O_q, there may be several objects in the database which, when considered as O_A^j, satisfy the previous equation. This means that the candidate set for approximate results is not necessarily singular. In the limit case, $d(Q, O_A^j) = d(Q, O_N^j)$ or even $O_A^j = O_N^j$.

To see how the search pruning tests of M-trees can be relaxed by respecting a

relative distance error ϵ, let us consider another form of Equations 6.6.2 and 6.6.1 as follows:

$$\frac{r_q}{d(O_i, O_q) - r_i} < 1 \tag{6.6.3}$$

and

$$\begin{cases} \frac{r_q}{|d(O_p,O_q)-d(O_i,O_p)|-r_i} < 1 & \text{if } \mid d(O_p, O_q) - d(O_i, O_p) \mid -r_i > 0 \\ false & \text{elsewhere} \end{cases} \tag{6.6.4}$$

Looking at these fractions, the numerators specify, the distance to the k-th nearest neighbor of O_q discovered so far, in case of k nearest neighbors search, or the maximum accepted distance from O_q, in case of range search. In case of k nearest neighbors search, provided the search has not finished, this distance can also be considered as the distance to the current approximate neighbor. The denominators, on the other hand, put the *lower bounds* (using the corresponding information about distances at hand) on possible nearest neighbors in the region $\mathcal{B}(O_i, r_i)$ with respect to O_q. In other words, the denominators represent the minimum distance that an object in the given region might have, with respect to O_q. Naturally, if the lower bounds (i.e. the denominators) are higher than the current radius of O_q, the region considered cannot contain any qualifying object and therefore can be ignored in the search from this point on.

In order to modify these tests to the case of approximate search, that is when $\epsilon > 0$, the lower bounds can be relaxed by the relative factor ϵ in the following way.

$$\frac{r_q}{d(O_i, O_q) - r_i} < 1 + \epsilon \tag{6.6.5}$$

and

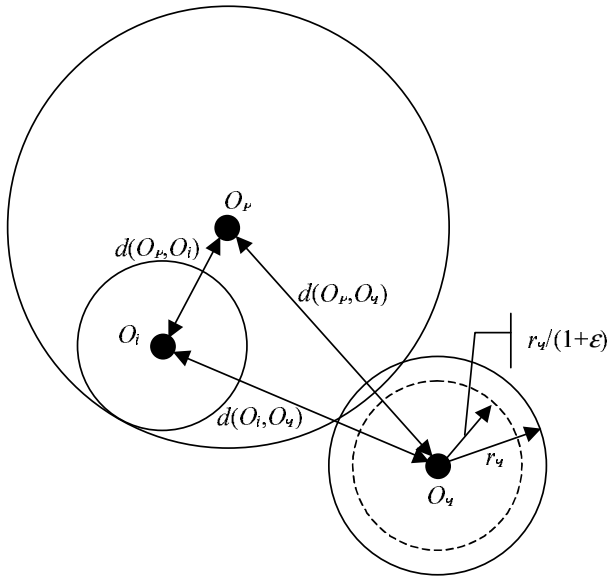


Figure 6.3: The region $\mathcal{B}(O_i, r_i)$, its parent region $\mathcal{B}(O_p, r_p)$, the query region $\mathcal{B}(O_q, r_q)$, and the reduced query region $\mathcal{B}(O_q, r_q/(1+\epsilon))$

$$\begin{cases} \frac{r_q}{|d(O_p,O_q)-d(O_i,O_p)|-r_i} < 1+\epsilon & \text{if } \mid d(O_p,O_q) - d(O_i,O_p) \mid -r_i > 0 \\ false & \text{elsewhere} \end{cases} \tag{6.6.6}$$

Naturally, relaxing these tests in the above way can never increase the similarity search costs, because both the number of distance computations and the number of node reads can only be reduced. In fact, relaxing the pruning tests corresponds to use a smaller query region $\mathcal{B}(O_q, r_q/(1+\epsilon))$, instead of the original query region, as shown in Figure 6.3.

Let us call $\epsilon Prune(\mathcal{B}(O_q, r_q), \mathcal{B}(O_i, r_i), \epsilon)$ the approximate pruning test defined in Equation 6.6.5 and $\epsilon PrePrune(\mathcal{B}(O_q, r_q), \mathcal{B}(O_i, r_i), \epsilon)$ the one defined in Equation 6.6.6. The pruning condition of Algorithms 6.3.1 and 6.3.2 is defined as follows:

$$Prune(\mathcal{B}(O_q, r_q), \mathcal{B}(O_i, r_i), x_p) = \text{ if } \epsilon PrePrune(\mathcal{B}(O_q, r_q), \mathcal{B}(O_i, r_i), x_p)$$
$$\text{return } true$$
$$\text{else}$$
$$\text{return } \epsilon Prune(\mathcal{B}(O_q, r_q), \mathcal{B}(O_i, r_i), x_p)$$

This method uses only the pruning condition, therefore the stop condition is always false:

$$Stop(RS_c, x_s) = false$$

We have tested this method both with range and nearest neighbor queries. The obtained results are discussed in the next section.

6.6.1 Results

Results of the tests performed by using the relative distance error based approximate similarity search algorithms are sketched in Figures 6.4, 6.5, and 6.6 for range queries and in Figures 6.7, 6.8, and 6.9 for nearest neighbor queries.

Using this approximation technique, the threshold x_p is interpreted as the upper bound on the relative error of the distances in the approximate result set with respect to the exact result set. In these experiments we ranged the threshold in the interval between 0 and 8. The approximation threshold x_s for the stop condition was not used, since the stop condition is always false.

A first general observation on the obtained results is that no high improvement of efficiency is obtained with this method. The approach tends to saturate before than valuable improvement of efficiency is obtained. In fact, even if high values of

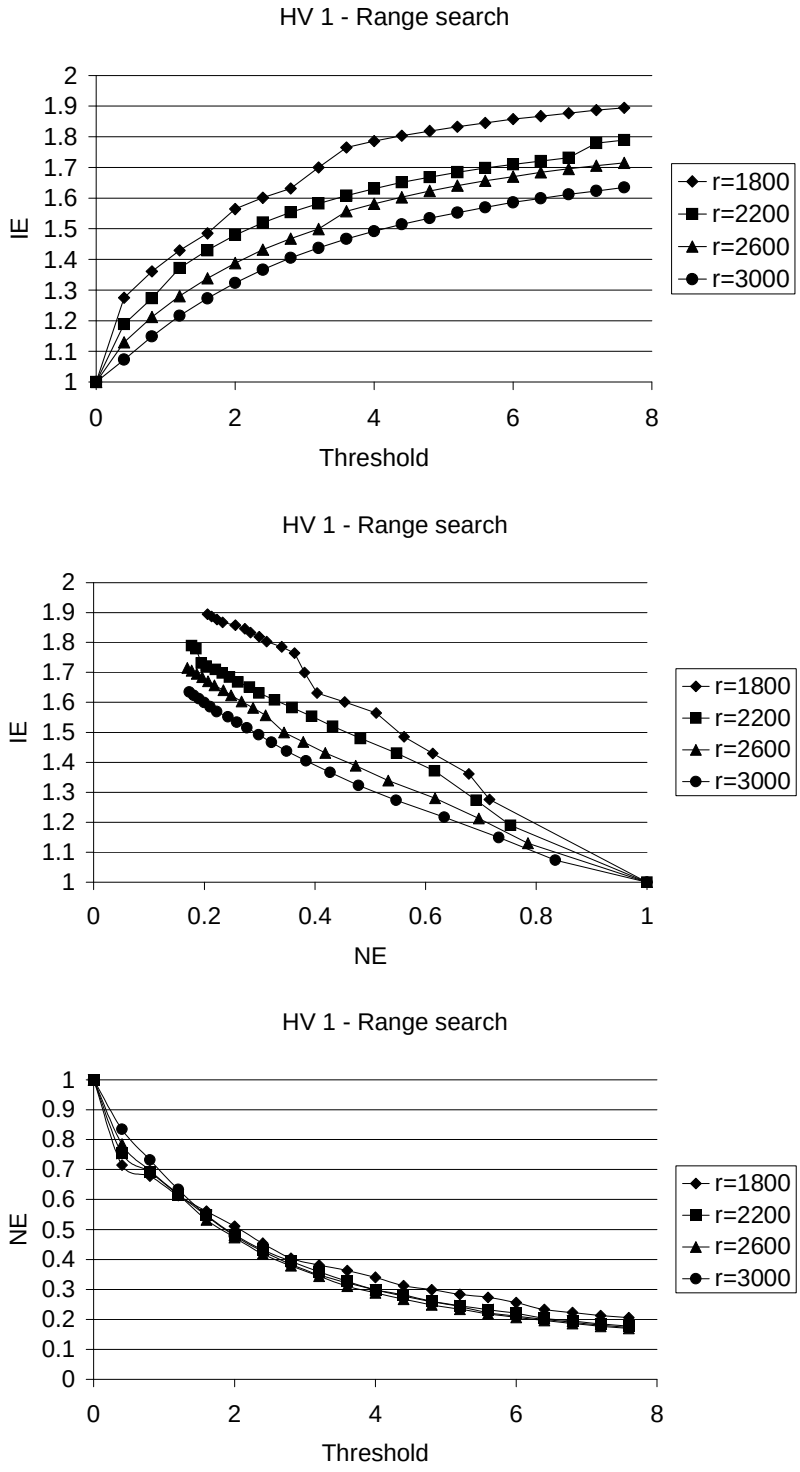
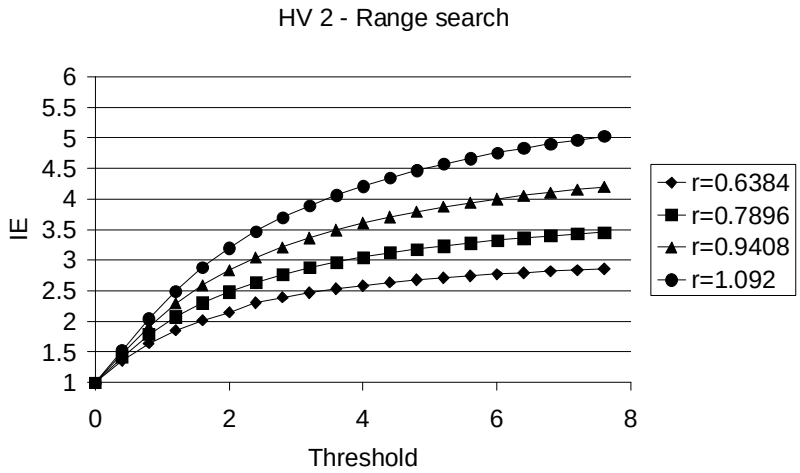
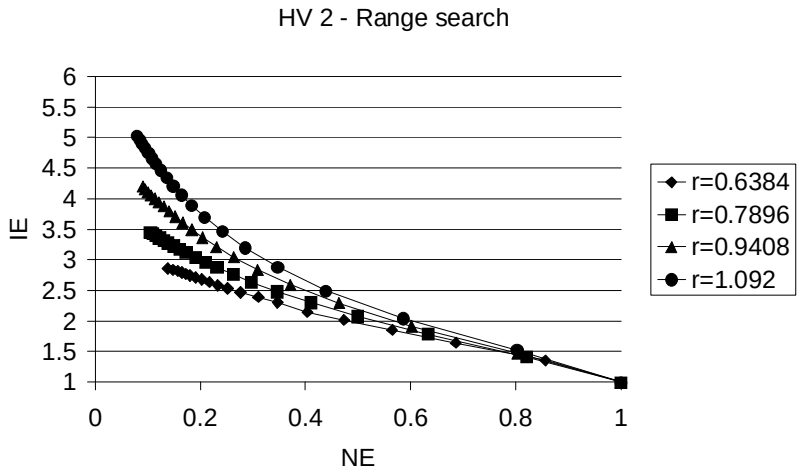


Figure 6.4: Improvement of efficiency (IE) as a function of the proximity threshold (x_p) and the fraction of exact results (NE). Range queries, HV1 data set.





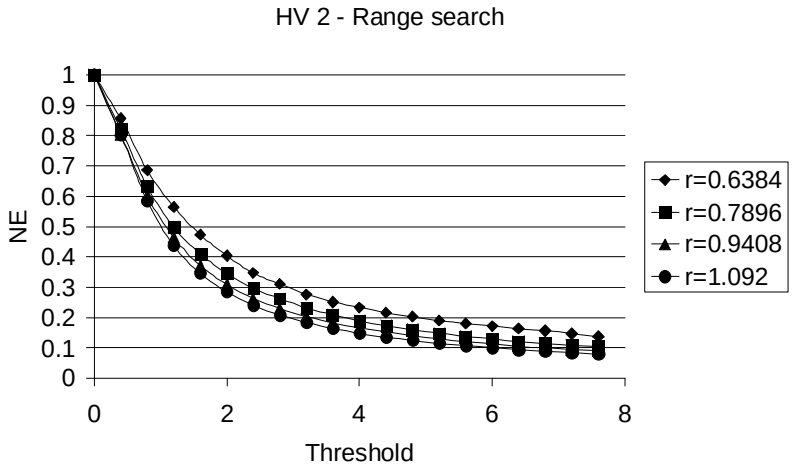


Figure 6.5: Improvement of efficiency (IE) as a function of the proximity threshold (x_p) and the fraction of exact results (NE). Range queries, HV2 data set.

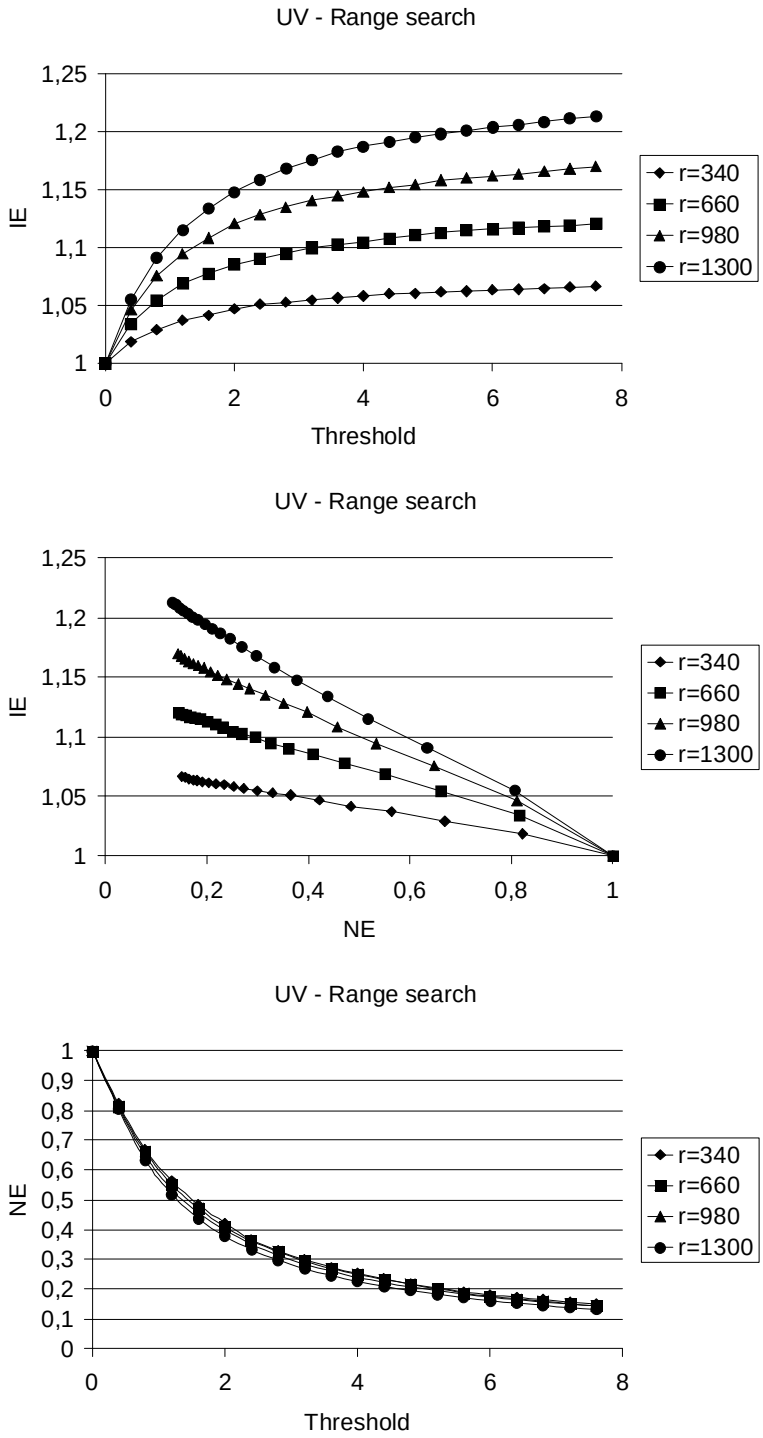
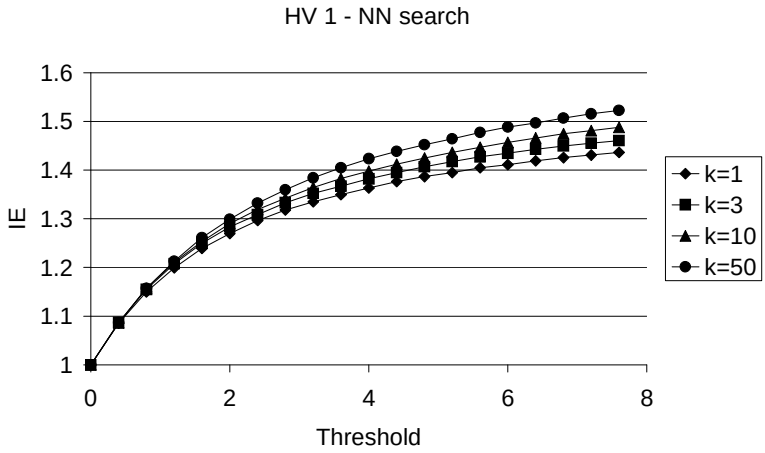
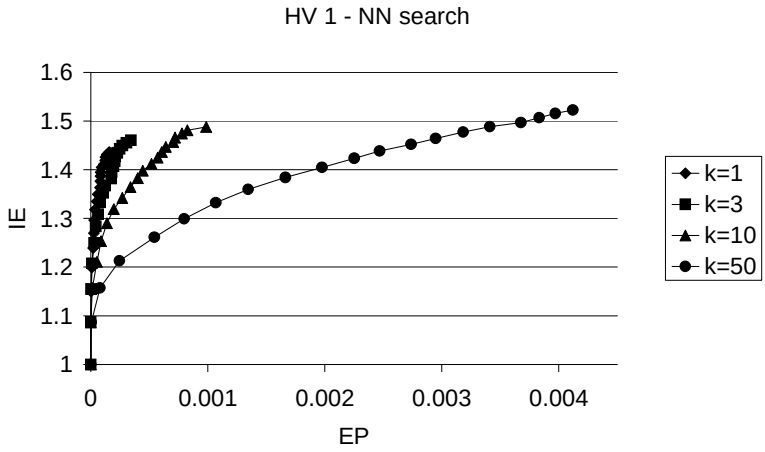


Figure 6.6: Improvement of efficiency (IE) as a function of the proximity threshold (x_p) and the fraction of exact results (NE). Range queries, UV data set.





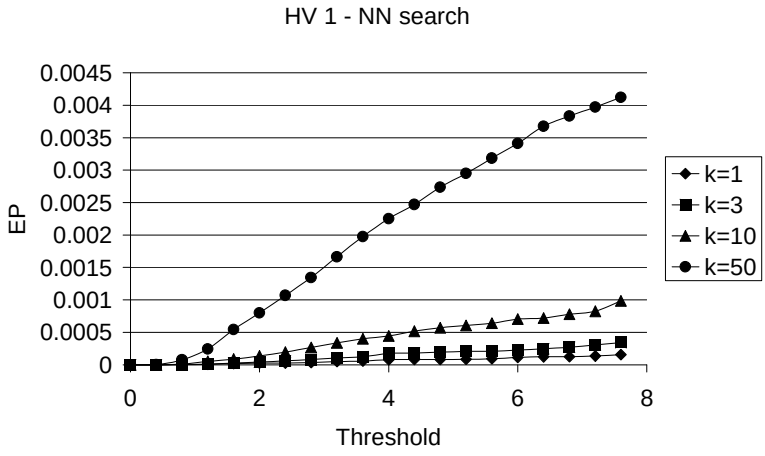


Figure 6.7: Improvement of efficiency (IE) as a function of the proximity threshold (x_p) and the position error (EP). Nearest neighbor queries, HV1 data set.

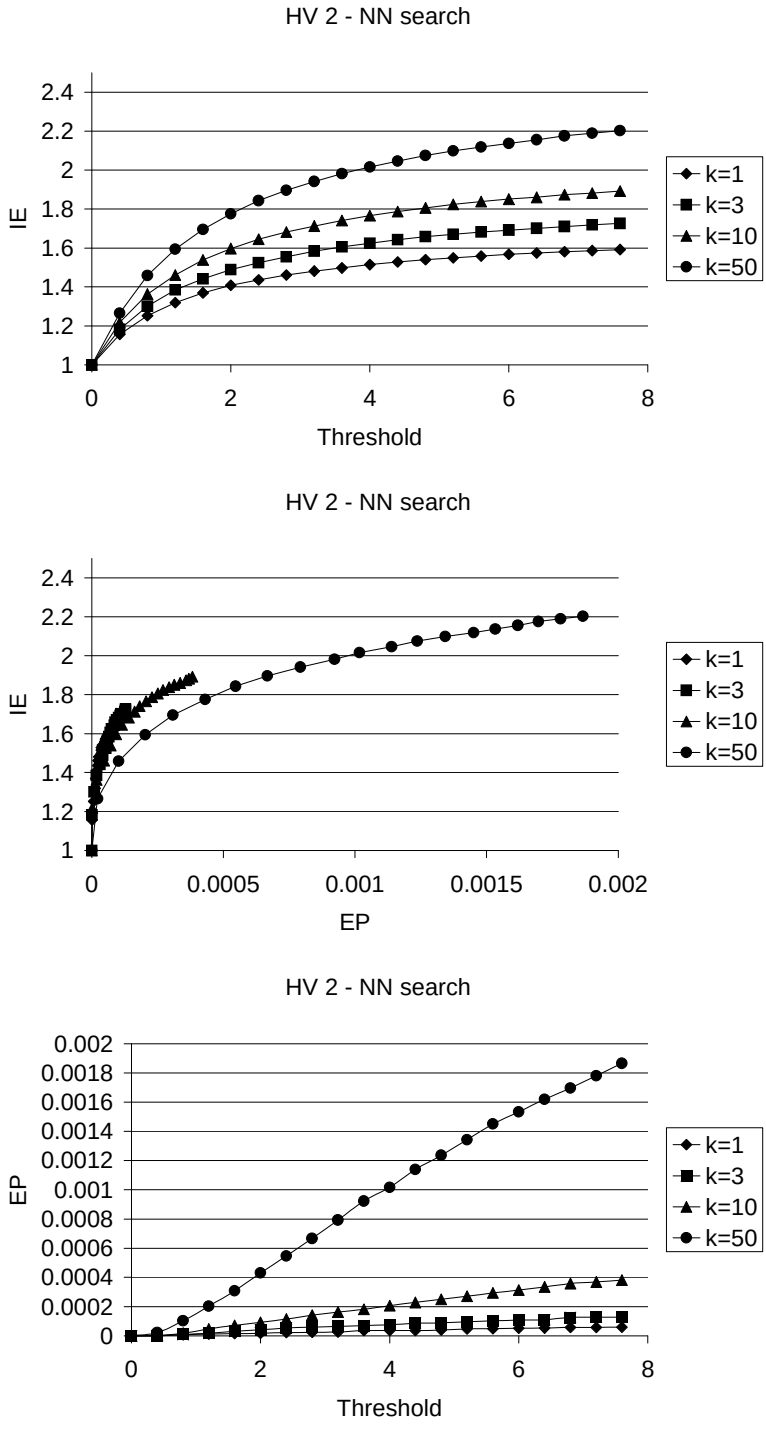


Figure 6.8: Improvement of efficiency (IE) as a function of the proximity threshold (x_p) and the position error (EP). Nearest neighbor queries, HV2 data set.

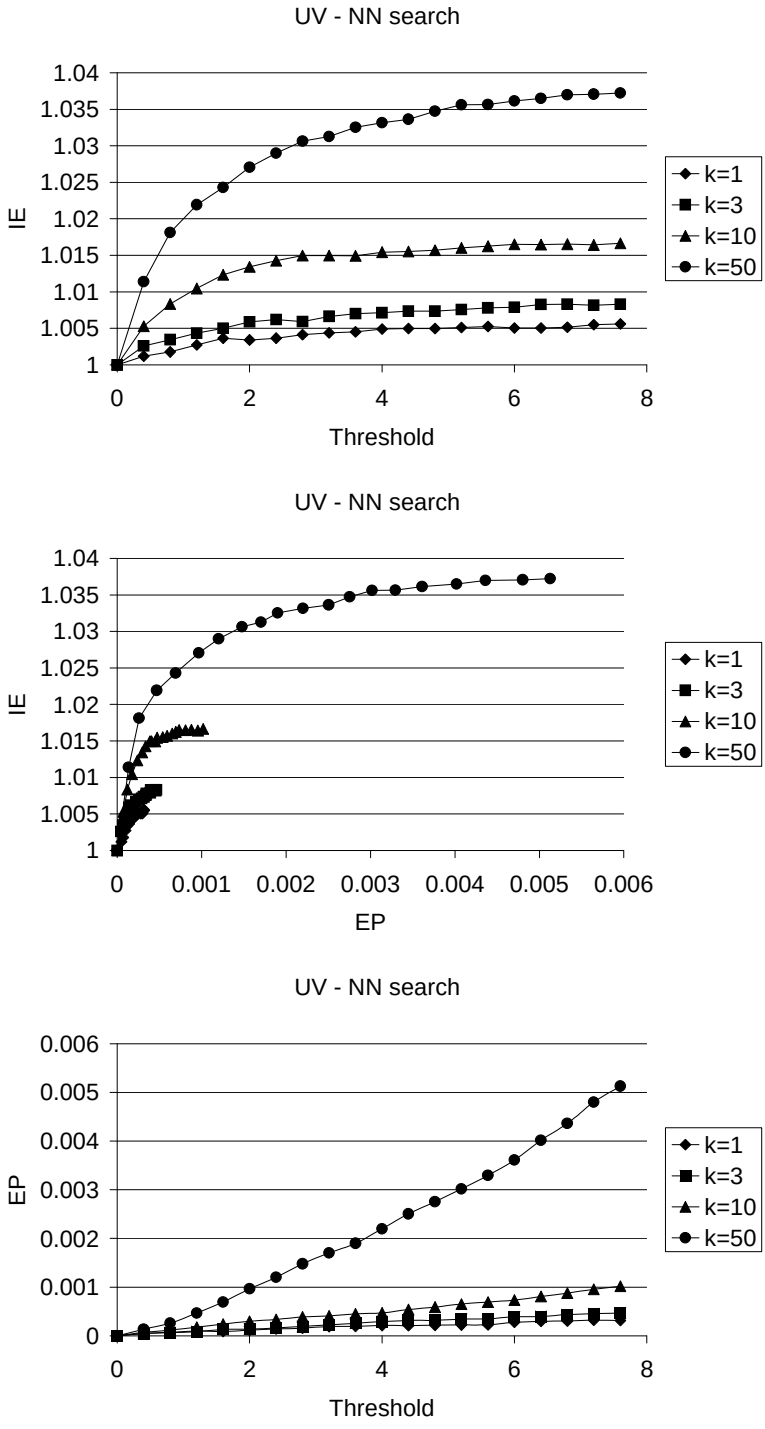


Figure 6.9: Improvement of efficiency (IE) as a function of the proximity threshold (x_p) and the position error (EP). Nearest neighbor queries, UV data set.

the threshold are used, no considerable improvement is achieved. The graphs show that the best results where obtained in the HV2 data set, with range queries, where the approximate range search was executed only five time faster than the exact range search.

Another observation is that there is not evidence of a common trend in the different data sets that we have used. In fact, in case of range queries, the graphs relating NE with IE show that in HV1 performance degrades when the radius of the query increases, while in HV2 and UV performance degrades when the radius decreases. On the other hand, in case of nearest neighbors queries, the graphs relating EP with IE show that in HV1 and HV2 performance degrades when k increases, while in UV performance degrades when k decreases. This means that in order to tune the algorithms, to optimize performance, a deep knowledge of the used data set should be obtained, and no general guidelines can be exploited.

Let us consider specifically the behavior of range queries. In HV1, the best results where obtained when the radius was small. For instance, when the query radius was 1800, a query was executed on average 1.9 times faster with $NE = 0.2$, that is 20% of objects retrieved by the exact search were found by the approximate search. In HV2, the best results were obtained when the radius was large. In fact, when the query radius was 1.092, improvement of efficiency was 3.7 with a value of the the NE measure of 0.2. In case of the UV data set, again, the best results were obtained when the radius was large. In fact, when the query radius was 1300, improvement of efficiency was 1.2 and the NE measure was 0.2.

Let us now consider results of nearest neighbors queries. In HV1, the best results where obtained when k was small. In fact, when k was 1, queries were executed

on average 1.43 times faster, with $EP = 0.0001$, that is, since HV1 contains 10000 objects, the approximate nearest neighbor was on average in the 2-nd position. In HV2, again the best results were obtained when k was small. In fact, with k set to 1, improvement of efficiency was 1.6 with a value of the the EP measure of 0.00005, that is, in almost all cases the real nearest neighbor was found. Finally, in case of the UV data set the best results were obtained when k was large. In fact, with k set to 50, improvement of efficiency was 1.027 and the EP measure was 0.001.

Notice the dependence between the threshold and the NE measure in Figures 6.4, 6.5, and 6.6. NE can be directly controlled by the approximation parameter, independently of the radius of the query. However this property cannot be observed for the nearest neighbors query. In fact, it can be noticed in figures 6.7, 6.8, and 6.9, that, with the same value of x_p, when k is set to 50 the error on the position is on average much higher than using smaller values for k.

6.7 Method 2: Approximate similarity search using distance distribution

As discussed in Section 6.3.2, the k-nearest neighbors search algorithm obtains the final result set by improving iteration after iteration a current result set, say RS_c. In fact, in every iteration, if a new object O is found whose distance from the query object O_q is shorter than that of some of the objects in RS_c, then the k-th object O_k in RS_c is removed and O is inserted in RS_c.

The idea at the basis of the approximate similarity search technique discussed in this section is to stop the search algorithm when the objects of RS_c belong to a user

specified fraction of the closest object to the query object O_q. As an example, let us suppose that the considered data set $\mathcal{DS}$ contains 10000 objects and that objects O_1, O_2,...,$O_{10000} \in \mathcal{DS}$ are ordered with respect to their distance from the query objects O_q. If the user chooses 1/200 (that is 0.5%) as the wanted fraction, then the algorithm should stop as soon as all objects in RS_c belong to the set $\{O_1, O_2, ..., O_{50}\}$ of the 50 closest objects to O_q (since $10000/200 = 50$).

The previous idea can be realized by using a probabilistic approach and exploiting the overall distance distribution (see Section 2.3.4) of the objects of the considered metric space. Let suppose that we have a metric space $\mathcal{M} = (\mathcal{D}, d)$. The distribution function $F_{O_i}(x)$ (the distance distribution relative to O_i, or the O_i's *viewpoint*) gives the probability that chosen a random object $\mathbf{O}$ from $\mathcal{D}$, its distance from O_i is smaller than x, that is $F_{O_i}(x) = \mathtt{Pr}\{d(O_i, \mathbf{O}) \leq x\}$.

Let us suppose that our data set $\mathcal{DS} \subseteq \mathcal{D}$ contains a sample of n objects of $\mathcal{D}$ selected in such a way that for any O_i, belonging to $\mathcal{DS}$, its distance distribution F_{O_i}, computed considering the objects of $\mathcal{DS}$ only, is the same than that computed considering all objects of $\mathcal{D}$[3]. According to this, $F_{O_i}(x)$ represents the fraction of objects in $\mathcal{DS}$ for which the distance to O_i is smaller than or equal to x. In fact, since the number of objects in $\mathcal{DS}$ is n, then the expectation is that $n \cdot F_{O_i}(x)$ objects should have a distance to O_i not greater than x.

Let us now consider the k-nearest neighbor search algorithm and the current result set RS_c obtained at a certain intermediate iteration. Let O_c^k be the current k-th object in RS_c and $d(O_q, O_c^k)$ its distance from O_q. As a consequence of previous

[3]This can be generally considered true when the number of objects of the data set is large. In these cases we can suppose that the data set characterizes the entire metric space which it belongs to (see Section 2.3.4).

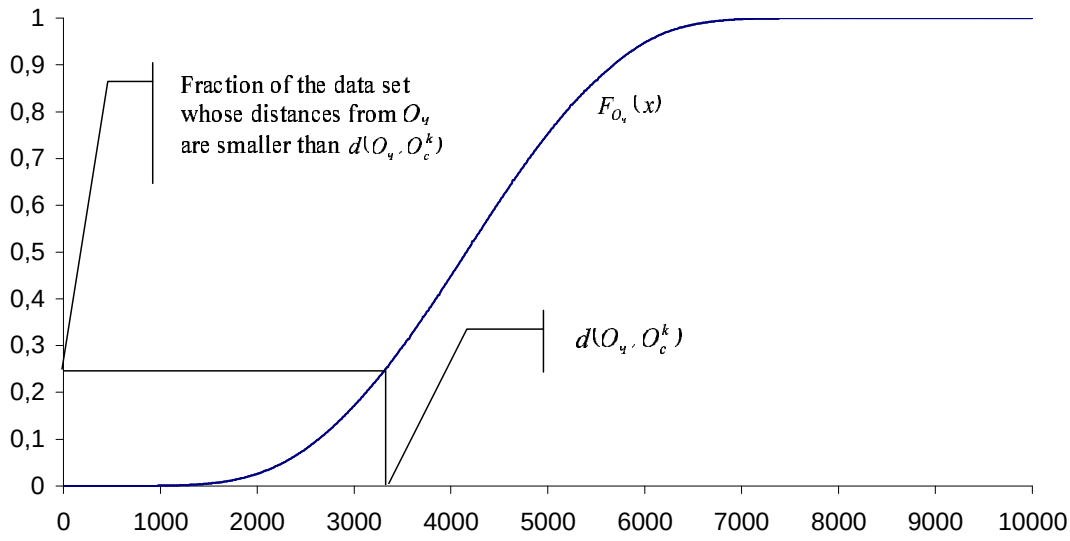


Figure 6.10: An estimation of the fraction of the objects closest to O_q, whose distances from O_q are smaller than $d(O_q, O_c^k)$, can be obtained by using $F_{O_q}(x)$.

discussions, we have that $F_{O_q}(d(O_q, O_c^k))$ corresponds to the fraction of objects in $\mathcal{DS}$ whose distances from O_q are smaller than or equal to $d(O_q, O_c^k)$, see Figure 6.10. Since all other objects in RS_c have a distance from O_q smaller or equal than $d(O_q, O_c^k)$ (since O_c^k is the k-th object), then all objects in RS_c are included in that fraction. For instance, when $F_{O_q}(d(O_q, O_c^k)) = 1/200$, RS_c is expected to be included in the set corresponding to the 0.5% of the objects closest to O_q.

This behavior can be exploited to obtain an approximate nearest neighbor search based on a stop condition. In fact, here the user can specify the value of the approximation parameter x_s corresponding to the wanted fraction, and the stop condition is defined as follows:

$$Stop(RS_c, x_s) = F_{O_q}(d(O_q, O_c^k)) \leq x_s$$

where O_c^k is the k-th object in RS_c.

Of course, since the stop condition may only stop the algorithm 6.3.2 earlier, it might only reduce the cost with respect to the exact similarity search. However, when $x_s < F_{O_q}(d(O_q, O_N^k))$, where O_N^k is the exact k-th nearest neighbor, no improvement is obtained and the exact similarity search is performed. In particular, when $x_s = 0$ the exact similarity search is always performed.

So far, we have assumed that the distribution function F_{O_q} is known. However computing and maintaining this information for any possible query object is totally unrealistic – query objects are not known a priori. A solution is to use, instead of F_{O_q}, the overall distance distribution defined as $F(x) = \mathtt{Pr}\{d(\mathbf{O_1}, \mathbf{O_2}) \leq x\}$, where $\mathbf{O_1}$ and $\mathbf{O_2}$ are random objects of $\mathcal{DS}$. In fact, as discussed in Section 2.3.4, F can be reliably used in these cases as a substitute of any F_{O_q}, given the high index of homogeneity of viewpoints in typical data sets. Furthermore, obtaining F does not present hard problems since it should be computed only once in $O(n^2)$, where n is the size of the considered data set.

The resulting stop condition uses the overall distance distribution F and is consequently defined as follows:

$$Stop(RS_c, x_s) = F(d(O_q, O_c^k)) \leq x_s$$

The pruning condition performs the usual exact overlap test, since this approximation method is only based on a stop condition:

$$Prune(\mathcal{B}(O_q, r_q), \mathcal{B}(O_i, r_i), x_p) = d(O_q, O_i) > r_q + r_i.$$

where, of course, the parameter x_p is not used.

6.7.1 Results

Results of the tests performed using the approximation method that exploits distance distribution are shown in Figures 6.11, 6.12 and 6.13.

The approximation threshold x_s specified by the user, is interpreted as the fraction of the objects closest to the query which the current result set should belong to when the algorithm is stopped. Here the approximation threshold ranges in the interval between 0 and 0.07 (that is a fraction corresponding to 7% of the objects closest to the query). The approximation threshold x_p for the pruning condition is not used, as we said previously.

A general observation is that the approximate algorithm degrades its performance when the number of objects retrieved increases. In fact when k is set to one, the improvement of efficiency almost arrives up to 800 (that is almost three orders of magnitude), while when k is set to 50 the improvement of efficiency arrives up to 45.

Very good results were obtained with HV1 and HV2, even though HV1 gave the best performance. With these two data sets, on average, improvement of efficiency up to 2 orders of magnitude was obtained, still maintaining good quality search results. However, in UV was difficult to obtain improvements comparable with the other two data sets. For instance, in HV1 the approximate algorithm can find the nearest neighbor 423 times faster with $EP = 0.004$. This means that, if the exact algorithm needs 7 minutes to find the nearest neighbor, the approximate algorithm needs only one second to find an approximate nearest neighbor that is, on average, the 40-th (out of 10000) actual nearest neighbor. In HV2, the approximate nearest neighbor can be found on average 100 times faster with an error on the position of 0.004. On the other hand, in the UV data set the nearest neighbor can be found only 45 times

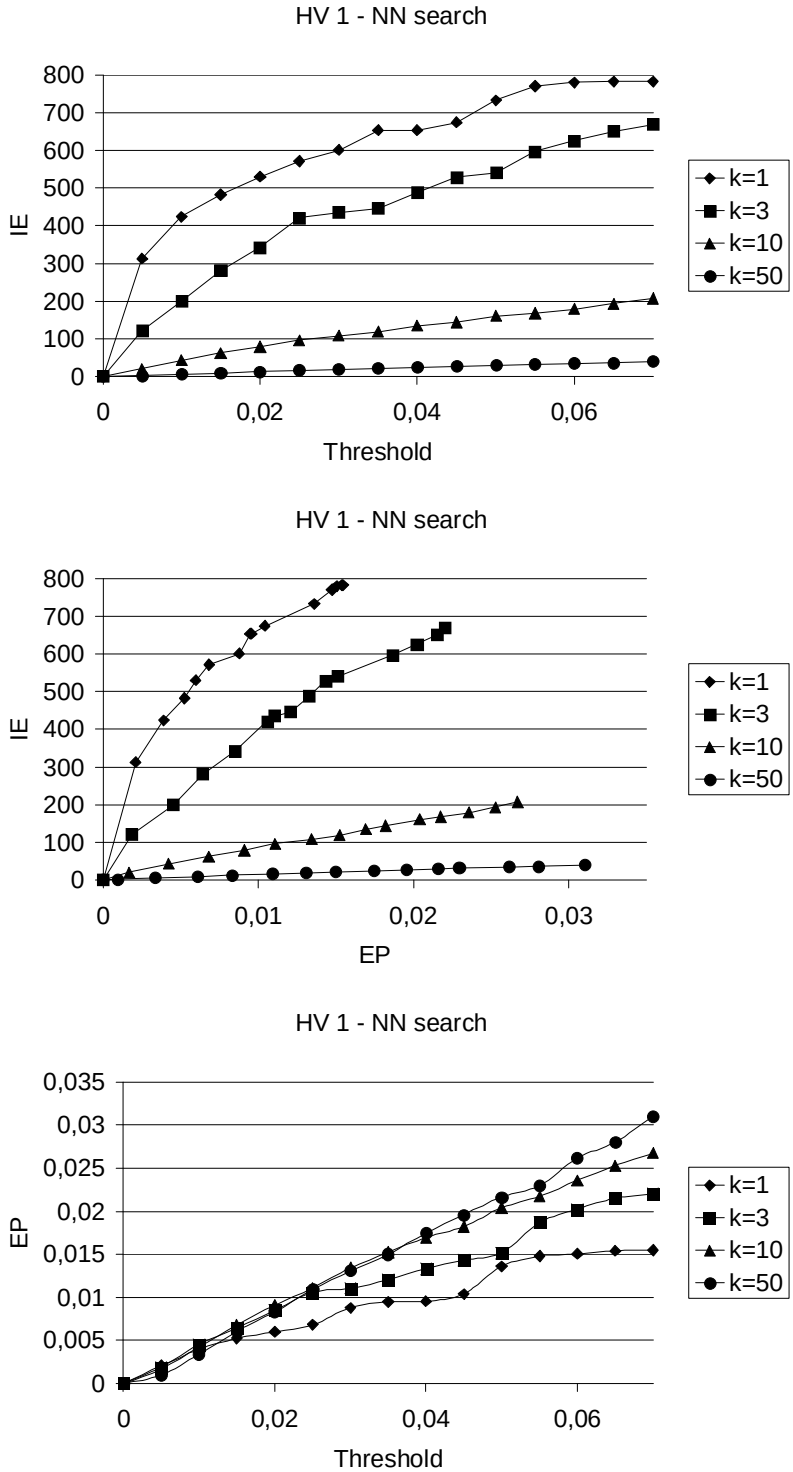
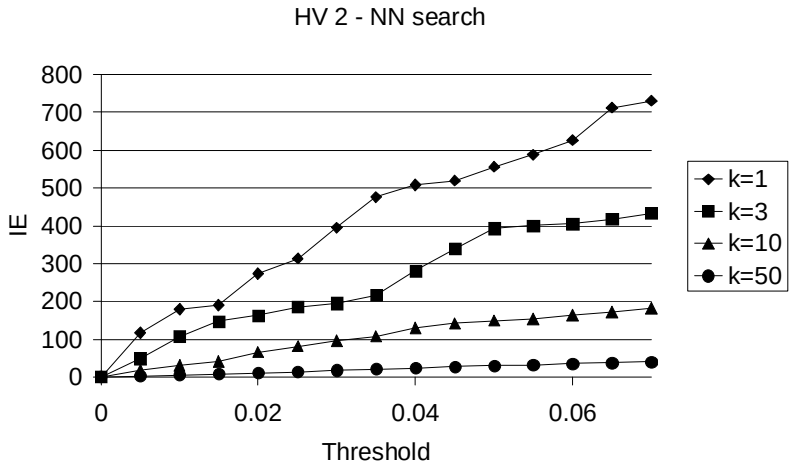
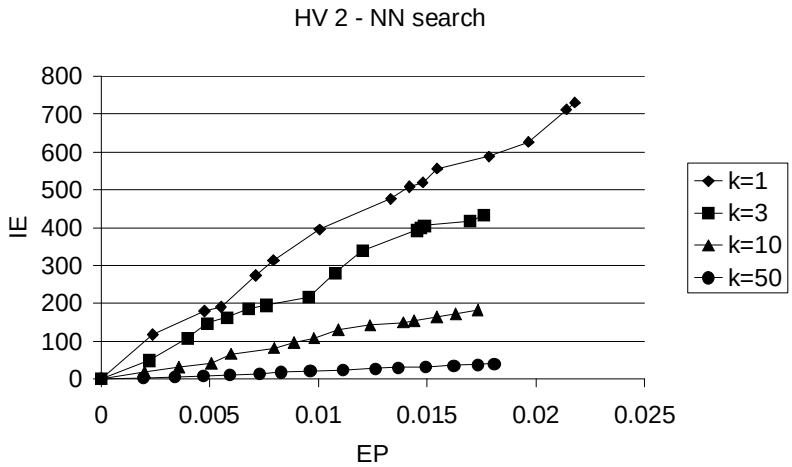


Figure 6.11: Improvement of efficiency (IE) as a function of the derivative threshold (x_s) and the position error (EP). Nearest neighbor queries, HV1 data set.





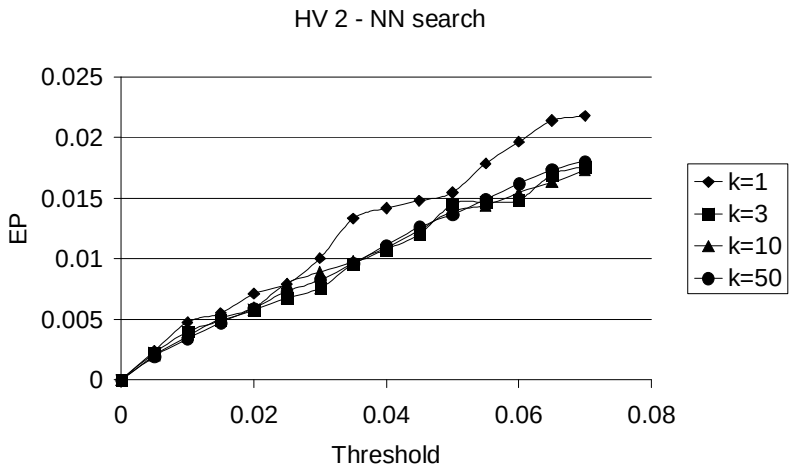


Figure 6.12: Improvement of efficiency (IE) as a function of the derivative threshold (x_s) and the position error (EP). Nearest neighbor queries, HV2 data set.

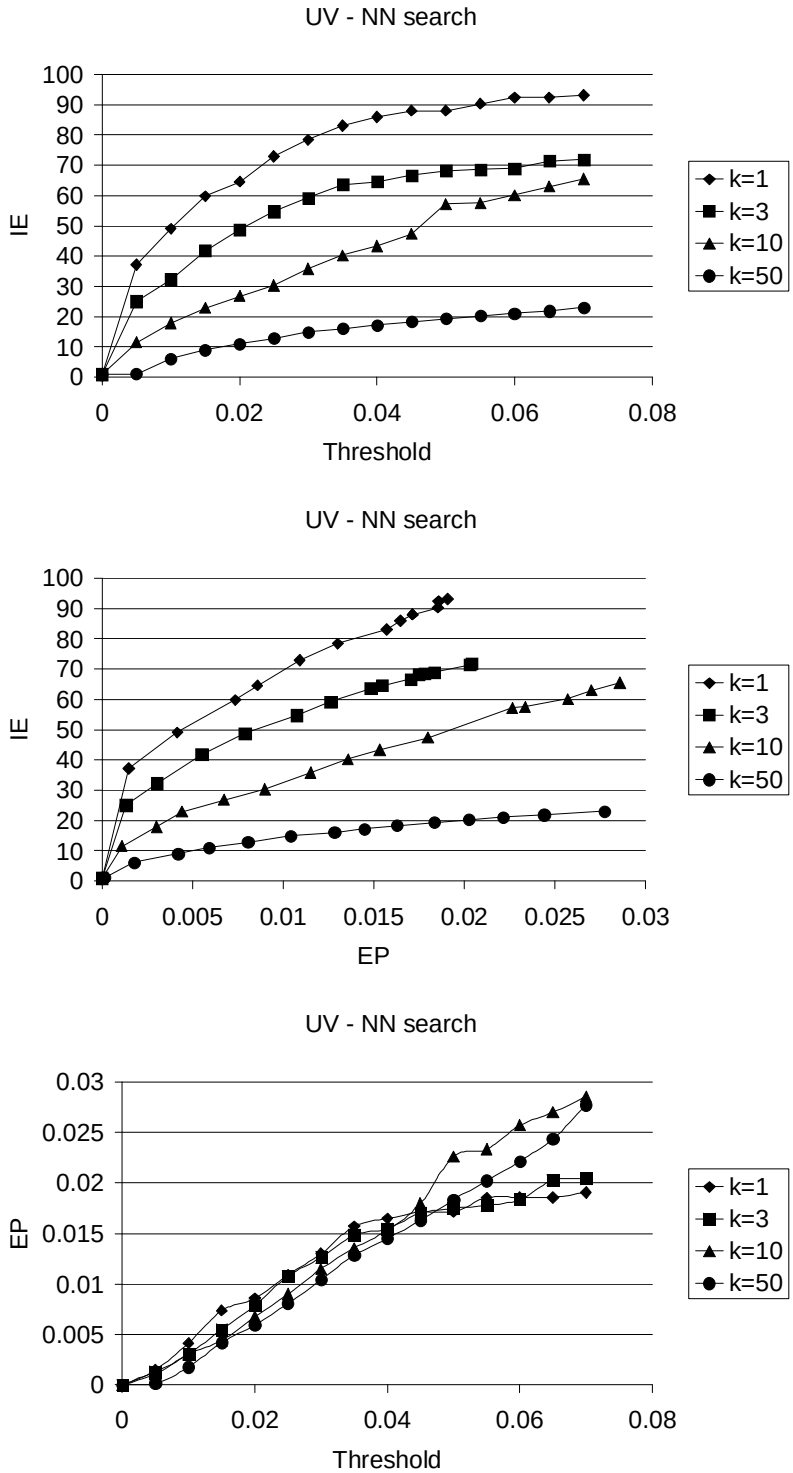


Figure 6.13: Improvement of efficiency (IE) as a function of the derivative threshold (x_s) and the position error (EP). Nearest neighbor queries, UV data set.

faster with $EP = 0.004$.

Figures 6.11, 6.12, and 6.13 show also the dependency between the approximation threshold and the EP measure. All curves are close enough to allow the user to control the quality of the approximation almost independently of k. Notice also that, since the dependency is almost linear, the user can linearly control the quality of approximation using the threshold parameter. However the improvement of efficiency, as we said before, strictly depends on the value of k. So, while the user might ignore k to control the quality of approximation by using the approximation parameter, she should also consider the value of k when she wants to control the improvement of efficiency.

6.8 Method 3: Approximate similarity search using the slowdown of distance improvement

The algorithm for nearest neighbors search (Algorithm 6.3.2) builds the result set through several iterations. In every iteration a new current result set can be obtained by improving the one resulting from the previous iteration. In fact, in every iteration some of the k current nearest objects, found in previous iterations, are possibly replaced by new nearest objects found in current iteration. This improvement can be observed as a reduction of the distance $d(O_k, O_q)$ of the k-th current object O_k from the query object. In fact, distances of the new objects found from the query object are shorter than distances of the replaced objects.

Let us define $d_{it}^{O_q,k}(iter)$ as follows

$$d_{it}^{O_q,k}(iter) = d(O_k(iter), O_q)/d_m$$

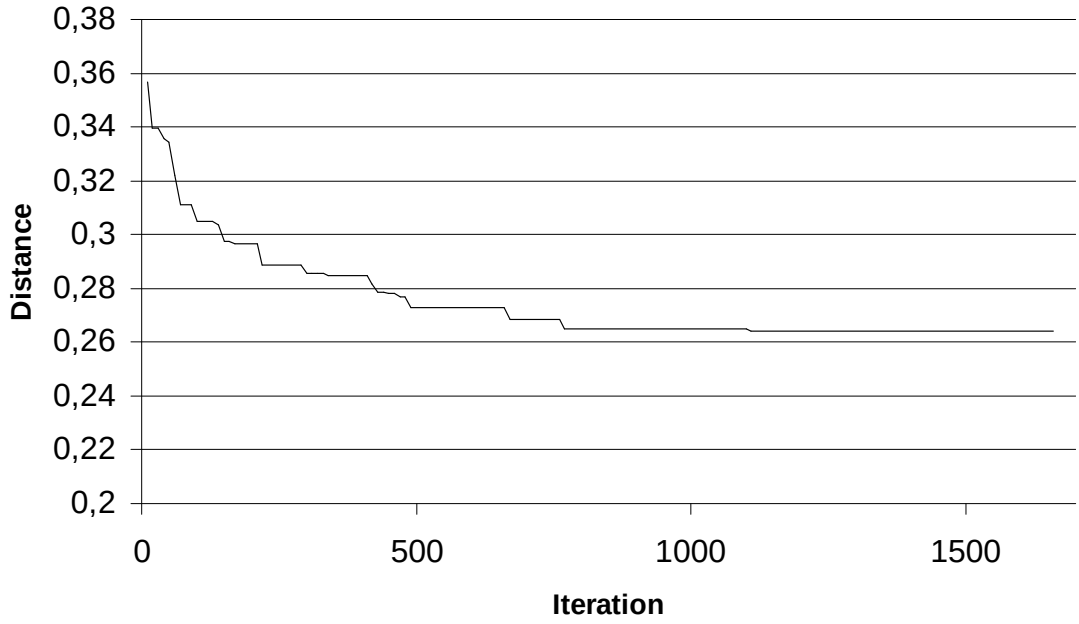


Figure 6.14: Trend of $d_{it}^{O_q,k}(iter)$, when k is 3, in HV1.

where $O_k(iter)$ is the k-th object in the $iter$ iteration and d_m is the maximum distance in the considered data set. $d_{it}^{O_q,k}(iter)$ is the function representing the normalized[4] distance of the current k-th object from the query object O_q at the iteration $iter$. As an example, Figure 6.14 shows the graph of $d_{it}^{O_q,k}(iter)$ for a 3 nearest neighbors search in data set HV1 for a particular query object. Considering this figure we can make some observations. On several iterations, the algorithm is not able to improve the current result set since no better objects are found. In fact, $d_{it}^{O_q,k}(iter)$ mainly assumes constant values in consecutive iterations. The iterations where an improvement of the current result set is obtained are a minority of the whole iterations. It can also be noticed that the improvement is very fast in the first iterations of the algorithm, then

[4]Different data sets have different range of distances. By the normalization, obtained by dividing the actual distance by d_m, the function is defined in a range that does not depend on the particular data set used.

it slows down and almost no improvement, or negligible improvement, occurs after a certain iteration.

We exploit this behavior of the nearest neighbors search algorithm to define a new approximate similarity search algorithm. The idea here is to stop the similarity search algorithm before the natural end, when the improvement slows down below a certain threshold specified by the user. Given the nature of this approximation technique, it cannot be applied to the range search algorithm.

Unfortunately, $d_{it}^{O_q,k}(iter)$ is a function that is not known a priori, in fact, its values become available as the search algorithm proceeds. In addition, as previously observed, it is a piecewise constant function, that is, there are portions where it is rigorously constant. Therefore, it cannot be directly used to effectively infer when improvement slows down. For instance, its derivative would be 0 as soon as in some consecutive iterations no improvement occurs, which may also happen when the algorithm is very far from the final result. To solve these problems we compute, as the algorithm proceeds, a regression curve, say $reg_d(iter)$, which approximate $d_{it}^{O_q,k}(iter)$, and we use it for deciding if the algorithms should be stopped. When the derivative $reg'_d(iter)$ of $reg_d(iter)$ is above[5] the user specified (negative) threshold x_s, in correspondence of the current iteration, the algorithm stops. The parameter x_s is used to control the tradeoff between approximation quality and performance improvement. Of course, small absolute values of x_s result in bad performance but high approximation quality, since the algorithm might stop close to the natural end. Large absolute values of x_s, on the other hand, result in high performance and bad quality, since the algorithm may stop too early, when the current result set is not close to the final

[5]Note that $reg_d(iter)$ will be defined to be monotonically decreasing so it will always have a negative derivative

result set. When the threshold is set to 0 the algorithm behaves as the exact nearest neighbors search, since the regression curve is defined in such a way that it has never a positive derivative.

This method is only based on the stop condition so the pruning condition performs the usual (exact) overlap test to discard a node only when its bounding region does not overlap the query region:

$$Prune(\mathcal{B}(O_q, r_q), \mathcal{B}(O_i, r_i), x_p) = d(O_q, O_i) > r_q + r_i$$

The stop condition, on the other hand, iteration after iteration, refines the regression curve and checks if the derivative of the regression curve, on the current iteration, is above the approximation threshold:

$Stop(RS_c, x_s) =$ if this is the first iteration
set $iteration = 1$
else
set $iteration = iteration + 1$;
let O_k be the k-th element of RS_c;
compute $reg_d(iter)$
using the new point $(iteration, d(O_k, O_q))$
in addition to the points previously used;
if $iteration == 1$
return $false$
else
return $(reg'_d(iteration) > x_s)$;

The mathematical details about the computation of the regression curve $reg_d(iter)$, which approximates the trend of the improvement of distances, is discussed in next

subsection.

6.8.1 Approximating the improvement of distances by a regression curve

The function $d_{it}^{O_q,k}(iter)$ cannot be directly used to decide when the search algorithms can be stopped, since it is a discrete piecewise constant function. However, $d_{it}^{O_q,k}(iter)$ has a shape that can be effectively approximated by a continuous function. We obtain such an approximation by performing the linear regression [Hal52] of points of $d_{it}^{O_q,k}(iter)$ using the method of the discrete least squares approximation [BFR78].

Let us suppose that we know the values of a discrete function $f(i)$ when $i = x_1, ..., x_j$. Then, we can chose n $(n < j)$ real functions $\varphi_1(i), ..., \varphi_n(i)$ to obtain, as a linear combination, a regression curve $\varphi(i)$ that approximate $f(i)$:

$$\varphi(i) = c'_1 \cdot \varphi_1(i) + ... + c'_n \cdot \varphi_n(i)$$

The least squares method says that $c'_1, ..., c'_n$ should be chosen such that $\phi(c'_1, ..., c'_n)$, defined as follows, is minimum:

$$\phi(c_1, ..., c_n) = \sum_{i=1}^{j} \left(\sum_{s=1}^{n} c_s \cdot \varphi_s(x_i) - f(x_i) \right)^2$$

Let us see how this can be applied to our problem. Let us suppose that search algorithm arrived at iteration j. This means that we have the pairs $(i, d_{it}^{O_q,k}(i))$, for $1 \leq i \leq j$, where i corresponds to the i-th iteration and $d_{it}^{O_q,k}(i)$ to the distance of the k-th object from the query in the i-th iteration. These points can be used to obtain a regression curve that approximate $d_{it}^{O_q,k}(iter)$.

In our case the regression curve is obtained by using two real functions $\varphi_1(i)$ and $\varphi_2(i)$. The specific definition of $\varphi_1(i)$ is discussed later, while $\varphi_2(i) = 1$. Therefore, our regression curve $reg_d(iter)$ has the following form:

$$reg_d(iter) = \varphi(iter) = c_1 \cdot \varphi_1(iter) + c_2$$

According to the least square approximation method we should find c_1' and c_2' such that $\phi(c_1', c_2')$, defined as follows, is minimum:

$$\phi(c_1, c_2) = \sum_{i=0}^{j} \left(c_1 \cdot \varphi_1(i) + c_2 - d_{it}^{O_q,k}(i) \right)^2$$

It can be easily shown that if $\varphi_1(i)$ is chosen such that $\varphi_1(i) \geq 0$ when $1 \leq i \leq j$, then $\phi(c_1, c_2)$ is minimum when its partial derivatives are equal to 0. This corresponds to find c_1' and c_2' as a solution to the following:

$$\begin{cases} \frac{\partial \phi(c_1,c_2)}{\partial c_1} = 2 \left(c_1 \sum_{i=1}^{j} g^2(i) + c_2 \sum_{i=1}^{j} \varphi_1(i) - \sum_{i=1}^{j} \varphi_1(i) d_{it}^{O_q,k}(i) \right) = 0 \\ \frac{\partial \phi(c_1,c_2)}{\partial c_2} = 2 \left(c_1 \sum_{i=1}^{j} \varphi_1(i) + j \cdot c_2 - \sum_{i=1}^{j} d_{it}^{O_q,k}(i) \right) = 0 \end{cases}$$

that is

$$c_1' = \frac{j \sum_{i=1}^{j} \varphi_1(i) d_{it}^{O_q,k}(i) - \left(\sum_{i=1}^{j} d_{it}^{O_q,k}(i) \right) \left(\sum_{i=1}^{j} \varphi_1(i) \right)}{j \cdot \sum_{i=1}^{j} g^2(i) - \left(\sum_{i=1}^{j} \varphi_1(i) \right)^2}$$

and

$$c_2' = \frac{\left(\sum_{i=1}^{j} d_{it}^{O_q,k}(i) \right) \left(\sum_{i=1}^{j} g^2(i) \right) - \left(\sum_{i=1}^{j} \varphi_1(i) d_{it}^{O_q,k}(i) \right) \left(\sum_{i=1}^{j} \varphi_1(i) \right)}{j \cdot \sum_{i=1}^{j} g^2(i) - \left(\sum_{i=1}^{j} \varphi_1(i) \right)^2}$$

Notice that the number of iterations j, might be so high that computing $reg_d(iter)$ might be inefficient. However, here we used an optimization. As we said previously,

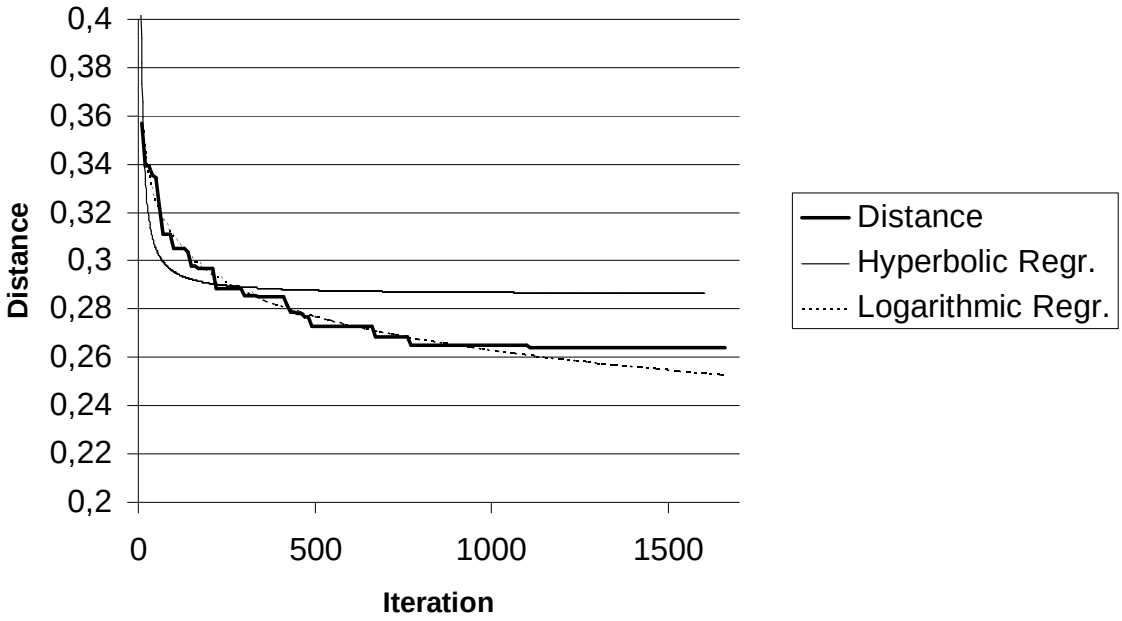


Figure 6.15: Trend of $d_{it}^{O_q,k}(iter)$, when k is 3 in HV1, and two possible regression curves.

$d_{it}^{O_q,k}(iter)$ is constant for several consecutive iterations. Therefore, instead of using all points, $reg_d(iter)$ can be computed by only using points where the value of $d_{it}^{O_q,k}(iter)$ changes. Specifically, $reg_d(iter)$ is computed using a set of points obtained as follows. Let i be the current iteration and $(i, d_{it}^{O_q,k}(i))$ the last point added to the set of points. At iteration $i+1$, if $d_{it}^{O_q,k}(i) = d_{it}^{O_q,k}(i+1)$, then the point $(i, d_{it}^{O_q,k}(i))$ is replaced by $(i{+}1, d_{it}^{O_q,k}(i{+}1))$, elsewhere, the point $(i, d_{it}^{O_q,k}(i))$ is maintained and $(i{+}1, d_{it}^{O_q,k}(i{+}1))$ is added to the list of points. This reduces enormously the number of points to be considered, maintaining a good approximation of $d_{it}^{O_q,k}(iter)$.

We still have to discuss the definition of function $\varphi_1(i)$. Its choice depends on the peculiar properties of $d_{it}^{O_q,k}(iter)$. In fact, different choices of $\varphi_1(i)$ might give different approximation qualities depending on their capability of miming the trend

of $d_{it}^{O_q,k}(iter)$. Given the typical shape of $d_{it}^{O_q,k}(iter)$, we have tried the hyperbola, that is $\varphi_1(i) = 1/i$ and the logarithm, that is $\varphi_1(i) = log(i)$. Figure 6.15 shows graphically $d_{it}^{O_q,k}(iter)$ and the two resulting approximations. We have used both choices in the approximate similarity search algorithm, and the average behaviour was practically the same in both cases. In fact, both choices gave almost overlapped results in term of IE and EP, even tough different ranges of the threshold parameters x_s had to be used. Next subsection presents the obtained results.

6.8.2 Results

We have tested the the approximate similarity search algorithm with the slow-down of distance improvement, by obtaining the regression curve through both the logarithm and the hyperbola as a definition for $\varphi_1(i)$. As previously anticipated, the obtained results were practically the same, so here we only discuss those obtained by using the logarithm. The same observations hold also for the other possibility considered. Results are summarized in Figures 6.16, 6.17 and 6.18.

In these experiments the approximation threshold x_s, is interpreted as a threshold on the derivative of $reg_d(iter)$, the curve that approximate the trend of distance of the current k-th object from the query object. The approximation threshold ranges in the interval between 0 and -0.004. When the derivative of $reg_d(iter)$, in the current iteration, is above the specified threshold, the algorithm stops. The approximation threshold x_p for the pruning condition is not used, since the pruning condition performs the exact overlap test.

A first general observation is that, even if the improvement of efficiency might arrives up to two orders of magnitude, the approximation threshold x_s is not easy

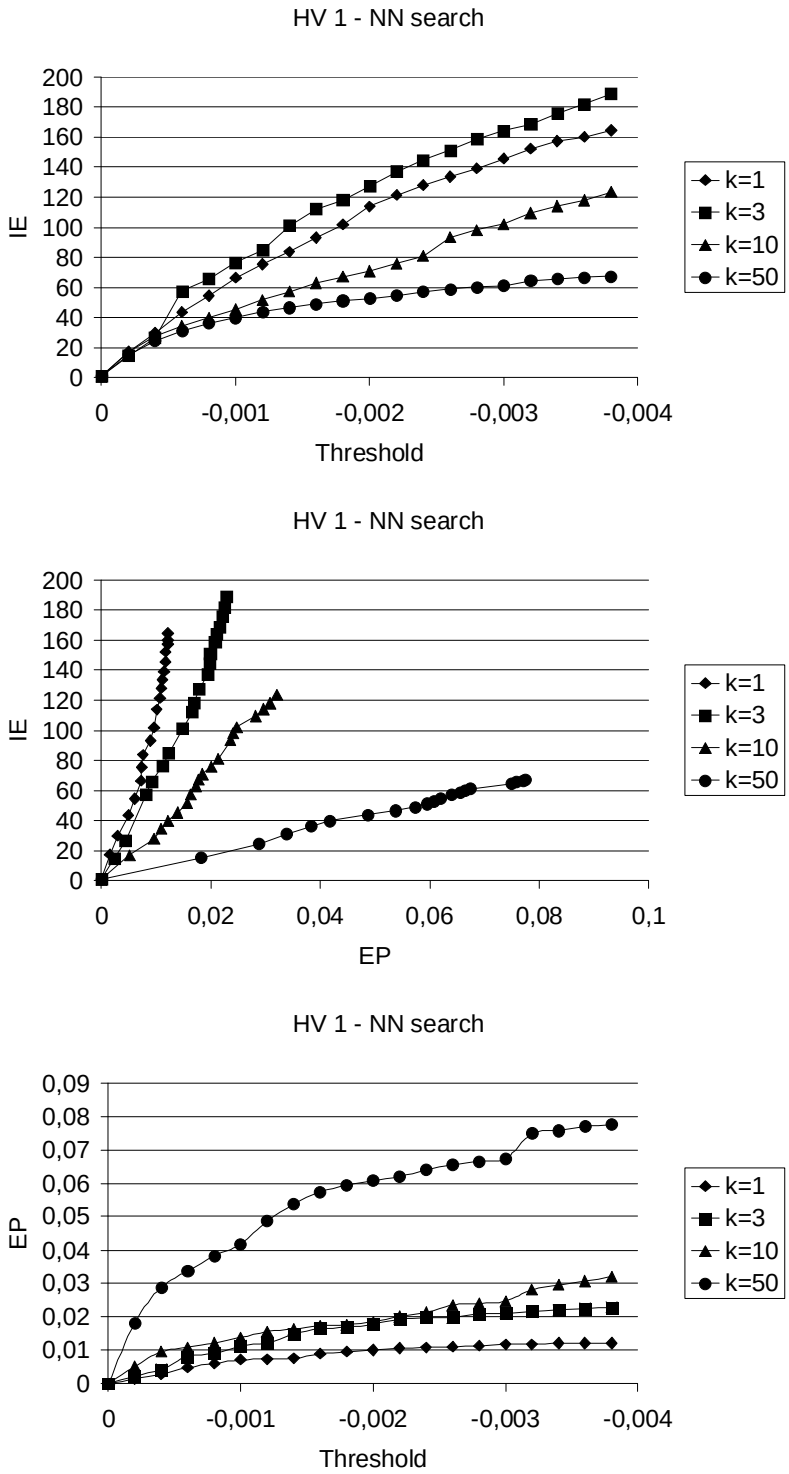


Figure 6.16: Improvement of efficiency (IE) as a function of the derivative threshold (x_s) and the position error (EP). Nearest neighbor queries, HV1 data set.

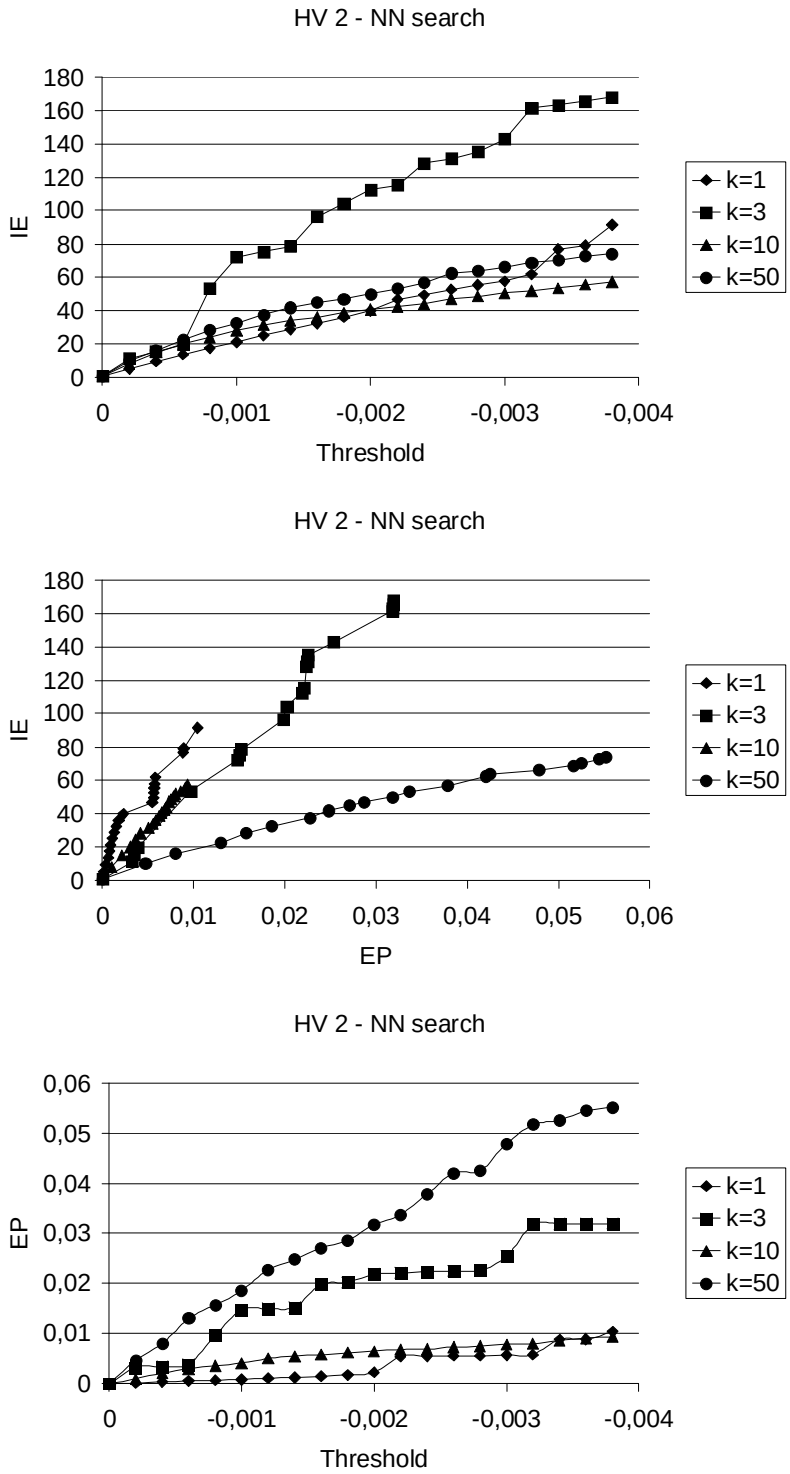


Figure 6.17: Improvement of efficiency (IE) as a function of the derivative threshold (x_s) and the position error (EP). Nearest neighbor queries, HV2 data set.

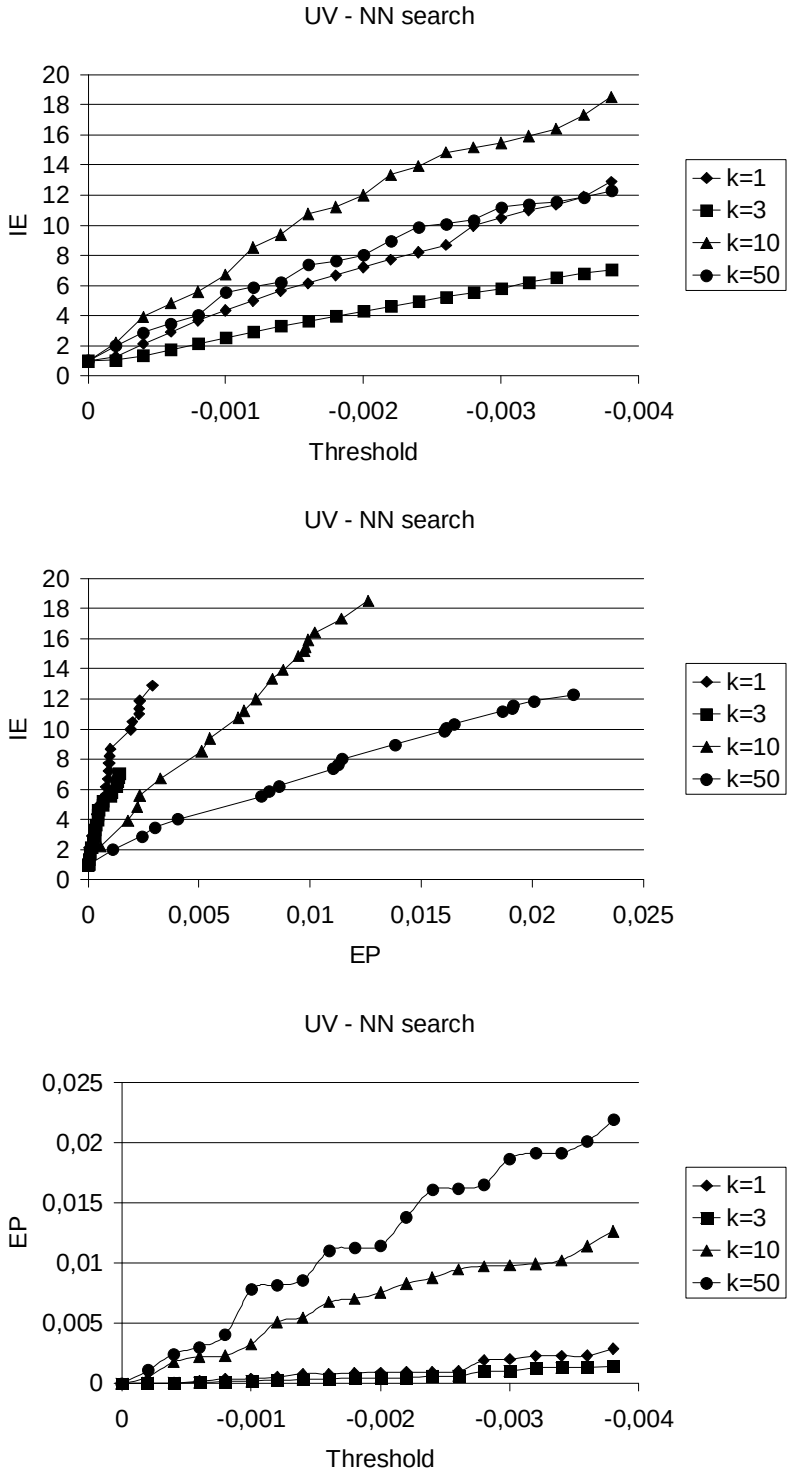


Figure 6.18: Improvement of efficiency (IE) as a function of the derivative threshold (x_s) and the position error (EP). Nearest neighbor queries, UV data set.

to be used to control the tradeoff between performance and effectiveness, when the number of retrieved objects k is varied. For instance in HV1, given a fixed threshold, the best improvement of efficiency where observed for $k = 3$, then $k = 1$, $k = 10$, and, finally, $k = 50$. That is, the improvement of efficiency depends on k but the corresponding relationship is not trivial. However, the error on the position EP, with the exception of data set UV, is almost directly proportional to k. In fact EP degrades as k increases when the approximation threshold is fixed.

Let us now discuss more in details the obtained results. An improvement of efficiency up to 2 orders of magnitude was obtained, maintaining a relatively good quality search results. In fact, in HV1 the nearest neighbor can be obtained on average 190 times faster, with $EP = 0.023$. Performances obtained for HV1 and HV2 are comparable, even though slightly better results are obtained for HV2. However, in UV results are clearly worse. The approximate algorithm can find the nearest neighbor 60 times faster with $EP = 0.006$ in HV1 and $EP = 0.005$ in HV2. This means that, for instance in in HV1, the approximate nearest neighbor is, on average, the 60-th nearest neighbor (out of 10000). However, when the precise search takes 1 minute to compute, the approximated result is obtained in 1 second. Similar relationships between HV1 and HV2 can be observed also for other values of k. On the other hand, in the UV data set, using the same range of approximation threshold, the improvement of performance arrives only up to one order of magnitude and the results where typically worse than in the other data sets. In fact, in UV, the nearest neighbor was found 13 times faster with $EP = 0.0025$, while in HV1 and HV2 the nearest neighbor was found with the same value of EP respectively 30 and 40 times faster.

6.9 Method 4: Approximate similarity search using the region proximity

Access methods for metric spaces partition the searched data, and bound element of the partition by ball regions. In order to find qualifying objects, search algorithms should access all nodes of the tree corresponding to regions that overlap the query region. All regions that overlap the query region may potentially contain objects that are also covered by the query regions. Search algorithms might ignore all regions that do not overlap the query region in order to be more efficient, since accessing a node (corresponding to a region) has a high cost. However, notice that when a data region and the query region overlap with each other, there is no guarantee that objects are included in their intersection. In fact depending on the data distribution, it may happen that the intersection covers a portion of the space containing very few objects (or no objects at all). Therefore, some of the data regions that overlap the query region may not contain searched data, and some regions are more likely to contain the query response than the others.

In Figure 6.19, it can be seen that, although the query region Q intersects regions $\mathcal{R}_1$, $\mathcal{R}_2$ and $\mathcal{R}_3$, the intersection with $\mathcal{R}_1$ and $\mathcal{R}_3$ is empty and thus it is not necessary to access these regions.

Proximity between ball regions, discussed in Chapter 4 may help to handle this situation. In fact, as defined by Equation 4.2.2, proximity of two regions is the probability that objects can be found in their intersection. The idea here is to use the proximity to decide when data regions should be accessed to find qualifying objects, in fact, the higher the proximity, the more "interesting" the regions.

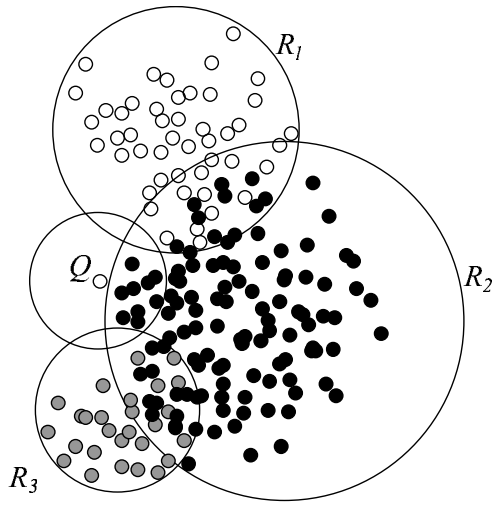


Figure 6.19: Overlap between the query region and data regions: not all data regions that overlap the query region share objects with it.

This idea constitutes the basis for the approximate similarity search by using proximity. In fact, approximated results are obtained by considering, in the similarity search algorithms, only regions with proximity greater than a certain threshold, say x_p, i.e. regions in which the probability of containing a qualifying object is greater than x_p. The approximation threshold x_p is specified by the user and can be used to control the approximation. High values correspond to high performance but bad quality. Small values corresponds to low performance but high quality. When x_p is set to 0, an exact similarity search is performed, since proximity is greater than 0 as soon as two regions overlap.

Approximate range search and approximate nearest neighbor search, which use this specific strategy, can be obtained from Algorithms 6.3.1 and 6.3.2 by defining the pruning condition and the stop condition as follows.

The pruning condition $Prune(\mathcal{B}(O_q, r_q), \mathcal{B}(O_i, r_i), x_p)$ is defined as

$$Prune(\mathcal{B}(O_q, r_q), \mathcal{B}(O_i, r_i), x_p) = X(\mathcal{B}(O_q, r_q), \mathcal{B}(O_i, r_i)) < x_p \quad (6.9.1)$$

The stop condition on the other hand is always false:

$$Stop(RS_c, x_s) = false$$

The proximity $X(\mathcal{B}(O_q, r_q), \mathcal{B}(O_i, r_i))$ between the query region and the selected data regions was computed using the parallel heuristic, described in 4.5, which proved to be the more precise among the ones tested.

6.9.1 Results

Results of the experiments using the proximity based approximate similarity search algorithms are summarized in Figures 6.20, 6.21 and 6.22 for range queries and in Figures 6.23, 6.24 and 6.25 for nearest neighbor queries.

In these experiments the approximation threshold x_p, is interpreted as the proximity threshold. When the proximity of the query region and a data region is below the specified threshold, the data region is discarded. The proximity threshold ranges in the interval between 0 and 0.06. The approximation threshold x_s for the stop condition is not used, since the stop condition is always false.

The results show that the best improvement of efficiency is achieved when the size of the result set is small. The number of retrieved objects is explicitly specified for the nearest neighbors queries, but is quite difficult to control by specifying a range. Note that the response sets for our range queries contains on average more than 100 objects (1% of 10,000). In fact, the approach offers better performance for nearest neighbors queries with small k rather than for the range queries.

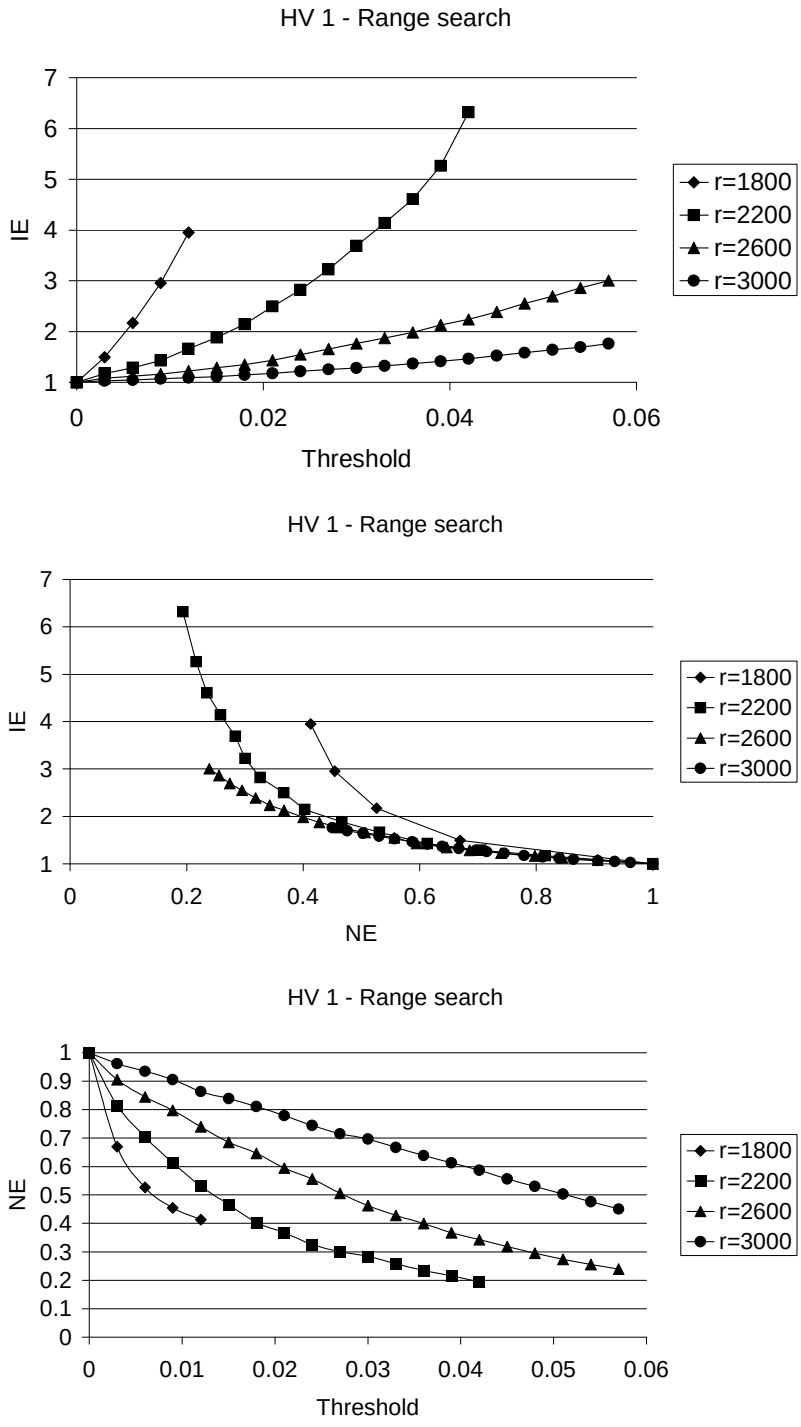
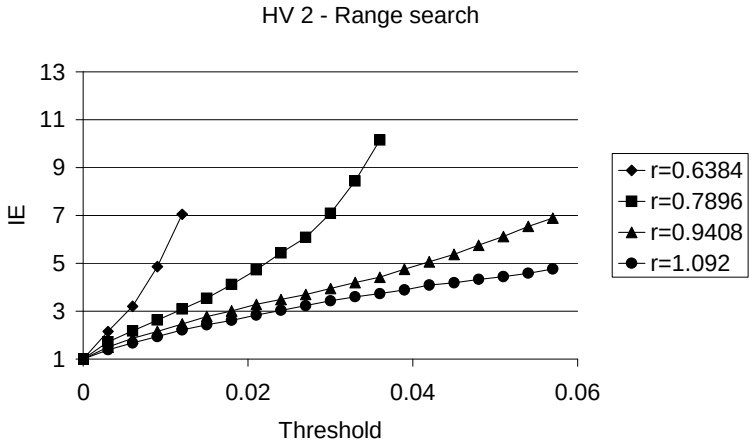
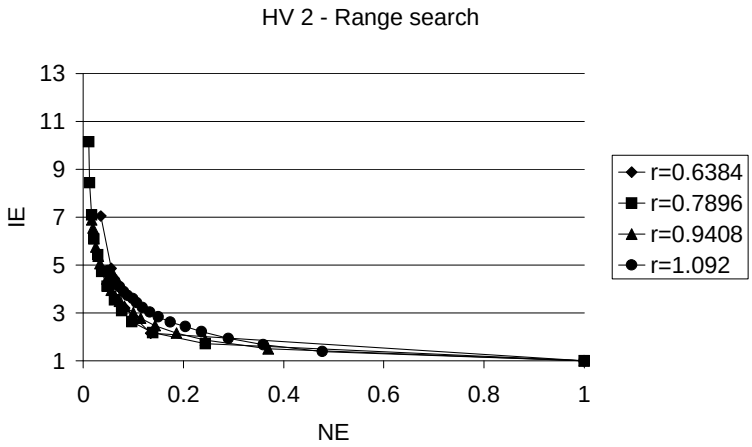


Figure 6.20: Improvement of efficiency (IE) as a function of the proximity threshold (x_p) and the fraction of exact results (NE). Range queries, HV1 data set.





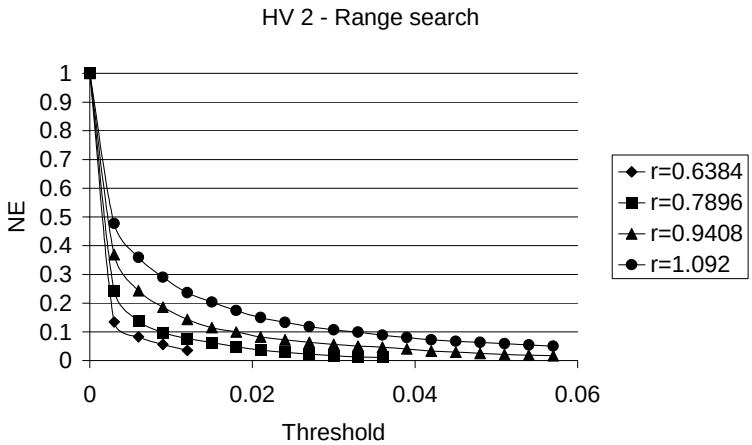


Figure 6.21: Improvement of efficiency (IE) as a function of the proximity threshold (x_p) and the fraction of exact results (NE). Range queries, HV2 data set.

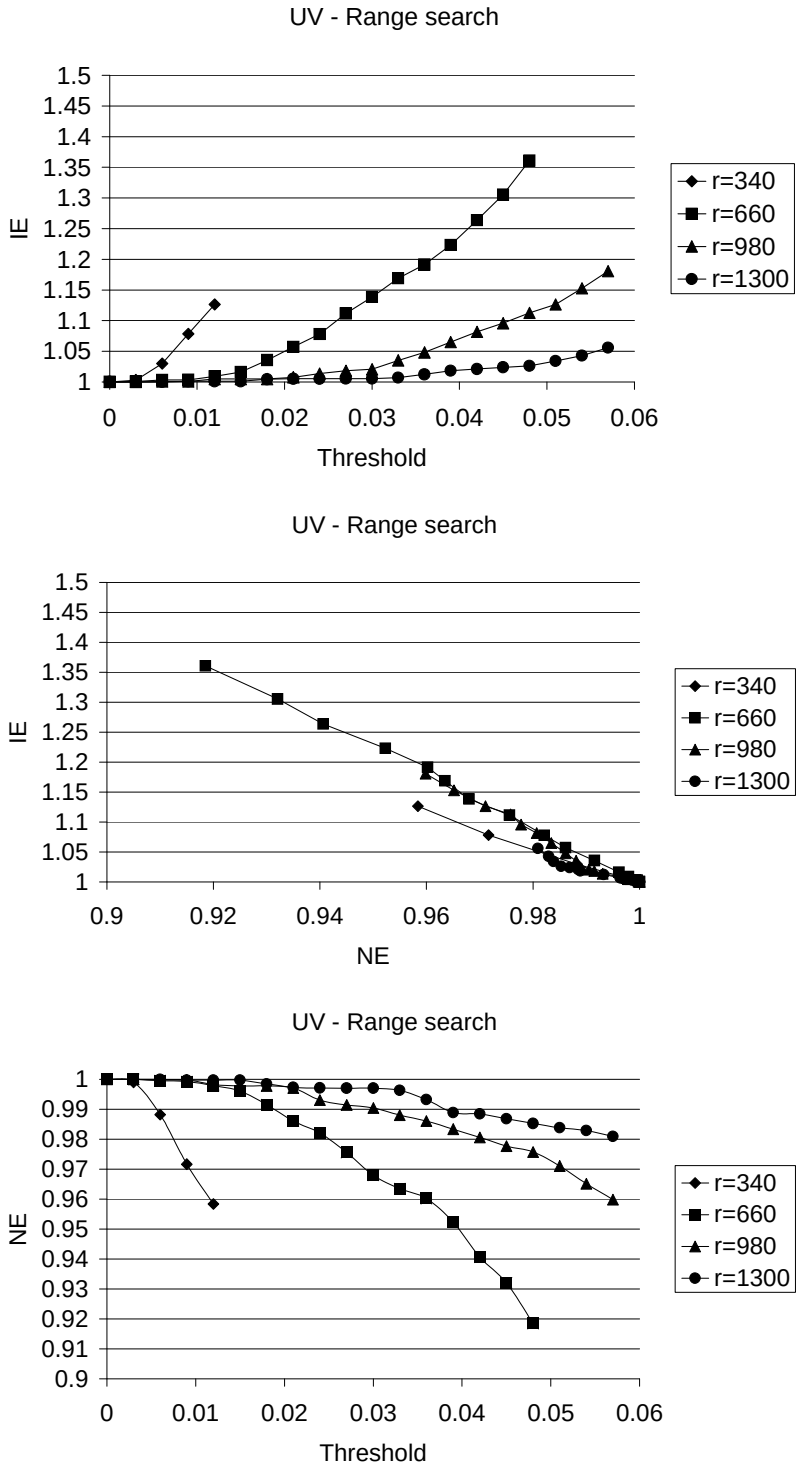
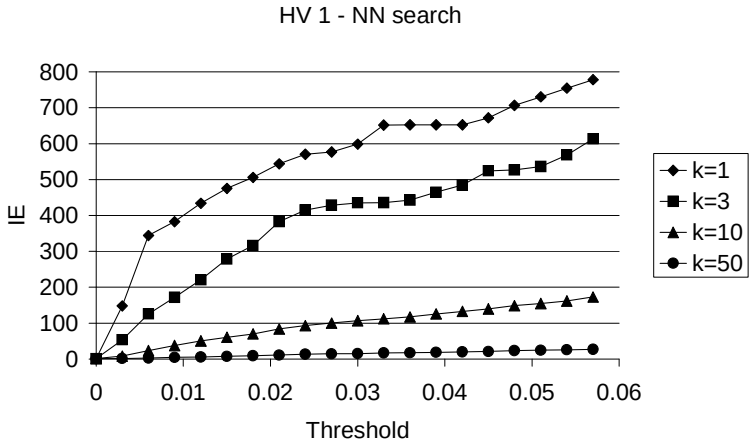
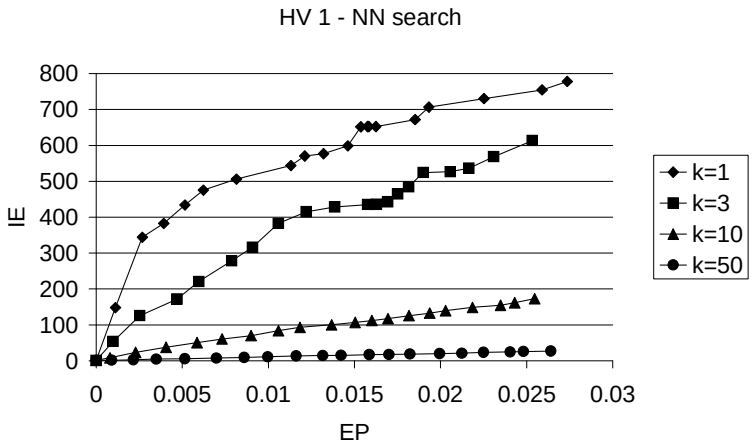


Figure 6.22: Improvement of efficiency (IE) as a function of the proximity threshold (x_p) and the fraction of exact results (NE). Range queries, UV data set.





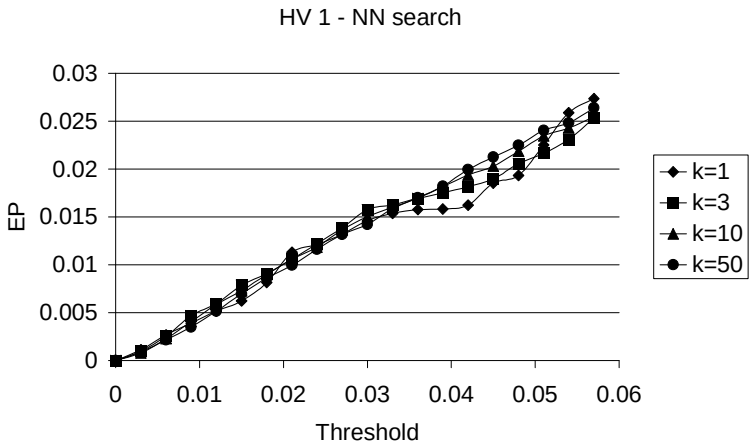


Figure 6.23: Improvement of efficiency (IE) as a function of the proximity threshold (x_p) and the position error (EP). Nearest neighbor queries, HV1 data set.

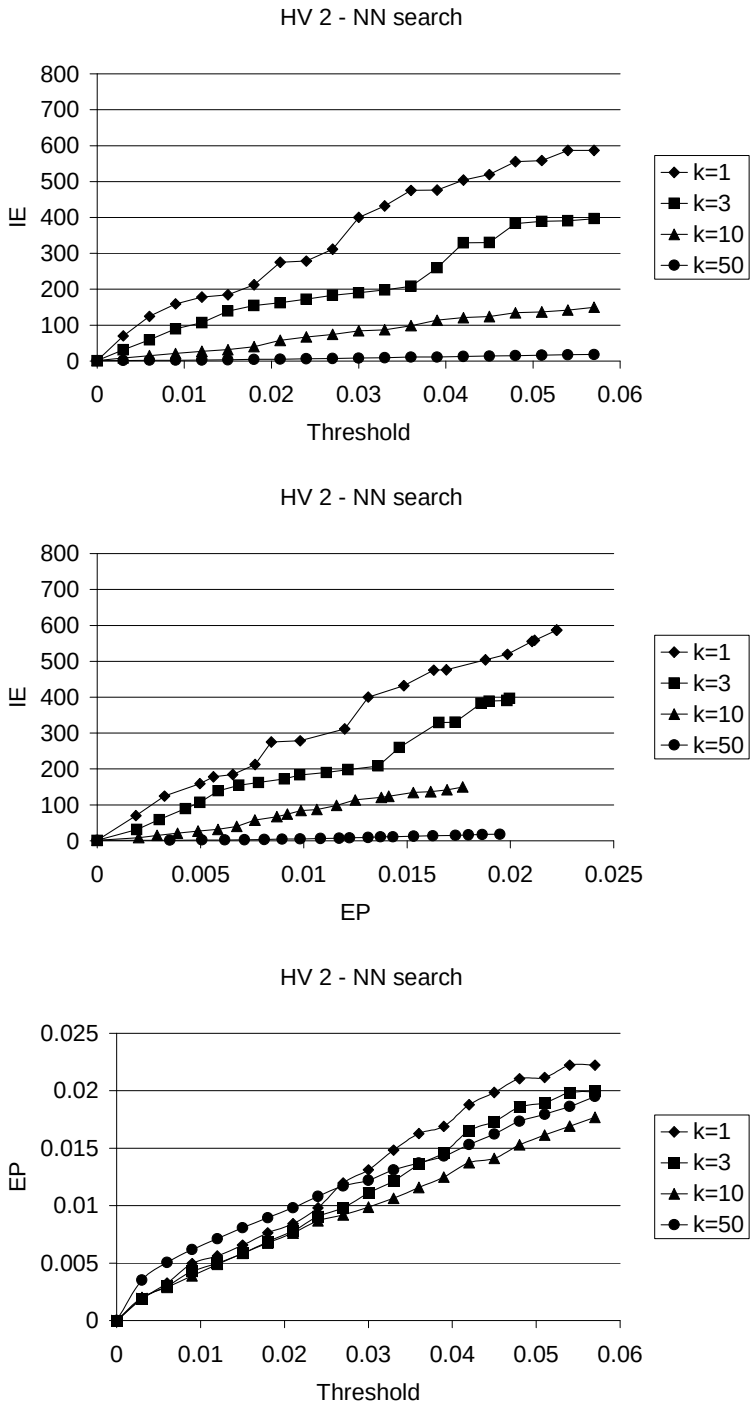
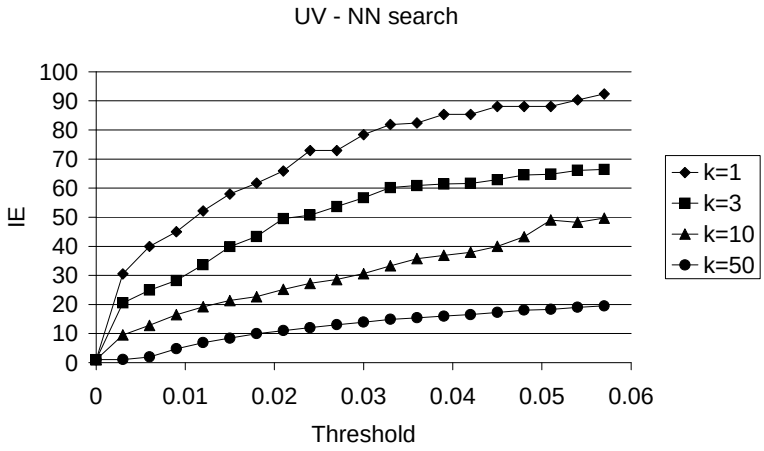
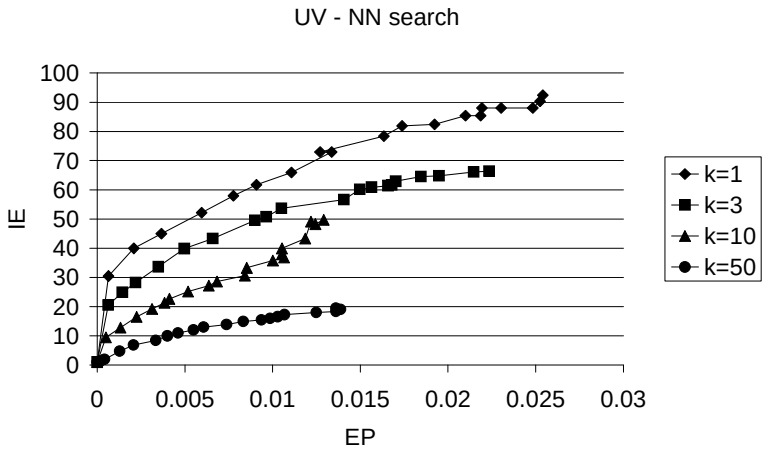


Figure 6.24: Improvement of efficiency (IE) as a function of the proximity threshold (x_p) and the position error (EP). Nearest neighbor queries, HV2 data set.





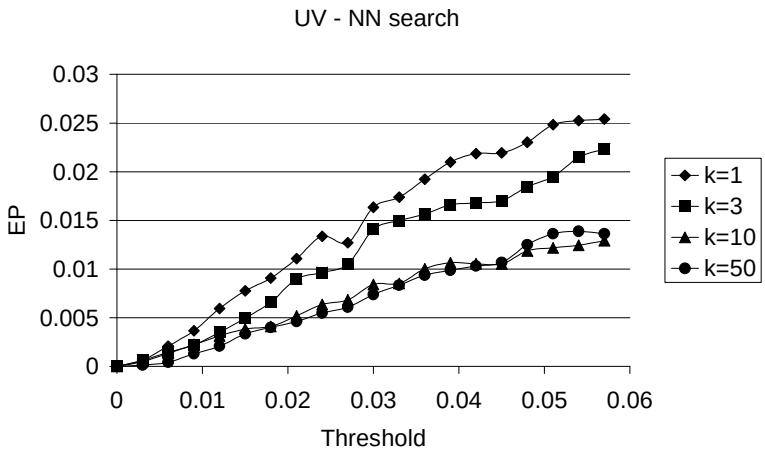


Figure 6.25: Improvement of efficiency (IE) as a function of the proximity threshold (x_p) and the position error (EP). Nearest neighbor queries, UV data set.

Let us now consider the range queries more closely. The improvement of efficiency decreases when the query radius grows. In all data sets, the improvement obtained is less than one order of magnitude. As expected, IE increases as the NE measure decreases. Note that we do not report results for $NE = 0$. Results seem to be better for the HV1 data set rather than the HV2 and UV data sets. In HV1, for instance, when the query radius is 2200, a query can be executed on average 6.5 times faster, with $NE = 0.2$, that is 20% of objects retrieved by the exact search were found by the approximate search. When the HV2 data set is used, results are slightly better in terms of an improvement of efficiency, however the NE measure returns worse results than for the HV1 data set. In fact, improvement of efficiency higher than 3 typically results in NE close to 0. For all radii considered, when the improvement of efficiency is 6.5, the NE measure is about 0.05. In case of the UV data set it was not possible to obtain improvement of efficiency higher than 1.4. In fact, when the approximation parameter was chosen a little bit larger suddenly the number of retrieved objects was 0. Next sub-section gives an explanation for this behaviour.

In the case of nearest neighbor queries, results are much better. In fact, an improvement of efficiency up to 2 orders of magnitude was obtained, still maintaining good quality search results. There is no significant difference between HV1 and HV2, even though slightly better results are obtained for HV1. However, in UV was again difficult to obtain improvements comparable with the other data sets. For instance, the approximate algorithm can find the nearest neighbor in HV1 60 times faster with $EP = 0.0005$. This means that the approximate nearest neighbor is on average actually the 5-th nearest neighbor. However, provided the precise search takes 1 minute to compute, the approximated result is obtained in 1 second. If the requirements

regarding precision are not so high, the approximate algorithm can perform much faster. For example, a 300 times faster approximate search implies an error in position $EP = 0.003$. In this case, the approximate nearest neighbor is the 30-th actual neighbor, but even queries which would require 5 minutes to get precise results, can be performed through approximation in 1 second. On the other hand, in the UV data set the nearest neighbor can be found 60 times faster with $EP = 0.008$. That is, the approximate nearest neighbor is the 80-th nearest neighbor.

Notice the dependency between the proximity threshold and the EP measure in Figures 6.23 and 6.24. All curves are almost overlapped and this suggests that it is possible to control the effectiveness by using the threshold parameter, independently of the value of k. In addition, since the dependency is almost linear, it means that the user can linearly control the quality of approximation using the threshold parameter. However, in UV this linear dependency is less marked, see Figure 6.25, and no profitable dependency between the threshold and NE can be observed in case of the range queries, see Figures 6.20, 6.21, and 6.22.

We have also tested our approach to approximate similarity search by substituting the probabilistic proximity with the trivial one, defined by Equation 4.2.1. Not only did much higher values of approximation threshold have to be used, but also the performance of approximated queries with respect to trivial proximity was systematically worse. When the same values of efficiency are considered, it is evident that the quality of approximation for the probabilistic approach is much higher than for the trivial approach. For example, see Figure 6.26 where the error on the position EP and improvement of efficiency IE are related for $k = 10$, separately for HV1 and

HV2 data sets. In HV1, IE increases almost linearly with EP for the probabilistic approach, while IE is almost constant up to values of $EP = 0.01$ and then it increases slowly. In HV2, again, IE increase almost linearly with EP for the probabilistic approach, while IE is almost constant up to $EP = 0.04$ and then it increases, but more slowly than in HV1. For example, when $EP = 0.01$, the probabilistically approximated query performed 70 times faster than the precise query and about 25 times faster than the approximated query with the trivial proximity. Such behavior was identical both for HV1 and HV2.

6.9.2 Further observations

When the query radius is small, the query response set can become empty, even for small values of the proximity threshold. Though it might look strange, the explanation is easy and provisions can be made to avoid such situations to happen.

It is easy to show that the proximity of two ball regions is smaller than or equal to the probability that a randomly chosen point belongs to the smaller of the two regions. Such probability can be approximated by $F(r)$ (see Section 2.3.4), where F is the overall distance distribution and r is the radius of the smaller region. Since for small values of the query radius $F(r_q)$ is very small, the proximity between the query region and any other region is also small. When $x_p > F(r_q)$, the pruning condition prunes every node of the tree. In fact, in this case, proximity between the query region and any other region is always smaller than x_p. Notice that the result is anyway correct, but empty approximated result is meaningless. In order to avoid such situations, the relationship between the proximity threshold and the query radius must be respected.

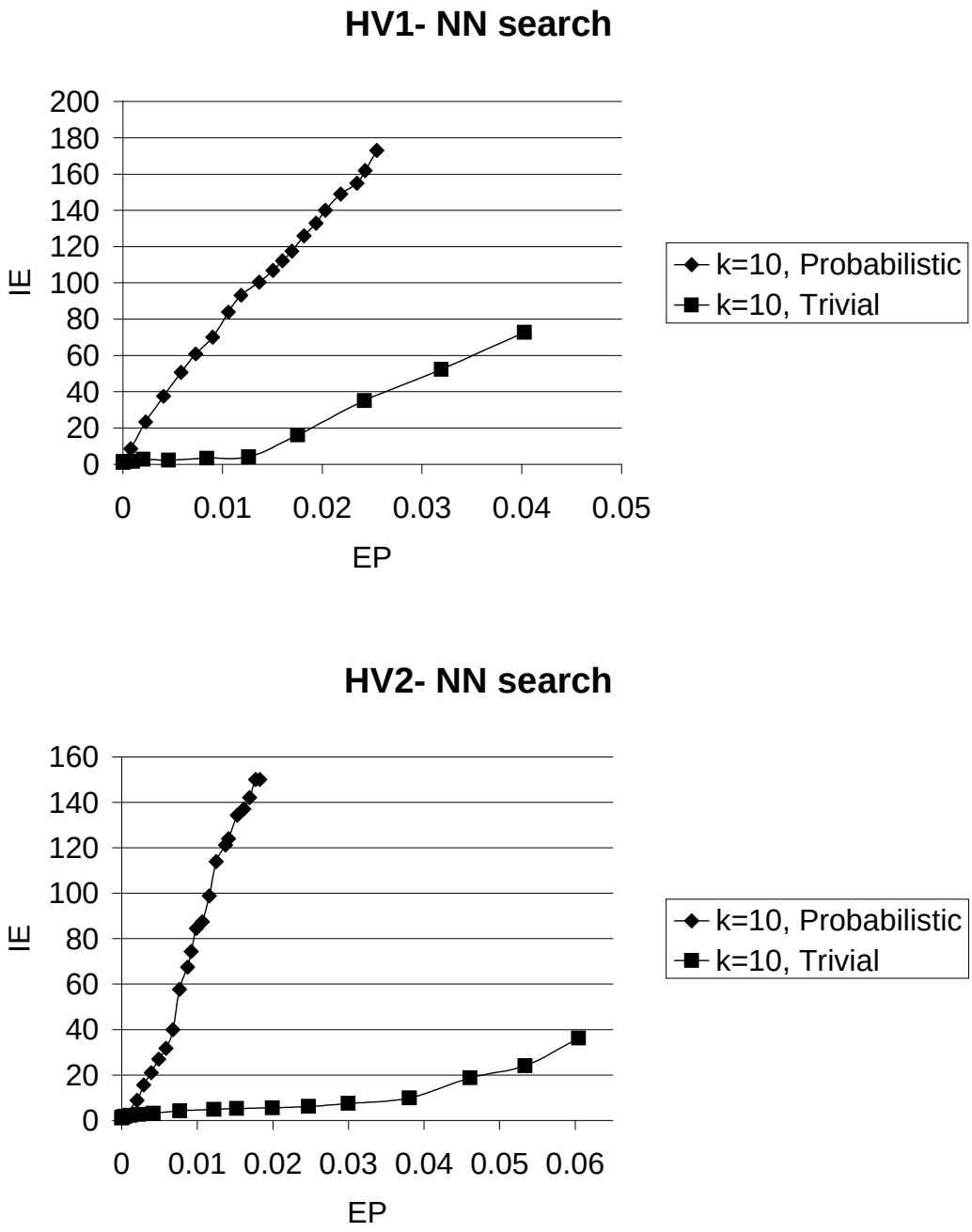


Figure 6.26: Comparison of the trivial and probabilistic approximation techniques

Several strategies can be used to find an appropriate query specification, because the distance distribution function F is known. For example, the system can suggest the smallest threshold that can be specified for a given query radius. Vice-versa, given a threshold, the system can tell which is the smallest radius that can be used with that threshold. Another possibility is to have the threshold automatically corrected (normalized) by the system. For instance the pruning condition $X(\mathcal{B}(O_q, r_q), \mathcal{B}(O_i, r_i)) < x_p$, can be substituted by $X(\mathcal{B}(O_q, r_q), \mathcal{B}(O_i, r_i))/F(r_q) < x_p$, obtaining an automatic normalization.

This problem is not relevant for nearest neighbors queries. In fact, for this type of similarity query, the radius of the query region is dynamically changed as the algorithm proceeds. It is very big at the beginning and it becomes small just towards the end of query evaluation, so the influence of the threshold on the number of requested objects is less evident and a response set of k objects is always provided. However, in case of nearest neighbors queries, this phenomenon has another effect. In fact, when the dynamic radius of the query region reduces such that $x_p > F(r_q)$, then all remaining regions are pruned and, since the queue of pending requests become immediately empty, the algorithm immediately stops. The result is that, even if this method is defined as an approximate pruning condition, due to this phenomenon, this method implicitly acts also as if a stop condition was defined.

6.10 Cross comparisons

Previous sections were devoted to the individual description of the various approximation methods proposed in this thesis and to the analysis of the experimental results obtained by testing them. In this section, on the other and, we compare together all

methods proposed and we discuss their advantages and disadvantages. First a direct comparison of the performance of the various methods is presented, separately for range and nearest neighbors search. Then some global considerations not strictly related to performance are discussed.

6.10.1 Range queries

Figure 6.27 compares the results obtained by executing range queries. Only the first approximation method, the one based on the relative error on the distances, and the fourth, the one based on the proximity, are presented, since only these two methods can be used for range queries. To simplify the presentation, we do not show the comparison for each radius considered in the tests, but results obtained for only one radius are shown. In particular the comparison is presented by using the third radius used in each data set, specifically 2600 in HV1 , 0.9408 in HV2, and 980 in UV. The graphs in the figure relate the NE measure with the improvement of efficiency IE.

In HV1 the best results are achieved by the fourth method (proximity). In fact, given any value of NE, the corresponding value of IE is always higher for the proximity based method than for the other. In HV2 the method based on the relative error of distances is better. However, for small values of NE, that is when the accuracy of the approximation is low, the trend of fourth method is to rapidly increase the performance in term of improvement of efficiency. In UV, again, the best results are given by the fourth method. In fact, in this case, it offers high values of IE even when NE is still high.

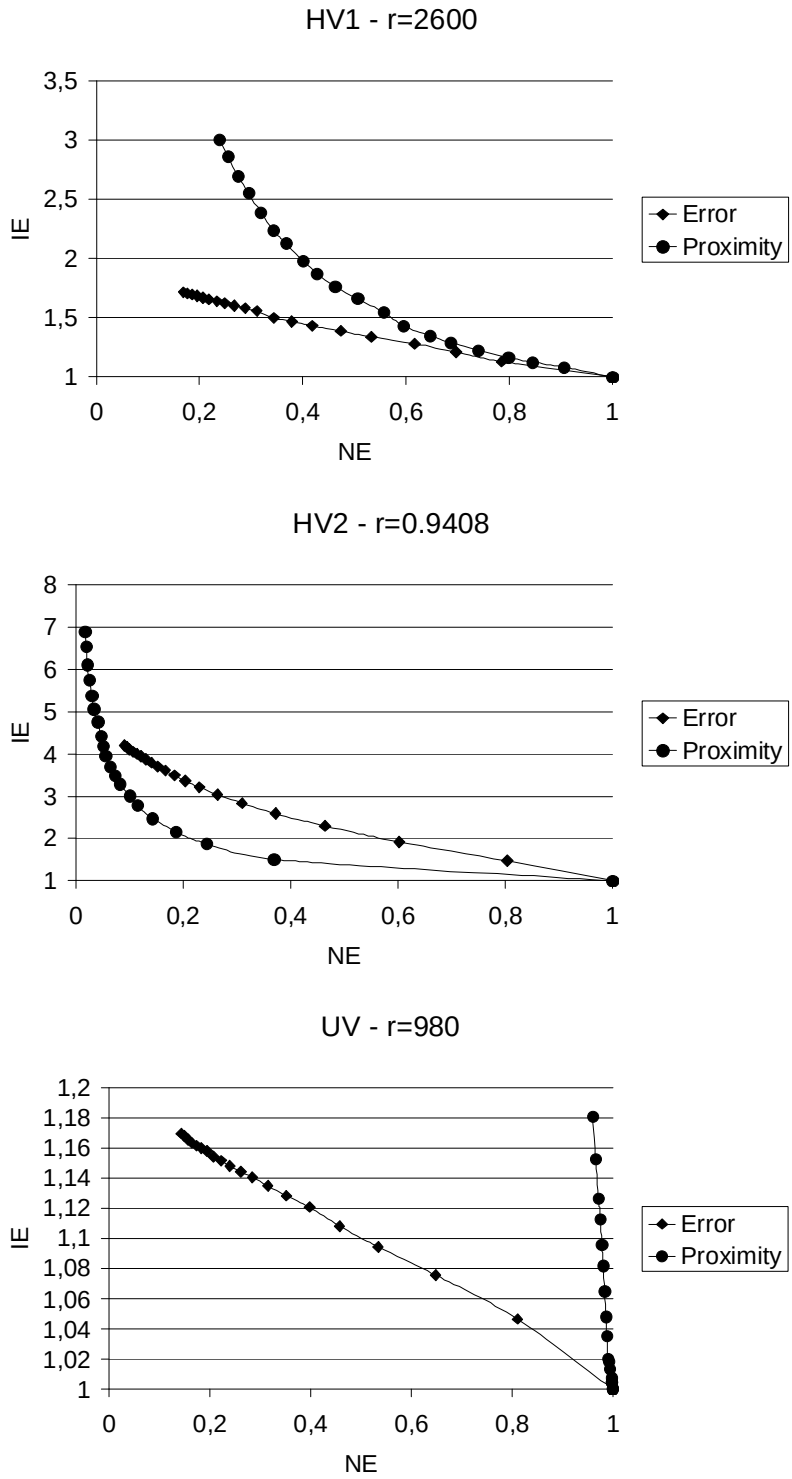


Figure 6.27: Comparison of the approximation methods that support range queries in the various data sets.

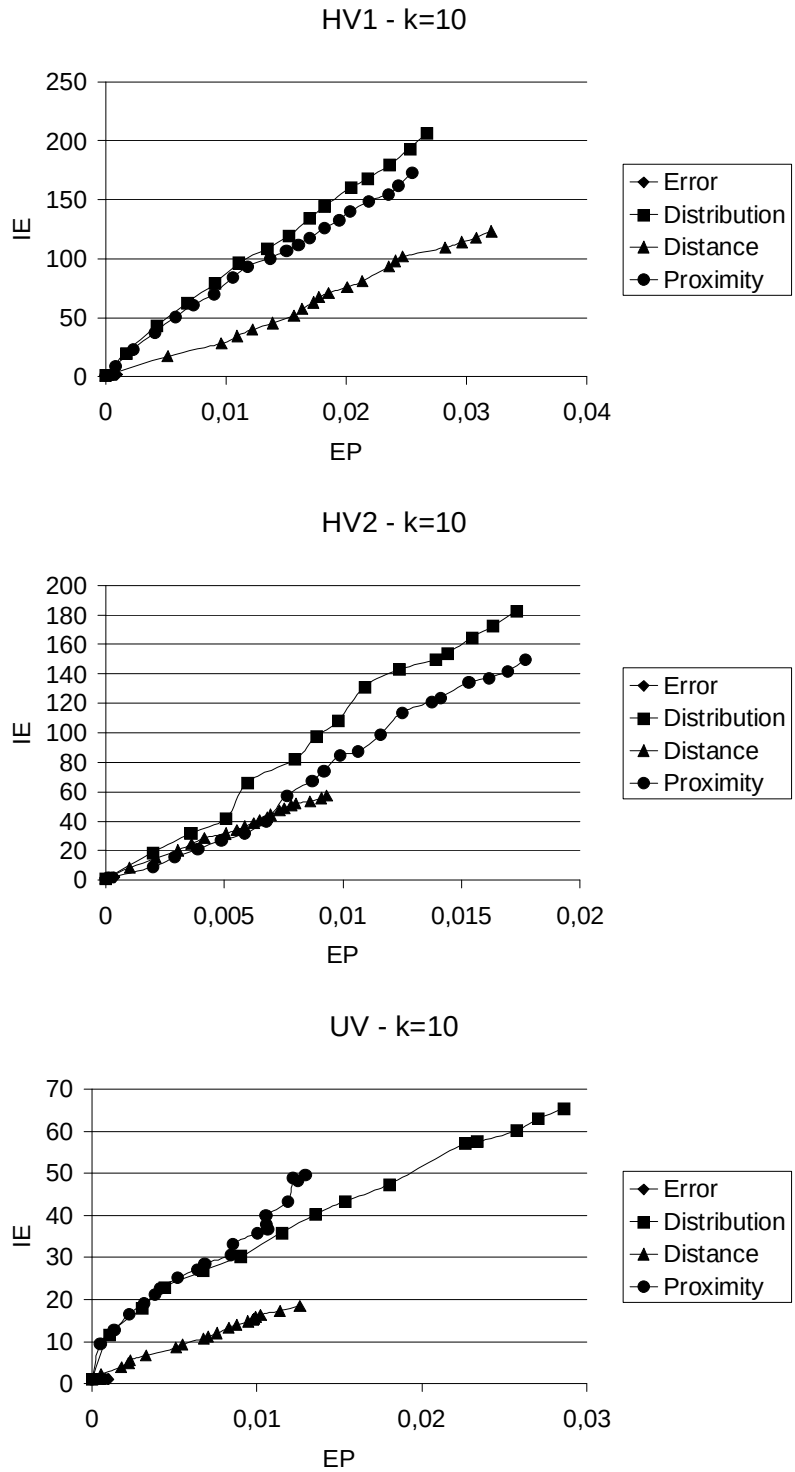


Figure 6.28: Comparison of all approximation methods for nearest neighbor queries in the various data sets.

6.10.2 Nearest neighbors queries

Let us now consider the results of executing nearest neighbor queries. Comparisons are presented in Figure 6.28. All methods can be used to execute nearest neighbor queries so all methods are compared. Also here for simplicity, as we did for range queries, we do not show the comparisons for all k values that were tested. The comparison is presented only for $k = 10$ in all data sets. The graphs presented relate the EP measure with the improvement of efficiency IE.

With exception of the first method (distance error), all methods offer an improvement of efficiency up to two orders of magnitude. The first method offers a very limited improvement in efficiency. Even using high approximation thresholds high improvements cannot be obtained. In the graph, the experimental values of this methods are very close to the origin of the axis and can be barely noticed, given the higher performance of the other methods. As we said in section 6.6, where the first method was presented, it immediately saturates and no relevant improvement is obtained. In HV1 and HV2 the highest performance is offered by the second method (distance distribution), while in UV the best method is the fourth method (proximity). In HV1 and UV, the curves of the second and the fourth method, however, are close to each other, and their performance is definitively better than that of the third method. In HV2, on the other hand, there is no significant difference among the performances of all methods.

Looking at the maximum performance obtained by our approximate nearest neighbors search algorithms in the different data sets, we can see that the improvement of efficiency seems to be more significant for the 45 (that is HV1) rather than the 32-dimensional (that is HV2) vectors. In addition, the improvement in efficiency is

negligible in case of the 2-dimensional vectors (that is UV). We have also experimented with other data sets and the general conclusion is that our methods of approximation for nearest neighbors queries are suitable above all when the precise similarity search tends to access many data nodes in the supporting tree structure. Such a situation often happens when the data partitioning results in highly overlapping regions, which is common for high-dimensional vector spaces. In these cases, the improvement is typically registered in hundreds, which is not possible to achieve for low dimensional spaces where even the exact similarity search algorithms are efficient. For this reason, results with the 2-dimensional UV data set are always worse that those obtained with the other data sets.

As previously pointed out, no clear relationship between the performance and the number of retrieved nearest neighbor was found for the first and the third method in the various data sets. On the other hand, in the second and in the fourth method, performance systematically deteriorates when the number of nearest neighbors retrieved increases. This can be explained as follows. We can observe that at any specific iteration of the search algorithm the distance of the current k-th nearest neighbor from the query object increases as k increases. To illustrate this, consider Figure 6.29 which relates the distance of the current k-th nearest neighbor from the query object and the number of iteration of the exact nearest neighbors search algorithm, separately for $k = 1, 3$, and 10. Observe that these distances for $k = 1$ are systematically below those for $k = 3$, and these are systematically below those for $k = 10$. In case of the second method, this means that the higher k, the larger $F(O_q, O_c^k)$, so more iterations should be executed before that it goes below the approximation threshold. On the other hand, in case of the fourth method, at each iteration of the nearest neighbors

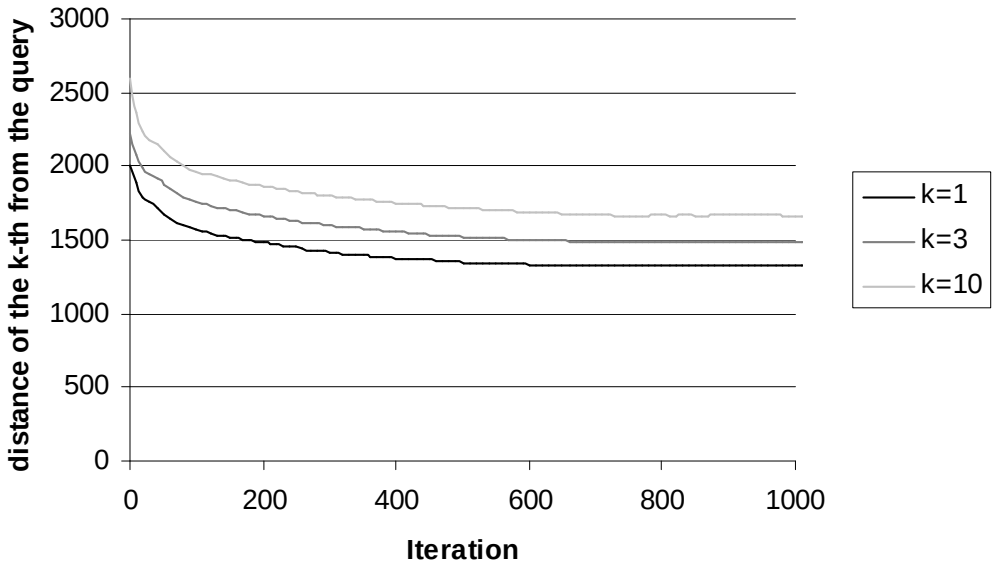


Figure 6.29: Average trend of the distance of the current k-th object from the query object during the exact nearest neighbor search execution in HV1

algorithm, the query radius is set to the distance between the query and the current k-th nearest neighbor. This means that a higher k systematically results in bigger query regions. While searching, the set of data regions is fixed so the bigger the query region, the higher the proximity of specific data and query regions. Consequently, a systematically higher proximity has a greater chance of exceeding the approximation threshold, thus more data regions are accessed.

6.10.3 Global considerations

Summarizing, the approximation methods that we have proposed give moderate improvement of performance for range queries, while very high improvement is obtained for nearest neighbor queries. The method that offers the highest performance is the

second one, however it can only be used for nearest neighbor queries. On the other hand, the fourth method offers similar, even if slightly worse, results for nearest neighbors queries, and it can also be used for range queries. The improvement of performance of the third method also arrives up to two orders of magnitude, but it is always below that of the other two. The first method can hardly be compared with the others given its very low performance. The (minor) drawback of the second and the fourth methods, is that they require to pre-compute, store, and handle distribution functions (both second and fourth) and density functions (the fourth). However, as discussed earlier, this overhead can be easily handled. The first and the third methods do not need any pre-analysis of the data sets and do not require any other storage overhead, however their performance is worse that that of the other two methods and they are also more difficult to be used since, as we said when we discussed in details their results, there is not a clear dependency between the threshold and the improvement of efficiency in the various data sets, when k varies. In conclusion it appears that the most promising method is the fourth one, since it offers very high performance for nearest neighbor queries and it also does a good job in case of range queries.

6.11 Comparison with other techniques

In Section 5.3 we have described the most significant approaches for approximate similarity search, which belong to the category of algorithms reducing the examined result set [AM95, AMN+98, PAL99, PL99, CP00]. In this section we make some performance and qualitative comparisons of our proposed techniques with these techniques.

Notice that a rigorous performance comparison cannot be assessed by using the performance figures reported by the articles describing these techniques. In fact, typically different data sets were used for the specific experiments, different underlying access methods were used, in some cases the efficiency was measured by considering the computational complexity in terms of floating point operations, and different measurements for determining the accuracy were performed. However we are able to make some informal comparison anyway. For what concerns the efficiency, the improvement provided by these techniques is generally lower than that provided by our techniques. Specifically, no other techniques offer improvement of efficiency analogous to that offered by our best techniques, namely the second and the fourth proposed approaches. For what concerns the accuracy, given that some of these methods used as indication of the accuracy the relative distance error, whose use we have contested in Section 6.4, we have also performed some preliminary tests using it, to have similar accuracy figures, and our results confirm that the trade-off between improvement of efficiency and loss of accuracy is generally more advantageous in our cases.

The behaviour of our algorithms is more general than the other techniques. We offer wider applicability since our techniques were defined for generic metric spaces and some of them can be used both for nearest neighbors and range queries. In the following we discuss specifically the limits of the other techniques.

The technique for nearest neighbor searching using BBD trees (see Section 5.3.3 and [AMN+98]) can only be used with data represented in vector spaces and distances measured by using functions of the Minkowski family. Its performance depends exponentially on *dim*. As a consequence the algorithm can be practically used when the vector space has number of dimensions in the range from 2 to 20. For vector spaces

with higher dimension this approach suffer from the dimensionality curse.

The technique for approximate range queries using BBD trees (see Section 5.3.4 and [AM95]), solves the counting version of the range search problem in vector spaces when distances are Minkowski distances. Specifically it can only be used to count or compute the weight of objects contained in the result set of an approximate range query. It does not retrieve the set of objects qualifying for the approximate range query. The counting problem is generally easier to be addressed than the problem of obtaining the set of objects qualifying for a range query, since it might be solved without accessing the, otherwise needed, leaf nodes of the tree, where pointers to real objects are stored. Performance of this algorithm is moderate.

The techniques based on the angle property (see Section 5.3.5 and [PAL99, PL99]) are only able to answer to nearest neighbors search in vector spaces and Minkowski distances. The approach described in [PAL99], in its original definition, cannot be tuned to trade performance with quality of results, in fact it does not use any approximation parameter that can be used by the user to chose the desired degree of approximation. The performance obtained by these algorithms is generally low compared to ours.

The PAC nearest neighbor algorithm (see Section 5.3.6 and [CP00]) can be used in generic metric spaces. However in its original definition it is limited to the case of just a single nearest neighbor search and fails to handle both generic nearest neighbors and range queries. It offers moderate performance, limited to the case of one nearest neighbor search.

Summarizing, with the exception of the PAC nearest neighbor searching technique all other techniques can only be used in vector spaces and distances should be

measured using functions of the Minkowski family. In addition all of them, with the exception of the technique for approximate range queries using BBD trees, can only be used for nearest neighbors queries and in some cases just for one nearest neighbor queries. To the best of our knowledge, there are no techniques that support at the same time range and nearest neighbor queries in generic metric spaces.

Chapter 7

Conclusions

In this thesis, we have dealt with the issue of approximate similarity search in metric spaces. Even though there are several existing access methods that aim at increasing performance of similarity search, current technologies cannot be considered satisfactory and the issue of approximate similarity search has emerged as a relevant research topic. This approach offers higher performance at the price of imprecision in the results.

In its more general definition, similarity search only needs a similarity function to be defined in order to compare data. Metric spaces are thus very suitable as a framework for this problem. Contrary to other formalizations (for instance vector spaces), metric spaces do not pose any restriction on the possible representation of data. They only require that distance functions satisfy the metric postulates: symmetry, positiveness, reflexivity, and triangular inequality.

In the thesis we defined four new approaches for approximate similarity search in metric spaces. In addition, since one of the techniques is based on the measurement of the proximity of ball regions defined in metric spaces, we have also proposed some heuristics to compute this proximity efficiently and accurately.

7.1 Approximate similarity search in metric spaces

We have investigated techniques that relax the problem of similarity search allowing higher performance to be achieved at the price of some imprecision in the result sets. Our analysis shows that imprecision can be effectively controlled, while still guaranteeing very high performance compared to search algorithms on traditional access methods for similarity search. We conclude that approximate similarity search may represent a valid solution to the intrinsic inefficiency of the similarity search problem.

We have proposed and extensively tested four different techniques. All of them were implemented as search algorithms on M-Tree [CPZ97] access methods. However, given their generality, applying them to other tree search structures would be straightforward.

We have specified a number of measurements to assess the tradeoff between the speedup achieved and the quality of approximation. Experimental results on real-life data files are very promising, and efficiency improvement of two orders of magnitude has been achieved for acceptable approximations.

The performance and accuracy of our techniques, especially that of the strategies based on the exploitation of distance distribution (see Section 6.7) and proximity (see Section 6.9), are higher than those obtained by other approximate similarity search techniques proposed in the past (see Section 5.3). In addition our techniques are more flexible. In fact, they are defined for metric spaces, which also includes the case of vector spaces, and can be used both for nearest neighbors queries and range queries. As far as we know, there are no other techniques that can be applied at the same time to range and nearest neighbors queries. Most existing techniques can only be

applied to nearest neighbors search, and in some cases to a single nearest neighbor search.

7.2 Proximity of metric ball regions

We have also proposed techniques for the efficient and effective estimation of the proximity of metric ball regions. The approximate similarity search approach proposed, that gives the best performance relies on this measurement.

The proximity of two ball regions was formalized, using a probabilistic approach, as the probability that a random object belongs to the intersection of the two ball regions. Given the computational difficulty of exactly evaluating the proximity, some heuristics were defined that relax the original problem in such a way that a very efficient and accurate estimation was guaranteed.

In accordance with our objectives, the methods proposed are flexible and only require that the distance functions are metric. The effectiveness of the methods is high and depends only on the overall distance distribution. The computational complexity of the techniques proposed is linearly proportional to the granularity of distance distribution samples, thus it is also applicable at run-time. The storage overhead for maintaining the distance distribution functions needed are low. Since our methods only assume the generic metric postulates, our results are automatically valid for the important class of vector spaces.

We have also shown that the precision of proximity evaluation is determinant for the accuracy of the approximate similarity search algorithm. In fact, in terms of efficiency and accuracy, the performance of proximity based similarity search algorithms using a trivial measurement of proximity is much lower than using our approaches.

7.3 Future directions

Many issues in approximate similarity search remain to be investigated. We have proposed a general framework that can be used in all applications where data are represented in a metric space and we have applied some objective measures to assess the accuracy and efficiency of the approaches proposed. However, different applications may have different requirements in terms of efficiency and accuracy. For instance, in image database systems, image similarity is highly subjective and the image similarity search application has low criticality, therefore missing some images in the results set is not a big problem. On the other hand, other applications such as for instance, stock quotes analysis or DNA sequences retrieval, have well defined ways to measure similarity between objects, and the accuracy of the result set might have a high impact. Approaches that take into account application considerations need to be designed using ad-hoc techniques. Their performance should be tested by using application-driven measures.

Incremental approximation techniques should also be investigated. The techniques that we have developed cannot incrementally use results obtained in previous queries. After a query has been processed, if the user wants to refine it (i.e. approximate less), a new query must be executed using the new approximation parameters, since it is currently not possible to exploits results obtained in the previous sessions. The possibility of incrementally reusing previous results would be very important and useful in highly interactive systems.

It would also be interesting to investigate how approximate similarity search algorithms can be used when processing complex similarity queries involving combinations of several similarity predicates. This is the case for instance of systems that perform

multi-feature searches, where similarity with respect to several features is evaluated at the same time in the same query.

We have also defined techniques for efficient and effective estimation of the proximity between ball regions and we have applied it as a basis for an approximate similarity search algorithm. However, other applications can benefit from this measurement. As an example, we can consider node allocation strategies, used to decide what is the best way to allocate nodes on the disk in order to have higher search performance, or partition strategies, that can be used to decide what is the best way to partition entries of a node when a split has to be performed. In both cases, the probability that two regions will be accessed together during query execution can help to take an optimal decision. This probability can be inferred by using the proximity measurement. Investigating other applications where the measurement of the proximity can be exploited and refining the techniques for computing it accordingly would also be an interesting line for future research direction.

Bibliography

[ABH97] P. Apers, H. Blanken, and M. Houtsama. *Multimedia Databases in Perspective*. Springer, 1997.

[AFS93] R. Agrawal, C. Faoutsos, and A. Swami. Efficient similarity search in sequence databases. In *FODO'93, Chicago, IL, October 1993*, pages 69–84, 1993.

[AG87] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algoritmica*, 2:315–336, 1987.

[All96] James Allan. Incremental relevance feedback for information filtering. In *SIGIR*, pages 270–278, 1996.

[AM95] Sunil Arya and David M. Mount. Approximate range searching. In *Symposium on Computational Geometry 1995*, pages 172–181, 1995.

[AMN+98] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, and Angela Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of ACM*, 45(6):891–923, 1998.

[BBK01] Christian Böhm, Stefan Berchtold, and Daniel Keim. Searching in high-dimensional spaces: Index structures for improving the performance

of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, September 2001.

[BC92] Nicholas J. Belkin and W. Bruce Croft. Information filtering and information retrieval: Two sides of the same coin. *Communication of the ACM*, 35(12):29–38, December 1992.

[Ben75] J. Bentley. Multidimensional binary search trees used for associative searching. *Communication of the ACM*, 18(9):509–517, 1975.

[Ben79] J. Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering*, 5(4):333–340, 1979.

[Bes95] S. N. Bespamyatnikh. An optimal algorithm for closest pair maintenance (extended abstract). In Joan Peckham, editor, *Proceedings of 11th ACM Symposium on Computational Geometry*, pages 152–161, 1995.

[BET95] Marshall W. Bern, David Eppstein, and Shang-Hua Teng. Parallel construction of quadtrees and quality triangulations. In *WADS '93, Proceedings of 4rd Algorithms and Data Structures, August 11-13 1997, Montral, Canada*, volume 709 of *LNCS*, pages 188–199. Springer Verlag, 1995.

[BFR78] R.L. Burden, J. Douglas Faires, and A.C. Reynolds. *Numerical Analysis*. Prindle, Weber & Schmidt, 1978.

[BGRS99] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In Catriel Beeri and Peter Buneman, editors, *Database Theory - ICDT '99, 7th International Conference, Jerusalem, Israel, January 10-12, 1999, Proceedings*, volume 1540 of *Lecture Notes in Computer Science*, pages 217–235. Springer, 1999.

[BKK96] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The x-tree: An index structure for high-dimensional data. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 28–39. Morgan Kaufmann, 1996.

[BKK97] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *PODS'97, Proceedings of 16th ACM Symposium on Principles of Database Systems, May 1997, Tucson, AZ*, pages 78–96. ACM, 1997.

[BKSS90] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM-SIGMOD International Conference on Management of Data, May 199a, Atlantic City, NJ*, pages 322–331. ACM, 1990.

[BM72] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.

[BÖ97] Tolga Bozkaya and Z. Meral Özsoyoglu. Distance-based indexing for high-dimensional metric spaces. In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 357–368. ACM Press, 1997.

[BÖ99] Tolga Bozkaya and Z. Meral Özsoyoglu. Indexing large metric spaces for similarity search querie. *TODS*, 24(3):361–404, 1999.

[Bri95] Sergey Brin. Near neighbor search in large metric spaces. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 574–584. Morgan Kaufmann, 1995.

[BWY80] J. Bentley, B. Weide, and A. Yao. Optimal expected-time algorithms for closest point problems. *ACM Transaction on Mathematical Software*, 6(4):563–580, 1980.

[Cas96] K.R. Castelman. *Digital Image Processing.* Prentice-Hall, Inc., 1996.

[CF80] J. M. Chang and K. S. Fu. A dynamic clustering technique for physical database design. In Peter P. Chen and R. Clay Sprowls, editors, *Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data, Santa Monica, California, May 14-16, 1980*, pages 188–199. ACM Press, 1980.

[Cha97] Timothy M. Chan. Approximate nearest neighbor queries revisited. In *Proceedings of the thirteenth annual symposium on Computational geometry, 1997 , Nice, France*, pages 352 – 358. ACM Press,New York, NY, USA, 1997.

[Chi94] Tzi-cker Chiueh. Content-based image indexing. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 582–593. Morgan Kaufmann, 1994.

[CK95] Paul B. Callahan and S. Rao Kosaraju. Algorithms for dynamic closest pair and n-body potential fields. In *SODA, Proceedings of 6th ACM-SIAM Symposium on Discrete Algorithms*, pages 263–272, 1995.

[CNBYM01] Edgar Chávez, Gonzalo Navarro, Ricardo Baeza-Yates, and José L. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–221, September 2001.

[Com79] Douglas Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[CP00] Paolo Ciaccia and Marco Patella. Pac nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In *Proceedings of the 16th International Conference on Data Engineering, 28 February - 3 March, 2000, San Diego, California, USA*, pages 244–255. IEEE Computer Society, 2000.

[CPZ97] Paolo Ciaccia, Marco Patella, and Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 426–435. Morgan Kaufmann, 1997.

[CPZ98a] Paolo Ciaccia, Marco Patella, and Pavel Zezula. A cost model for similarity queries in metric spaces. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington*, pages 59–68. ACM Press, 1998.

[CPZ98b] Paolo Ciaccia, Marco Patella, and Pavel Zezula. Processing complex similarity queries with distance-based access methods. In Hans-Jörg Schek, Fèlix Saltor, Isidro Ramos, and Gustovo Alonso, editors, *Advances in Database Technology - EDBT'98, 6th International Conference on Extending Database Technology, Valencia, Spain, March 23-27,*

1998, Proceedings, volume 1377 of *Lecture Notes in Computer Science*, pages 9–23. Springer, 1998.

[Dam64] F. Damerau. A technique for computer detection and correction of spelling errors. *Communication of the ACM*, 7(3):171–176, 1964.

[Fal96] C. Faloutsos. *Searching Multimedia Databases by Content.* Kluwer Academic Publishers, 1996.

[FB74] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[FK97] C. Faloutsos and I. Kamel. Beyond uniformity and independence: Analysis of r-trees using the concept of fractal dimension. In *Proceedings of the 13th ACM Symposium on Principles of Database Systems, El Paso, Texas, USA, May 4-6, 1997*, pages 609–617. ACM, 1997.

[FSA+95] Myron Flickner, Harpreet S. Sawhney, Jonathan Ashley, Qian Huang, Byron Dom, Monika Gorkani, Jim Hafner, Denis Lee, Dragutin Petkovic, David Steele, and Peter Yanker. Query by image and video content: The qbic system. *IEEE Computer*, 28(9):23–32, 1995.

[FTAA00] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi. Vector approximation based indexing for non-uniform high dimensional data sets. In *Proceedings of the 2000 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA, November 6-11, 2000*, pages 202–209. ACM, 2000.

[FTAA01] H. Ferhatosmanoglu, E. Tuncel, D. Agrawal, and A. El Abbadi. Approximate nearest neighbor searching in multimedia databases. In *Proceedings of the 17th International Conference on Data Engineering, April*

2-6, 2001, Heidelberg, Germany, pages 503–511. IEEE Computer Society, 2001.

[Fuk90] K. Fukunaga. *Introduction to Statistical Pattern Recognition.* Academic Press, 2nd edition, 1990.

[GG98] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[GPR97] Alberto Gobbi, D. Poppinger, and B. Rohde. Finding biological active compounds in large databases. In *ECSOC-1, Proceedings of First International Electronic Conference on Synthetic Organic Chemistry, September 1-30, 1997*, 1997.

[GSZ00] Claudio Gennaro, Pasquale Savino, and Pavel Zezula. A hashed schema for similarity search in metric spaces. In *Proceedings of First DELOS Network of Excellence Workshop, Information Seeking, Searching and Querying in Digital Libraries, Zurich, Switzerland, December 11-12*, 2000.

[GSZ01] Claudio Gennaro, Pasquale Savino, and Pavel Zezula. Similarity search in metric databases through hashing. In *Proceedings of MIR 2001 - 3rd Intl Workshop on Multimedia Information Retrieval October 5, 2001. Ottawa, Canada*, 2001. In conjunction with ACM Multimedia 2001.

[GSZ02] Claudio Gennaro, Pasquale Savino, and Pavel Zezula. D-index: Distance searching index for metric data sets. Technical report, IEI-CNR, 2002.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data, Boston, MA*, pages 47–57, 1984.

[Hal52] A. Hald. *Statistical Theory with Engineering Applications.* John Wiley & Sons, Inc., 1952.

[HB99] S. Hettich and S. D. Bay. The uci kdd archive. Irvine, CA: University of California, Department of Information and Computer Science, 1999. http://kdd.ics.uci.edu.

[HD80] P.A.V. Hall and G.R. Dowling. Approximate string matching. *ACM Computing Surveys*, 12(4):381–402, December 1980.

[HKR93] D.P. Huttenlocker, G.A. Klanderman, and W.J. Rucklidge. Comparing images using the hausdorff distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(9):850–863, September 1993.

[HNP95] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 562–573. Morgan Kaufmann, 1995.

[HPS71] P. Hoel, S. Port, and J. Stone. *Introduction to Probability Theory.* Houghton Mifflin Company, 1971.

[HS95] G.R. Hjaltason and H. Samet. Ranking in spatial databases. In *SSD'95, Portland, ME, August, 1995*, volume 951 of *LNCS*, pages 83–95. Springer Verlag, 1995.

[HSE+95] James L. Hafner, Harpreet S. Sawhney, William Equitz, Myron Flickner, and Wayne Niblack. Efficient color histogram indexing for quadratic form distance functions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(7):729–736, 1995.

[IM98] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 604–613, 1998.

[JMM95] H. V. Jagadish, Alberto O. Mendelzon, and Tova Milo. Similarity-based queries. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California*, pages 36–45. ACM Press, 1995.

[Kai85] T. Kailath. *Modern Signal Processing.* Springer Verlag, 1985.

[KF92] I. Kamel and C. Faloutsos. Parallel r-trees. In *Proc. of the ACM SIGMOD Conf., June, 1992*, pages 195–204, 1992.

[KF93] I. Kamel and C. Faloutsos. On packing r-trees. In *CIKM'93, Proceedings of the Second International conference on Information and Knowledge Management, Washington, DC, November, 1993*, pages 490–499, 1993.

[Kle97] Jon M. Kleinberg. Two algorithms for nearest-neighbor search in high dimensions. In *STOC'97, Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, 1997 , El Paso, Texas, United States*, pages 599–608, 1997.

[Knu98] D.E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Reading, Massachusetts: Addison-Wesley, 2nd edition, 1998.

[Koh84] T. Kohonen. *Self-Organization and Associative Memory.* Springer-Verlag, 1984.

[KOR99] Eyal Kushilevitz, Rafail Ostrovsky, and Yuval Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In

STOC'98, Proceedings of the thirtieth annual ACM symposium on Theory of computing, 1998Dallas, Texas, United States, pages 614 – 623, 1999.

[Lev65] V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problem of Information Transmission*, 1:8–17, 1965.

[MMLP97] Javed Mostafa, Snehasis Mukhopadhyay, W. Lam, and Mathew Palakal. A multilevel approach to intelligent information filtering: Model, system, and evaluation. *Transaction on Information Systems*, 15(4):368–399, October 1997.

[MS94] Masahiro Morita and Yoichi Shinoda. Information filtering based on user behaviour analysis and best match text retrieval. In W. Bruce Croft and C. J. van Rijsbergen, editors, *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval. Dublin, Ireland, 3-6 July 1994 (Special Issue of the SIGIR Forum)*, pages 272–281. ACM/Springer, 1994.

[Nav01] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[NBE+93] Wayne Niblack, Ron Barber, William Equitz, Myron Flickner, Eduardo H. Glasman, Dragutin Petkovic, Peter Yanker, Christos Faloutsos, and Gabriel Taubin. The qbic project: Querying images by content, using color, texture, and shape. In *Proceedings of Storage and Retrieval for Image and Video Databases (SPIE) 1993*, pages 173–187, 1993.

[NHS84] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *TODS*, 9(1):38–71, 1984.

[NW70] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amnio acid sequences of two protein. *Journal of Molecula Biology*, 48:444–453, 1970.

[OS89] A.V. Oppenheim and R.W. Schafer. *Discrete-Time Signal Processing.* Prentice-Hall, Inc., 1989.

[PAL99] S. Pramanik, S. Alexander, and J. Li. An efficient searching algorithm for approximate nearest neighbor queries in high dimensions. In *ICMCS 1999, IEEE International Conference on Multimedia Computing and Systems, June 7-11 , 1999, Florence, Italy*, volume 1. IEEE Computer Society, 1999.

[PL99] Sakti Pramanik and Jinhua Li. Ab-tree: Angle based index tree for approximate nearest neighbor search. Technical Report, Michigan State University, Department of Computer Science, February 1999. http://www.cse.msu.edu/ pramanik/research/papers/AB-tree.pdf.

[PM97] A. Papadopulos and Y. Manolopoulos. Performances of nearest-neighbor queries in r-trees. In *ICDT'97, Proceedings of the 6th International Conference on Database Theory, 1997, Delphi, Greece*, pages 394–408, 1997.

[PMD01] Dimitris Papadias, Nikos Mamoulis, and Vasilis Delis. Approximate spatio-temporal retrieval. *ACM Transactions on Information Systems (TOIS)*, 19(1):53 – 96, January 2001.

[RL85] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed r-trees. In Shamkant B. Navathe, editor, *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 28-31, 1985*, pages 17–31. ACM Press, 1985.

[Rob81] John T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In Y. Edmund Lien, editor, *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data, Ann Arbor, Michigan, April 29 - May 1, 1981*, pages 10–18. ACM Press, 1981.

[RV97] Paul Resnick and Hal R. Varian. Guest editors' introduction to the special issue on recommender systems. *Communication of the ACM*, 40(3), 1997.

[Sam88] Hanan Samet. Hierarchical representations of collections of small rectangles. *ACM Computing Surveys*, 20(4):271–309, 1988.

[Sam95] H. Samet. The quad-tree and related hierarchical structures. *ACM Computing Surveys*, 16(2):187–260, 1995.

[SK83] D. Sankoff and J. Kruskal. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, 1983.

[SM83] Gerald Salton and Michael J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Book Company, 1983.

[Smi97] John R. Smith. *Integrated Spatial and Feature Image Systems: Retrieval, Analysis, and compression.* PhD thesis, Graduate School of Arts and Sciences, Columbia University, 1997.

[SO95] M. Stricker and M. Orengo. Similarity of color images. In *Storage and Retrieval for Image and Video Databases III SPIE Proceedings 2420*, pages 381–392, 1995.

[SRF87] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In Peter M.

Stocker, William Kent, and Peter Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 507–518. Morgan Kaufmann, 1987.

[SW85] Hanan Samet and Robert E. Webber. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics*, 4(3):182–222, 1985.

[TS96] Y. Theodoridis and T. Sellis. A model for the prediction of r-tree performance. In *PODS'96, Proceedings of the 15th Symposium on Principles of Databses Systems, June 1996, Montreal, Canada*, pages 161–171, 1996.

[TTSF00] C. Traina, A.J. Traina, B. Seeger, and C. Faloutsos. Slim-trees: High performance metric trees minimizing overlap between nodes. In *EDBT 2000, Proceedings of the 7th EDBT International Conference, March 2000, Konstanz, Germany*, pages 51–65, 2000.

[Uhl91] J.K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, November 1991.

[Wat95] M. Waterman. *Introduction to Computational Biology.* Chapman and Hall, 1995.

[WFHC92] Steven P. Wartik, Edward A. Fox, Lenwood S. Heath, and Qi Fan Chen. Hashing algorithms. In William B. Frakes and Ricardo A. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*, pages 293–362. Prentice-Hall, 1992.

[WJ96] D.A. White and R. Jain. Similarity indexing with the ss-tree. In *Proceedings of the 12th International Conference on Data Engineering, New Orleans, USA*, pages 516–523, 1996.

[WSB98] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In Ashish Gupta, Oded Shmueli, and Jennifer Widom, editors, *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, pages 194–205. Morgan Kaufmann, 1998.

[YI99] A. Yoshitaka and T. Ichikawa. A survey on content-based retrieval for multimedia databases. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):81–93, 1999.

[Yia93] P.N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 311–321, 1993.

[Yia99] P.N. Yianilos. Excluded middle vantage point forests for nearest neighbor search. In *Sixth DIMACS Implementation Challenge: Nearest Neighbor Searches workshop*, January 1999.

[ZSAR98] Pavel Zezula, Pasquale Savino, Giuseppe Amato, and Fausto Rabitti. Approximate similarity retrieval with m-trees. *VLDB Journal*, 7(4):275–293, 1998.