DOTTORATO DI RICERCA

in

SCIENZE COMPUTAZIONALI ED INFORMATICHE

Ciclo XVII

Consorzio tra Università di Catania, Università di Napoli Federico II,
Seconda Università di Napoli, Università di Palermo, Università di Salerno

SEDE AMMINISTRATIVA: UNIVERSITÀ DI NAPOLI FEDERICO II

DIEGO REFORGIATO RECUPERO

DATA STRUCTURES AND ALGORITHMS FOR OPTIMIZATION AND SEARCHING PROBLEMS IN METRIC SPACES AND GRAPHS

TESI DI DOTTORATO DI RICERCA

To "Mary Kathryn"...

Table of Contents

Table of Contents ii

Abstract v

Acknowledgements vii

1 Introduction 1
1.1 Clustering 1
1.2 Graph Matching 5
1.3 Outline of the thesis 7

2 Basic Definitions 10

I Antipole Tree Indexing to Support Range Search and K-Nearest Neighbor Search in Metric Spaces 13

3 1-Median and Diameter (Furthest Pair) Approximated Algorithms 14
3.1 The 1-Median Algorithm 15
3.2 The 1-Median Algorithm over Graphs 17
3.3 Diameter (Furthest Pair) Algorithm 20
3.3.1 Running time analysis of Diameter (Furthest Pair) computation 21
3.3.2 A Diameter (Furthest Pair) computation on Euclidean spaces 25

4 The Antipole Clustering: A clustering scheme to index generic metric spaces 28
4.1 Introduction 29
4.2 Basic Definitions and Related Works 31
4.3 The Antipole Tree 34

4.3.1 The Antipole Tree data structure in general metric spaces . . 35
4.4 Range Search Algorithm 40
4.5 K-Nearest-Neighbor Algorithm 40
4.6 Experimental Analysis 43
4.6.1 Construction Time 45
4.6.2 Choosing the best cluster diameter 47
4.6.3 Range search analysis and comparisons 47
4.6.4 K-Nearest Neighbor comparisons 53
4.7 Approximate K-Nearest Neighbor search Via Antipole Tree 54
4.8 A comparison with linear scan 60
4.9 Secondary Memory management 62
4.9.1 Searching in Secondary memory via Antipole Tree 66
4.10 Dealing with Dynamic Updates 68

5 Applications 70
5.1 Protein Interaction Network 71
5.2 Graphs Clustering 72
5.2.1 Design 74
5.2.2 Algorithms 76
5.2.3 Complexity 78
5.2.4 Performance Studies 79
5.3 Texture synthesis 82
5.3.1 Texture synthesis via Antipole Tree Clustering 84
5.4 Image Colorization 89
5.4.1 Image Colorization using Antipole Tree Clustering 91

II Graph Searching Based on Indexing Techniques 96

6 GraphGrepVF: a new efficient method for exact and inexact graph matching 97
6.1 Introduction 98
6.2 GraphGrep 100
6.2.1 Building the sets of paths 101
6.2.2 Filtering the database 104
6.2.3 Matching 105
6.2.4 Data storage with BerkeleyDB 106
6.2.5 Complexity Analysis 107
6.3 VF 108

6.3.1 The Algorithm 108
6.4 GraphGrepVF for exact graph matching 111
6.5 Performance Analysis and Results 113
6.6 Graph LInear DEscription language (GLIDE) 117
6.6.1 Syntaxis and semantic of GLIDE 118
6.7 Extension of GraphGrepVF for inexact graph matching 120

7 Conclusions 126

Bibliography 129

Abstract

There has been increasing interest in building search/index structures to perform similarity search over high-dimensional data, e.g., image databases, document collections, time-series databases, and genome databases. A similarity search problem involves a collection of objects (e.g., documents, images) which are characterized by a collection of relevant features and represented as points in a high-dimensional attribute space.

The first part of this thesis will present a new hierarchical clustering algorithm called **Antipole Clustering**. The algorithm partitions the set of data objects in clusters such that each one has diameter approximately less than a given value σ. The algorithm returns a tree structure called *Antipole Tree* in which the leaves are the final clusters.

This thesis will prove that the Antipole Tree is a suitable data structure to index a metric space in order to solve efficiently problems such as *Range Search* and *Nearest Neighbor Search*. These are core problems in pattern recognition, where a database D of objects in a metric space M and a query object q in M are given, and one wants to find those objects in D that are similar to q. A range query describes a region in the space and asks for all points or the number of points in that region. A k-nearest-neighbor query asks for the k nearest (most similar) objects to the query.

Several applications of the Antipole Clustering will be presented and analyzed:

- texture synthesis;
- image colorization;
- clustering of labeled graphs according to their structure;

- discovering the genes correlations in a protein network environment.

The second part of this thesis will present, **GraphGrepVF**, an application-independent method for querying a database of graphs in order to find all the occurrences of a given subgraph. Many applications in industry, science and engineering share this problem, and increasing the size of the database requires efficient structure searching algorithms. New-generation database systems, dealing with Web, XML, network directories and structured documents, molecular database etc., often model data as graphs. In this context, several algorithms for graph querying have been developed. In this thesis we present the GraphGrepVF algorithm by which structured objects in the form of a graph can be retrieved efficiently. The method is an extension of two very well known techniques for graph matching: GraphGrep and VF. Our method enumerates and hashes node-paths of the graphs data set in order to create two fingerprints of the database. When a query has to be processed, we build its fingerprints, and we use them to prune the search database. Using the first fingerprint we prune database graphs, while using the second one we prune edges inside graphs. In this way we drastically reduce the search space. Moreover, the graphs are smaller than the originals. Finally, searching this small residual set for subgraph matching will be performed by VF. GraphGrepVF has been compared with the best known software, (GraphGrep, VF, Daylight and Frowns) and in every test GraphGrepVF has shown better performance. GraphGrepVF has been observed to be competitive also when inexact matching is considered.

Acknowledgements

First I would like to thank and to express my deep gratitude to my advisor Professor Alfredo Ferro, who has trusted in my capacities, given me suggestions and advice, and influenced me with his incredible enthusiasm.

I also wish to thank Professor Dennis Shasha whom I worked with for six months during my period abroad at New York University. He supported me tremendously, giving me interesting new ideas for my projects and helping me both as a friend and a mentor.

Thank you also to Professor Antonio Maugeri who has helped me and believed in my research skills.

Thank you very much to my family for having supported me in difficult moments of my life.

Results presented in this thesis are in collaboration with Professor Alfredo Ferro, Professor Domenico Cantone, Dr. Alfredo Pulvirenti, Dr. Rosalba Giugno, Dr. Gianluca Cincotti, Dr. Sebastiano Battiato, Dr. Simone Faro and Dr. Gianpiero Di Blasi from the University of Catania, and also with Professor Dennis Shasha, Dr. Rodrigo Gutierrez and Chih-Yi Hsu from New York University.

It has been a great pleasure for me to have worked with all of them.

Chapter 1

Introduction

Data mining and knowledge discovery in databases play an important role in the way people interact with databases, especially scientific databases where analysis and exploration operations are essential. Often databases are defined over a metric space, where the objects have a global distance function (the metric). There has been enormous interest in finding efficient solutions of problems, such as clustering, nearest neighbor and range search, 1-median, and furthest pair, for data defined over a metric space.

In recent years the importance of large datasets manipulation has grown. The way in which the data are presented has been influenced by the progress of science and technology. A new problem in database research is how to deal with objects which need to be represented as graphs that can be manipulated more efficiently. For this reason several researchers have begun to formulate models for next generation databases able to represent novel types of data and provide manipulation capabilities while efficiently supporting standard searching operations.

1.1 Clustering

Clustering is a very well known problem in computer science. Given points in some space, often a high-dimensional space, the problem consists of grouping the points

into a small number of clusters, each one containing points that are "similar" in some sense. The problem can be defined in one of the following forms:

1. given a set S and an integer $k > 1$, find a partition of points in k classes such that a certain function is minimized. Examples of such functions are: (a) the sum of the distances from the cluster objects to their cluster representative (k-median), (b) the maximum distance between the objects in a cluster (k-center), (c) the sum of the distances between points in the same cluster (k-clustering);

2. given a set S and a real number σ, partition S in the minimum number of clusters such that the maximum distance between two points in each cluster is at most σ (bounded diameter clustering).

Clustering was first applied many years ago, during a cholera outbreak in London, when a physician plotted the location of cholera cases on a map, getting a plot that looked like Fig. 1.1. Properly visualized, the data indicated that cases clustered

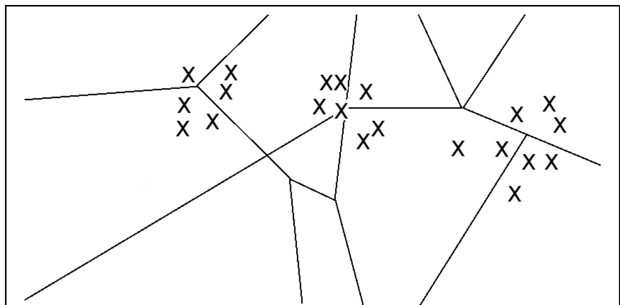


Figure 1.1: Clusters of cholera cases indicated the locations of polluted wells.

around certain intersections, where the water wells were polluted. Thus clustering, not only exposing the cause of cholera, but also indicated how to solve that problem.

At a high level, clustering algorithms can be divided into two broad classes:

- centroid approaches. The algorithms guess the centroids or central points in each cluster and assign points to the cluster according to the nearest centroid;

- hierarchical approaches. Assuming that each point is a cluster by itself, the algorithms repeatedly merge nearby clusters, using some measure of how close two clusters are (e.g., distance between their centroids) or how good a cluster the resulting group would be (e.g., the average distance of points in the cluster from the resulting centroid).

The problem of Clustering belongs to a class of fundamental proximity problems which involves the distance between objects. That class also includes :

- 1-median problem. Find the point which has the minimum average distance from the others points in the data set;

- furthest pair problem. Find the pair of objects in the set having maximum distance;

- range search problem. Given a set of objects in a metric space and a query region, find all the points which belong to the region;

- nearest neighbor search. Given a set S of objects in a metric space and a query object q, return the object of S which is closest to q.

These problems appear in many computer science areas. They have been investigated for long time, and some efficient solutions have been found. Unfortunately many applications require a higher space dimension (e.g. when the dimensionality of the space is $d = \Omega(\log n)$); in this case, algorithms for low dimensions become inefficient. Many researchers conjecture that no efficient solutions exist for these problems when the dimension is high enough [99]; this difficulty is called *the curse of dimensionality* or, for the more general case of the metric spaces, *intrinsic dimensionality.*

Two of the proximity problems just described, nearest neighbor and range search, are very frequent in pattern recognition. It is simple to prove the linearity of the solutions for both problems on the size of the input data set, but of course the goal is to solve them faster.

Nearest Neighbor search has been a core problem in pattern recognition since the early sixties, when, in one of the first applications presented [58, 49], it was proven that half the classification information in an infinite sample set is contained in the nearest neighbor. If Voronoi Diagram [108] is used to index the space, then the nearest neighbor problem has an optimal solution; however, in higher dimensional spaces it becomes harder, and classical indexing methods, used for low dimensional spaces, appear to be inadeguate.
One important application of nearest neighbor problem is in multimedia databases. Objects (i.e. images, documents) are represented as *feature vectors*, and the search for the nearest object is performed by searching for the nearest feature vector. A typical approach to solving nearest neighbor is to map the object space in a vector space and perform the search by using the Euclidean distance, taking into account that certain projected distances are dominated by those in the original space. This immersion often represents the most difficult and sensitive step because it affects the performance of the search in terms of running time and accuracy of the results. On the other hand, for several kinds of databases, such as genetic sequences and graphs, other techniques, which do not make use of immersion, are possible. For example, the editing distance is commonly used for searching databases of bio sequences [122].

Many applications involve huge data sets which have high dimensionality or in the more general case of metric spaces, high intrinsic dimensionality. It is crucial to solve such problems by designing algorithms that are both fast and scalable to the size of the database.

In the first part of this thesis, we show a new data structure for clustering which is able to perform range and nearest neighbor searches efficiently and quickly. In particular this algorithm uses the diameter and the furthest pair approximate computation to speed up the entire process. The 1-median algorithm used return an approximate solution in linear time which introduces a small error, as compared to the quadratic time of an exact algorithm.

1.2 Graph Matching

Many applications need to represent data as graphs. For example in a biochemical database, proteins are represented as labeled graphs: the vertices represent the atoms and the edges represent the links between the atoms Fig. 1.2 (image taken from [68]).

(a) (b)

Figure 1.2: (a) chemistry compound. (b) query containing wildcards.

Usually proteins are classified on the basis of common structures. One application of such classification is the prediction of the functionality of a new protein fragment (discovered or synthesized). The user queries the database and gets all the proteins containing that particular query fragment. Queries may also contain wildcards to perform inexact matching according to what the users want (the substitution of a vertex or a path in the data). Solving this problem requires tools to compare different graphs, recognize different part of graphs, and retrieve and classify them. In the

context of non-structured data (such as strings), string matching algorithms match query words against documents extremely efficiently. Just as string matching matches words against documents, graph matching matches query graphs against underlying data graphs. Many efforts have been made to generalize string matching to graph matching, but such generalization is not natural: string matching has polynomial complexity on the database size. Graph matching has exponential complexity, and thus belongs in a entirely different class of problems.

Three similar NP-complete problems about graph matching exists:

1. *graphs isomorphism.* Graphs G and H are isomorphic if there is a one-to-one correspondence between the nodes of G and the nodes of H such that an edge of H exists if and only if the corresponding nodes of G also are connected by an edge of G;

2. *subgraphs isomorphism.* Graphs G and H_1, subgraph of H, are isomorphic if there is a one-to-one correspondence between the nodes of G and the nodes of H_1 such that an edge of H_1 exists if and only if the corresponding nodes of G also are connected by an edge of G;

3. *graphs monomorphism.* Graphs G and H_1, subgraph of H, are monomorphic if there is a correspondence between the nodes of G and the nodes of H_1 such that an edge of G exists if the corresponding nodes of H_1 are also connected by an edge of H_1.

If we have a database of graphs instead of only one data graph, of course methods to prune and speed up the matching process are used. In graphs monomorphism, the problem we studied, a simple enumeration algorithm to find all the occurrences of a query graph in a data graph generates all possible maps between the vertices of the two graphs and checks whether the generated map is a match. Given that the

complexity of such an algorithm is exponential, there have been many research efforts to reduce the cost.

First of all, researchers have been studied matching algorithms which tried to take advantage of the particular kinds of graphs. (planar graphs, bounded valence graphs and associate graphs). Another direction taken was to study tricks to reduce the number of generate maps. Approximate methods with polynomial complexity but with no guarantee to find correct solutions have been studied also.

In the second part of this thesis, a new algorithm for graph matching, called GraphGrepVF, is presented. The new algorithm creates two data set fingerprints; once the user gives the query, the algorithm tries to prune as many graphs in the database as possible using the first fingerprint and as many paths as possible inside graphs. The matching step is performed over the remaining reduced graphs. GraphGrepVF, moreover, deals with both exact and inexact matching using the GLIDE query language [68, 69], which allows to the user to define his own query and to define where an approximation must be.

1.3 Outline of the thesis

This thesis is organized into two main parts (see Fig. 1.3). In the first part, we

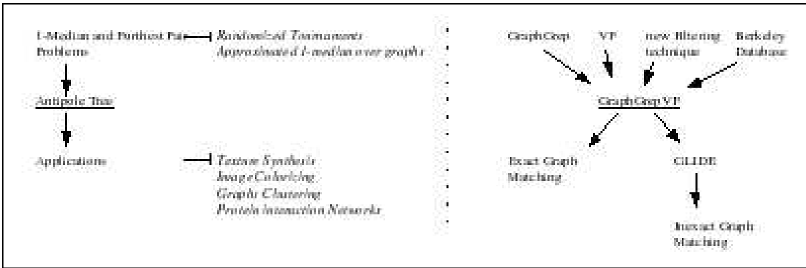


Figure 1.3: Structure of the thesis.

introduce several basic concepts and solutions for problems that can be modelled

by metric spaces. This section is divided in the following chapters: *1-median and furthest pair algorithms*, *the Antipole clustering*, *applications of Antipole clustering*. Each chapter is composed as follows:

- **1-median and diameter (furthest pair) approximated algorithms.** The concept of "randomized tournament" is introduced. Through using this concept we describe a general technique to solve optimization problems in metric spaces. This chapter contains a core technique to solve the approximate 1-median problem and the approximate diameter problem. In particular, this chapter introduces an approximation scheme for the diameter computation in the Euclidean space. Moreover, we present a new, very fast approximate algorithm to solve the 1-median problem in a weighted graph where the metric is the shortest path distance.

- **The Antipole clustering: a clustering scheme to index generic metric spaces.** This chapter presents the Antipole clustering method to build a data structure called Antipole Tree. The Antipole Tree is shown to be a suitable data structure to index a generic metric space. By using Antipole Tree, two problems are efficiently solved: range search problem and k-nearest neighbor problem. This method is also successfully compared with four state-of-the-art data structures: M-Tree, MVP-Tree, List of Clusters, TSVQ.

- **Applications.** Four important applications of the Antipole Tree clustering are discussed. The first one shows how the clustering can be used to support the discover, and study of gene correlations in a protein network environment. The second application deals with an interesting method to cluster labeled graphs by using Antipole data structure. We address concepts about documents and text clustering, in particular, how to map a document on a Euclidean space. The last two applications show how the Antipole Tree can be used to speed up

the synthesis of texture and the image colorization in computer graphics. The nearest neighbor search is used as a basic step to synthesize and to colorize the images.

The second part of the thesis presents GraphGrepVF, a new and efficient method of graph matching that came out of the study of two well known algorithms, GraphGrep [68, 71, 117] and VF [46]. The chapter is entitled *GraphGrepVF: a new efficient method for exact and inexact graph matching.* First of all, the chapter explains how GraphGrep [68, 71, 117] and VF [46] work. Secondly, we show that these two methods can be combined with a new filtering technique to create the superior GraphGrepVF. GLIDE (Graph LInear DEscription) [69], a graph linear query language, its semantics, syntax, and reasons for use are discussed and explained. Finally we explain how to deal with inexact matching.

Chapter 2

Basic Definitions

In this chapter, we introduce the basic definitions of all the problems discussed in the two parts of this thesis.

Definition 2.0.1. Metric Space. A metric space is a pair (X, d) such that X denotes the universe of valid objects and d is a function defined as follows:

$$d : X \times X \longrightarrow \mathbb{R}$$

with the following properties

1. *positiveness.* $\forall\, x, y \in X\; d(x, y) \geq 0$;
2. *symmetry.* $\forall\, x, y \in X\; d(x, y) = d(y, x)$;
3. *reflexivity.* $\forall\, x \in X\; d(x, x) = 0$;
4. *triangle inequality.* $\forall\, x, y, z \in X\; d(x, y) \leq d(x, z) + d(z, y)$;
5. *identity of indiscernibles.* $\forall\, x, y \in X$ if $d(x, y) = 0$ then $x = y$.

Definition 2.0.2. Range query. Given a set of objects S, a query object q, and a threshold t, a range query retrieves all elements in S that are within distance t from q. That is $\{x \in S \,|\, d(x, q) \leq t\}$.

Definition 2.0.3. Nearest neighbor query. Given a set of objects S, a query object q, a nearest neighbor query retrieves the closest elements to q in S. That is $\{x \in S \,|\, \forall\, y \in S\;\; d(x, q) \leq d(y, q)\}$.

Definition 2.0.4. k-Nearest neighbor query. Given a set of objects S, a query object q, and an integer value $k > 0$, a k-nearest neighbor query retrieves the k closest elements to q in S. That is $\{x_1, \ldots x_k \in S \,|\, \forall\, y \in S\,/\{x_1, x_2, \ldots, x_k\}\; d(x_i, q) \leq d(y, q)\; i = 1, \ldots, k\}$.

Definition 2.0.5. k-Medians problem. Given a set of objects S, a finite subset M of S, and an integer value $k > 0$, the k-medians problem asks k points $c_1, c_2, \ldots, c_k$ in M which minimize $\sum_{s \in M} \min_{i=1,\ldots,k} d(s, c_i)$.

Definition 2.0.6. Furthest Pair problem. Given a set of objects S, a finite subset M of S, the furthest pair problem asks a pair of points A and B such that $d(A, B) \geq d(x, y)\, \forall\, x, y \in M$.

Definition 2.0.7. Graph. A graph G is a pair of finite set (V, E). $V = \{v_0, v_1, \ldots, v_k\}$ is the set of vertices in G. $E = \{(v_i, v_j) \,|\, i, j \in \{1, \ldots, k\}\}$ is the set of edges in G.

Definition 2.0.8. Undirected graph. A graph $G = (V, E)$ is called undirected if (v_i, v_j) is the same of (v_j, v_i).

Definition 2.0.9. Directed graph. A graph $G = (V, E)$ is called directed if the edges have a direction. The (v_i, v_j) is an *outgoing* edge for v_i and an *incoming* edge for v_j.

Definition 2.0.10. Labeled graph. Let L a set of labels, $\mu : V \rightarrow L$ be a node labeling function and $\nu : E \rightarrow L$ be an edge labeling function. Then $G = (V, E, \mu, \nu)$ is a labeled graph. If L is a set of integers, real numbers or any type of quantity, an edge labeled graph is called weighted graph.

Definition 2.0.11. Dense and sparse graph. A graph is called dense if the number of edges $|E|$ is $O(|V|^2)$, otherwise is called sparse.

Definition 2.0.12. Complete graph. A graph is called complete if it is undirected and exists an edge between every pair of vertices. That is $|E| = |V|^2$.

Definition 2.0.13. Subgraph. A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

Definition 2.0.14. Path in a graph. A path of length k from a vertex u to a vertex u' in a graph $G = (V, E)$ is a sequence $(v_0, v_1, \ldots, v_k)$ of vertices such that $u = v_0$, $u' = v_k$ and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$.

Definition 2.0.15. Graphs Isomorphism. Graphs G and H are isomorphic if there is a one-to-one correspondence between the nodes of G and the nodes of H such that an edge of H exists if and only if the corresponding nodes of G also are connected by an edge of G.

Definition 2.0.16. Subgraphs Isomorphism. Graphs G and H_1, subgraph of H, are isomorphic if there is a one-to-one correspondence between the nodes of G and the nodes of H_1 such that an edge of H_1 exists if and only if the corresponding nodes of G also are connected by an edge of G.

Definition 2.0.17. Graphs Monomorphism. Graphs G and H_1, subgraph di H, are monomorphic if there is a correspondence between the nodes of G and the nodes of H_1 such that and edge of G exists if the corresponding nodes of H_1 also are connected by an edge of H_1.

Part I

Antipole Tree Indexing to Support Range Search and K-Nearest Neighbor Search in Metric Spaces

Chapter 3

1-Median and Diameter (Furthest Pair) Approximated Algorithms

Algorithms based on randomized tournaments for metric spaces have been widely studied and analyzed in [110, 9]. This chapter is based on [110].

Let (M, d) be a metric space with distance function $d : (M \times M) \longmapsto \mathbb{R}$ and let S be a finite subset of M. Given $k \in \mathbb{N}$, the *k-median problem* for S is the problem to find k points $c_1, c_2, \ldots, c_k$ in S which minimize $\sum_{s \in S} \min_{i=1,\ldots,k} d(s, c_i)$.

This computational geometry problem is $\mathcal{NP}$-complete (see [82]) and many approximation algorithms have been developed to solve it (see [32, 86]).

The *1-median problem*, for $k = 1$, is therefore the problem of selecting an element c in S which minimizes the sum $\sum_{s \in S} d(s, c)$ of all the distances from c to any other point of the input set S. It is sometimes referred to as "centroid" selection and its main application fields deal with both Euclidean and non-Euclidean metric spaces.

The *diameter (furthest pair) problem* looks for a pair of points, A and B in S such that $d(A, B) \geq d(x, y)\ \forall\, x, y\ \in S$. This problems seems to be easier than the 1-median computation. In metric space it is not true. Also, differently from the previous problem, this cannot be approximated with a linear randomized algorithm. As observed in [86], we can construct a metric space where all distances among objects are set to 1 except for one (randomly chosen) which is set to 2. In this case any

algorithm that tries to give an approximation factor greater than 1/2 must examine all pairs, so a randomized algorithm will not necessarily find that pair. Nevertheless, we expect a good outcome in nearly all cases.

Algorithmic solutions of computational geometry problems on metric spaces are very much affected by the high computational cost of distance calculations among objects of the space. The exact 1-median problem, for instance, has an obvious quadratic solution in the size of the input set but even a quadratic algorithm can be prohibitive with an extremely large size of the input. Moreover, in many applications the real solutions are not required, in the sense that it is sufficient to compute solutions close enough to the optimal one. The algorithms we propose here address such situations, allowing an efficient computation of an approximate solution to both problems.

3.1 The 1-Median Algorithm

Our randomized algorithm is based on a tournament played among the elements of the input set S. At each round, the elements which passed the preceding turn are randomly partitioned into subsets, say $X_1, \ldots, X_k$. Then, each subset X_i is locally processed through a procedure which computes its exact 1-median x_i. The elements $x_1, \ldots, x_k$ move to the next round. The tournament terminates when we are left with a single element $\overline{x}$, the final winner. The winner approximates the exact 1-median in S.

A possible implementation of the above general method consists of partitioning the elements at each round in subsets having the same size t, with the possible exception of one subset whose size must lie between t and $2t-1$. In addition, we can assume that the iteration stops when the number of elements falls below a given *threshold*. Finally the quadratic "brute force" computation returns the exact 1-median of the residual set. This is summarized in Fig. 3.1, where the local optimization procedure WINNER (X) returns the exact 1-median in X.

The approximate 1-Median Selection algorithm

```
WINNER (X)
 1  for each x ∈ X do
 2        σ_x ← Σ_{y∈X} d(x, y);
 3  Let m ∈ X be an element such that σ_m = min_{x∈X}(σ_x);
 4  return m;
    END WINNER.


APPROX_1_MEDIAN (S)
 1  while |S| > Threshold do
 2        W ← ∅;
 3        while |S| ≥ 2t do
 4              Choose randomly a subset T ⊆ S, with |T| = t;
 5              S ← S \ T;
 6              W ← W ∪ {WINNER (T)};
 7        end while;
 8        S ← W ∪ {WINNER (S)};
 9  end while;
10  return WINNER (S);
    END APPROX_1_MEDIAN.
```

Figure 3.1: Pseudo-code of the approximate 1-Median Selection algorithm.

Notice that, each random partitioning phase can be simplified introducing efficient pseudo-randomization methods (see [9]). It can be easily seen, in the case of unidimensional Euclidean spaces, the resulting approximate algorithm reduces exactly to the one given in [9].

A running time analysis, (see [29] for details), shows that above procedure takes time $\frac{t}{2}n + o(n)$ in the worst case.

3.2 The 1-Median Algorithm over Graphs

In this section we describe a new approximated algorithm to solve 1-median problem for graphs.

Over graphs, taking into account the shortest path distance, applying the above approximate 1-Median Selection Algorithm is not a good idea: as described in [29], given an input set of cardinality $n = t^r$, with $r \in \mathbb{N}$, the 1-Median Selection Algorithm performs exactly $\frac{t}{2}(n-1)$ distance computations. So we would have to call Dijkstra $\frac{t}{2}(n-1)$ times where n is the number of nodes $|V|$ getting a complexity of $\mathcal{O}(|V|^2 \log V)$ using heaps. A very simple method to compute an exact solution is to generate the map of all possible pairs of distances by running the AllPairShortestPath algorithm; the 1-median node v is the node such that the sum of the distances between the node v and each other node is miminum. The complexity of the exact method is $\mathcal{O}(|V|^2 \log V)$ using heaps. This is the same complexity as would be gotten by using the 1-Median Selection algorithm so it does not make sense to apply the 1-Median Selection Algorithm.

The idea of this new method is to give importance to the nodes through which the maximum number of shortest paths pass during the Dijkstra visit. For this the algortihm is called Walking Path algorithm. The algorithm deals with graphs either weighted or not and either directed or undirected.

Fig. 3.2 shows the two main routines of the algorithm. The main procedure, WALKING_PATH, starts finding the node u with greater degree. In the next lines the algortihm initializes *S_visited* to 0 and *min* to ∞; *S_visited* will contain all the nodes visited so far while *min* will contain the sum of the distances of the temporary optimal solution.

Once INNER_WALKING is executed, the number of Dijkstra shortest paths passing for the nodes adjacent to u is computed. The sum of distances from u to

The approximate 1-Median Walking Path algorithm

```
INNER_WALKING (G, u, min, S_visited, v)
 1  S_visited ← S_visited ∪ {u};
 2  compute the number of Dijkstra shortest paths passing for the nodes adjacents to u;
 3  sum_u ← sum of distances from u to each other node;
 4  if sum_u < min then
 5      v ← u;
 6      min ← sum_u;
 7      t ← adjacent node to u with maximum number of Dijkstra shortest path;
 8      if t ∉ S_visited then
 9          INNER_WALKING(G, t, min, S_visited, v);
10      end if;
11      strat_max ← greater distance of node q from u such that q is not adjacent to u;
12      strat_min ← smaller distance of node z from u such that z is not adjacent to u;
13      l ← (strat_max − strat_min) / 10;
14      let v1 the node not adjacent to u such that the distance between u and v1 is
        greater or equal than strat_min + l and less or equal than strat_min + 2l and
        v1 ∉ S_visited;
15      if v1 ≠ NULL then
16          INNER_WALKING(G, v1, min, S_visited, v);
17      end if;
18      let v2 the node not adjacent to u such that the distance between u and v2 is
        greater or equal than strat_min + 2l and less or equal than strat_min + 3l and
        v2 ∉ S_visited;
19      if v2 ≠ NULL then
20          INNER_WALKING(G, v2, min, S_visited, v);
21      end if;
22      let v3 the node not adjacent to u such that the distance between u and v3 is
        greater or equal than strat_min + 3l and less or equal than strat_min + 4l and
        v3 ∉ S_visited;
23      if v3 ≠ NULL then
24          INNER_WALKING(G, v3, min, S_visited, v);
25      end if;
26  end if;
    END INNER_WALKING.


WALKING_PATH (G, min, S_visited)
 1  find the node u with greater degree;
 2  S_visited ← {∅};
 3  min ← ∞;
 4  INNER_WALKING(G, u, min, S_visited, v);
 5  return v;
    END WALKING_PATH.
```

Figure 3.2: Pseudo-code of the approximate 1-Median Walking Path algorithm.

every other node is also determined, and the algorithm continues recursively only if this sum is less than the temporary optimal min. Once v and min are updated, let t be the node adjacent to u with the greatest number of Dijkstra shortest paths passing through it. Then the algorithm proceeds recursively with the node t if t has not been visited yet.

So far, (line 10) we have shown the basic algorithm and the basic idea of this method; however, sometimes, the node t with the greatest number of Dijkstra shortest path does not arrive to a good solution. In this case, the reason for the failure is that centroids may exist that are local to a region of the graph and are not good for the general graph. The next lines of this procedure provide a method to overcome such a local minima.

The idea to overcome such a local minima is to build a stratification of the nodes according to their distance from the temporary solution u. Let $strat_max$ and $strat_min$ be respectively the greater and the smaller distance from u to a node not adjacent to u. With l we divide the space of all distances of other nodes from u in ten strips. Empirically, taking a node in first strip ($[strat_min + l, strat_min + 2l]$), another in the second strip ($[strat_min + 2l, strat_min + 3l]$) and another in the third strip ($[strat_min + 3l, strat_min + 4l]$), it is highly probable that the path visited by the algorithm will come out from the local minima: the error is kept low, and also the number of nodes to visit does not grow too much.

Such an approximated algorithm is very fast and also performs very well. The experiments performed have been executed on 100 connected sparse random graphs, 100 connected dense random graphs and 100 grid connected graphs; the number nodes has been fixed for all the experiments to 1000. Table 3.1 shows comparisons between the exact method and our approximated method. The second and the third column show the average time, respectevely, for the exact method and for our approximated method; the forth and fifth column show respectevely the average number of nodes

our approximated method visits and the average number of nodes the exact method visits (we remind that exact method visits 1000 nodes); the sixth column shows the average position of the results returned by our method with respect to the exact centroid; finally, last column shows the error.

For every graph, we have observed that our method is always faster than the exact method and the error is kept low.

Graph Typology	Avg Exact	Avg Walking	Avg Exact Size	Avg Walking Size	Avg Pos	Error
sparse random	10.20 secs	0.85 secs	1000	24.16	2.66	0.006
dense random	32.86 secs	2.65 secs	1000	27.76	3.87	0.008
grid	1.4 secs	1.2 secs	1000	105.75	1.03	0.00004

Table 3.1: Average results on 100 sparse random graphs, 100 dense random graphs and 100 grid graphs.

3.3 Diameter (Furthest Pair) Algorithm

Here we introduce a randomized algorithm inspired by the one proposed for the 1-median computation [29] and reviewed in the preceding section. In this case, each subset X_i is locally processed by a procedure LOCAL_WINNER which computes its exact 1-median x_i and then returns the set $\overline{X}_i$, obtained by removing the element x_i from X_i. The elements in $\overline{X}_1 \cup \overline{X}_2 \ldots \cup \overline{X}_k$ are used in the subsequent step. The tournament terminates when we are left with a single set, $\overline{X}$, from which we extract the final winners A, B, as the furthest points in $\overline{X}$. The pair A, B is called the *Antipole pair* and their distance represents the approximate diameter of the set S.

The pseudo-code of the Antipole algorithm, similar to that of the 1-Median algorithm, is given in Fig. 3.3.

A faster (but less accurate) variant of APPROX_ANTIPOLE(S) can be used. Such variant, called FAST_APPROX_ANTIPOLE, consists of taking $\overline{X}_i$ as the

farthest pair of X_i. Its pseudocode can therefore be obtained simply by replacing in APPROX_ ANTIPOLE each call to LOCAL_WINNER by a call to FIND_ANTIPOLE. In the next section we will prove that both variants have a linear running time in the number of elements. We will also show that FAST_APPROX_ANTIPOLE is also linear in the tournament size τ, whereas APPROX_ANTIPOLE is quadratic with respect to τ.

3.3.1 Running time analysis of Diameter (Furthest Pair) computation

Two fundamental parameters present in the algorithm reported in Fig. 3.3 (also reported in Fig 3.1), namely the *splitting factor* τ (also referred to as the *tournament size*) and the parameter *threshold*, need to be tuned.

The splitting factor τ is used to set the size of each subset X processed by procedure LOCAL_WINNER, with the only exception of one subset for each round of the tournament (whose size is at most $(2\tau - 1)$), and the argument of the last call to FIND_ANTIPOLE (whose size is at most equal to *threshold*). It is clear that the larger values of τ are, the better the output quality is and the higher the computational costs are. In many cases a satisfying output quality can be obtained even with small values for τ.

A good trade-off between output quality and computational cost is obtained by choosing as value for τ one unit more than the dimension that characterizes the investigated metric space [35]. This suggestion lies on intuitive grounds developed in the case of a Euclidean metric space $\mathbb{R}^m$ and is largely confirmed by the experiments reported in [29]. The parameter *threshold* controls the termination of the tournament. Again, larger values for *threshold* ensure better output quality, though at increasing cost. Observe that the value $(\tau^2 - 1)$ for *threshold* forces the property that the last set of elements, where the final winner is selected, must contain at least τ elements,

The approximate Antipole Selection algorithm

```
LOCAL_WINNER (T)
 1  return T \ WINNER (T);
    END LOCAL_WINNER.


FIND_ANTIPOLE (T)
 1  return P1, P2 ∈ T such that dist(P1, P2) ≥ dist(x, y) ∀ x, y ∈ T;
    END FIND_ANTIPOLE.


APPROX_ANTIPOLE (S)
 1  while |S| > threshold do
 2          W ← ∅;
 3          while S ≥ 2 * τ do
 4                  Choose randomly a subset T ⊆ S : |T| = τ;
 5                  S ← S \ T;
 6                  W ← W ∪ {LOCAL_WINNER(T)};
 7          end while
 8          S ← W ∪ {LOCAL_WINNER(S)};
 9  end while
10  return FIND_ANTIPOLE(S);
    END APPROX_ANTIPOLE.
```

Figure 3.3: Pseudo-code of the Antipole Selection algorithm.

provided that $|S| \geq \tau$. Moreover, in order to ensure a linear computational complexity of the algorithm, the *threshold* value need to be $\mathcal{O}(\sqrt{|S|})$. Consequently, a good choice is *threshold* $= \min\left\{\tau^2 - 1, \sqrt{|S|}\right\}$.

The algorithm APPROX_ANTIPOLE given in Fig. 3.3 is characterized by its simplicity and hence it is expected to be very efficient from the computational point of view, at least in the case in which the parameters τ and *threshold* are taken small enough. In fact, we will show below that our algorithm has a worst-case complexity of $\frac{\tau(\tau-1)}{2}n + o(n)$ in the input size n, provided that *threshold* is $o(\sqrt{n})$.

Plainly, the complexity of the algorithm APPROX_ANTIPOLE is dominated by the number of distances computed by it within calls to procedure LOCAL_WINNER. We shall estimate below such a number.

Let $W(n, \tau, \vartheta)$ be the number of calls to procedure LOCAL_WINNER made within the **while**-loops by APPROX_ANTIPOLE, with an input of size n and using parameters $\tau \geq 3$ and threshold $\vartheta \geq 1$. Plainly, $W(n, \tau, \vartheta) \leq W(n, \tau, 1)$, for any $\vartheta \geq 1$, thus it will suffice to find an upper bound for $W(n, \tau, 1)$. For notational convenience, let us put $W_1(n) = W(n, \tau, 1)$, where τ has been fixed. It can easily be seen that $W_1(n)$ satisfies the following recurrence relation:

$$W_1(n) = \begin{cases} 0 & \text{if } 0 \leq n \leq 2, \\ 1 & \text{if } 3 \leq n < 2\tau, \\ \left\lfloor \frac{n}{\tau} \right\rfloor + W_1\left((\tau - 1) \cdot \left\lfloor \frac{n}{\tau} \right\rfloor\right) & \text{if } n \geq 2\tau. \end{cases}$$

By induction on n, we can show that $W_1(n) \leq n$. For $n < 2\tau$, our estimate is trivially true. Thus, let $n \geq 2\tau$. Then, by inductive hypothesis, we have

$$\begin{aligned} W_1(n) &= \left\lfloor \frac{n}{\tau} \right\rfloor + W_1\left((\tau - 1) \cdot \left\lfloor \frac{n}{\tau} \right\rfloor\right) \leq \left\lfloor \frac{n}{\tau} \right\rfloor + (\tau - 1) \cdot \left\lfloor \frac{n}{\tau} \right\rfloor \\ &= \left\lfloor \frac{n}{\tau} \right\rfloor \cdot (1 + (\tau - 1)) = n. \end{aligned}$$

The number of distance computations made by a call LOCAL_WINNER(X) is equal to $\sum_{i=1}^{|X|}(i-1) = \frac{|X|(|X|-1)}{2}$. At each round of the tournament, all the calls to procedure

LOCAL_WINNER have an argument of size τ, with the possible exception of the last call, which can have an argument of size between $(\tau+1)$ and $(2\tau-1)$. We notice that the last call to procedure FIND_ANTIPOLE made within the **return** instruction of APPROX_ANTIPOLE has argument of size at most ϑ. Since there are $\lceil \log_{\tau/(\tau-1)} n \rceil$ rounds, it follows that the total number of distances computed by a call of APPROX_ANTIPOLE(S), with $|S| = n$, tournament size τ, and threshold ϑ, is majorized by the expression

$$W(n,\tau,\vartheta)\cdot\frac{\tau(\tau-1)}{2}+\lceil\log_{\tau/(\tau-1)} n\rceil\cdot\left[\frac{(2\tau-1)(2\tau-2)}{2}-\frac{\tau(\tau-1)}{2}\right]+\frac{\vartheta(\vartheta-1)}{2}$$
$$=\frac{\tau(\tau-1)}{2}n+\mathcal{O}(\log n+\vartheta^2)\,.$$

By taking $\vartheta = o(\sqrt{n})$, the above expression is easily seen to be $\frac{\tau(\tau-1)}{2}n + o(n)$.

Summing up, we have:

Theorem 3.3.1. *Given an input set of size $n \in \mathbb{N}$, a constant tournament size $\tau \geq 3$, and a threshold $\vartheta = o(\sqrt{n})$, the algorithm* APPROX_ANTIPOLE *performs $\frac{\tau(\tau-1)}{2}n + o(n)$ distance computations.* ■

Concerning the complexity of the faster variant FAST_APPROX_ANTIPOLE, we have the following recurrence relation $W_1(n) = \lfloor \frac{n}{\tau} \rfloor + W_1\left(2\cdot\lfloor\frac{n}{\tau}\rfloor\right)$, for $n \geq 2\tau$. By induction on n, we can show that the number of calls to the subroutine FIND_ANTIPOLE is $W_1(n) \leq \lceil \frac{n}{\tau-2} \rceil$. For $n < 2\tau$, our estimate is trivially true. Thus, let $n \geq 2\tau$. Then, by inductive hypothesis, we have

$$\begin{aligned} W_1(n) &= \left\lfloor\frac{n}{\tau}\right\rfloor + W_1\left(2\cdot\left\lfloor\frac{n}{\tau}\right\rfloor\right) \leq \left\lfloor\frac{n}{\tau}\right\rfloor + \left\lceil\frac{2\cdot\lfloor\frac{n}{\tau}\rfloor}{\tau-2}\right\rceil \\ &\leq \left\lfloor\frac{n}{\tau}\right\rfloor\cdot\left\lceil 1+\frac{2}{\tau-2}\right\rceil \leq \left\lceil\frac{n}{\tau-2}\right\rceil. \end{aligned}$$

Finally, much by the same arguments as those preceding theorem 3.3.1, we can show that the following holds:

Theorem 3.3.2. *Given an input set of size $n \in \mathbb{N}$, a constant tournament size $\tau \geq 3$, and a threshold $\vartheta = o(\sqrt{n})$, the algorithm* FAST_APPROX_ANTIPOLE *performs* $\frac{\tau(\tau-1)}{2(\tau-2)}n + o(n)$ *distance computations.* ■

3.3.2 A Diameter (Furthest Pair) computation on Euclidean spaces

In this subsection we describe an approximation algorithm for the diameter computation on the Euclidean plane. Several studies in the literature [4, 8, 77, 31] have provided efficient algorithms for the approximate diameter computation in multidimensional Euclidean Spaces. Our approach can be regarded as the binary search version of [4]. For the sake of simplicity, we will start with a finite set of points in the plane S. We perform the Antipole search as follows. Let (P_{X_m}, P_{X_M}), (P_{Y_m}, P_{Y_M}) be four points of S having minimum and maximum Cartesian coordinates: the so called minimum area bounding box *BBox*. Notice that such four points belong to the convex hull of the set S and all of S is included in the rectangle bounded by $(P_{X_M}.x - P_{X_m}.x)$ and $(P_{Y_M}.y - P_{Y_m}.y)$ The two endpoints (A, B) of the diameter of such four points constitute our Antipole pair. The Antipole distance (the pseudo-diameter) is not less than $Diagonal/\sqrt{2}$. This yields $\frac{Antipole}{Diameter} \geq \frac{1}{\sqrt{2}}$ proving that our approximation ratio in the plane is $1 - 1/\sqrt{2}$.

Now we describe a generalization of this method giving us an approximation algorithm able to obtain an exponentially arbitrary low approximation factor δ for the real diameter. We perform a $\pi/4$ rotation of our Cartesian coordinates, which implies a bisection of the axes, and compute the maximum and minimum coordinate points for such two new axes. We obtain 8 points. Let A, B be the diameter of this set. It is easy to see (middle picture in Fig. 3.4) that $dist(A, B)/\cos\frac{\pi}{8} > diameter(S)$. By iterating the bisecting process d times, we get $dist(A, B)/\cos\frac{\pi}{2^{d+2}} > diameter(S)$ (see the pseudocode in Fig. 3.5). Therefore the error introduced by the algorithm is:

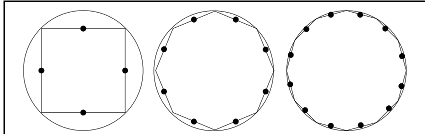


Figure 3.4: Worst cases in the first three iteration of the algorithm.

$$\delta = \frac{|Diameter - Pseudo_Diameter|}{Diameter} \leq \left|1 - \cos\frac{\pi}{2^{(d+2)}}\right|$$

So we can conclude that:

Theorem 3.3.3. *Let S be a set of points in the plane and let $0 < \delta \leq 1 - \sqrt{2}/2$. Iterating the algorithm for $i = 0, \cdots \lceil \log\left(\frac{\pi}{\arccos(1-\delta)}\right) - 2 \rceil$ the algorithm returns an Antipole pair A, B (say Pseudo-Diameter) which approximates the diameter with an error less than δ.* ■

Experiments in n-dimensional Euclidean spaces show that the Antipole pair distance is fully comparable with the first iteration of the above Pseudo_Diameter algorithm, for tournaments with subset size $t = n + 1$. This suggests that one could calculate the intrinsic dimension n of a metric space S and then use tournaments of size $t = n + 1$.

```
DIAGONAL(S)
1  Let BBox = {P_{X_m}, P_{X_M}, P_{Y_m}, P_{Y_M}} be the minimum bounding box of S;
2  V ← {{S}};
3  for i = 1 to ⌈π / (4×arccos(1−δ)) − 1⌉ do
4      V' = ROTATE_SET (V, π/2^{i+1});
5      Let BBox_{π/2^{i+1}} = {P_{X_m}, P_{X_M}, P_{Y_m}, P_{Y_M}}
       be the minimum bounding boxes of the rotated sets in V';
6      V ← Set catalog of V';
7      BBox = BBox ∪ BBox_i;
8  end for
9  return FIND_ANTIPOLE(BBOX);
   END DIAGONAL.
```

Figure 3.5: Pseudo-Diameter Computation.

Chapter 4

The Antipole Clustering: A clustering scheme to index generic metric spaces

Clustering techniques have been studied in statistics, machine learning and database. Each community focuses on different aspects of clustering. These techniques are able to identify clusters in a very large data set, but usually they do not deal with the efficiency of data search and data retrieval.

Recently clustering technique in high-dimensional spaces for efficient indexing have been explored. The well-known problem in such cases is the one about the curse of dimensionality. One solution to high dimensional settings consists in reducing the dimensionality of the input space. Traditional feature selection algorithms select certain dimensions in advance. Methods such as Principal Component Analysis (PCA) [61] transform the original input space into a lower dimensional space by constructing dimensions that are linear combinations of the given features. While PCA may succeed in reducing the dimensionality, the new dimensions can be difficult to interpret, making it hard to understand clusters in relation to the original space. Anyway this scheme is not designed to tackle the search efficiency problem that this study focuses on.

Here we present the Antipole Clustering method. It is an indexing scheme designed to support Range search query and Nearest Neighbor search query.

Range and k-nearest neighbor searching are core problems in pattern recognition. Given a database S of objects in a metric space M and a query object q in M, in a range searching problem the target is to find the objects of S within some threshold distance to q, whereas in a k-nearest neighbor searching problem, the k elements of S closest to q must be produced. These problems can obviously be solved with a linear number of distance calculations, by comparing the query object against every object in the database. However, the goal is to solve such problems much faster.

We combine and extend ideas from the M-Tree, the Multi-Vantage Point structure, and the FQ-Tree to create a new structure in the "bisector tree" class, called the Antipole Tree. Bisection is based on the proximity to an "Antipole" pair of elements generated by a suitable linear randomized tournament. The final winners a, b of such a tournament are far enough apart to approximate the diameter of the splitting set. If $dist(a, b)$ is larger than the chosen cluster diameter threshold, then the cluster is split. The proposed data structure is an indexing scheme suitable for (exact and approximate) best match searching on generic metric spaces. The Antipole Tree compares very well with existing structures such as List of Clusters, M-Trees and others, and in many cases it achieves better results. The Antipole Tree has been widely analyzed also in [110].

4.1 Introduction

Searching is a basic problem in metric spaces. Hence, much efforts have been spent both in clustering algorithms, which are often included in the searching process as a preliminary step (see BIRCH [142], DBSCAN [55], CLIQUE [6], BIRCH* [64], WaveClusters [119], CURE [76], CLARANS [103]) and in the development of new indexing techniques (see, for instance, MVP-Tree [22], M-Tree [41], SLIM-Tree [89],

FQ-Tree [140], List of Clusters [33], SAT [101]; the reader is also referred to [35] for a survey on this subject). For the special case of Euclidean spaces, one can see [5, 67, 14], X-Tree [11] and CHILMA [124].

We combine and extend ideas from the M-Tree, MVP-Tree, and FQ-Tree structures together with randomized techniques coming from the approximate algorithms community [9], to design a simple and efficient indexing scheme called Antipole Tree. This data structure is able to support range queries and k-nearest neighbor queries in generic metric spaces.

The Antipole Tree belongs to the class of "bisector trees" [35, 28, 104], which are binary trees whose nodes represent sets of elements to be clustered. Its construction begins by first allocating a root r and then selecting two splitting points c_1, c_2 in the input set, which become the children of r. Subsequently, the points in the input set are partitioned according to their proximity to the points c_1, c_2. Then one recursively constructs the tree rooted in c_i associated with the partition set of the elements closer to c_i, for $i = 1, 2$.

A good choice for the pair (c_1, c_2) of splitting points consists in maximizing their distance. For this purpose, we propose a simple approximate algorithm based on tournaments of the type described in [9]. Our tournament is played as follows. At each round, the winners of the previous round are randomly partitioned into subsets of a fixed size τ and their 1-medians[1] are discarded. Rounds are played until one is left with less than 2τ elements. The farthest pair of points in the final set is our Antipole pair of elements.

This chapter is organized as follows. In the next section, we give the basic definition of range search and k-nearest-neighbor queries in general metric spaces and we briefly review relevant previous work, with special emphasis on those structures which

[1]We recall that the 1-median of a set of points S in a metric space is an element of S whose average distance from all points of S is minimal.

have been shown to be the most effective, such as List of Clusters [33], M-Trees [41] and MVP-Trees [22]. The Antipole Tree is described in Section 4.3. In section 4.4, we present a procedure for range searching on the Antipole Tree. Section 4.5 presents an algorithm for the exact k-nearest neighbor problem. The Antipole Tree is experimentally compared with List of Clusters, M-Tree and MVP-Tree in Section 4.6. In particular, cluster diameter threshold tuning is discussed. An approximate k-nearest neighbor algorithm is also introduced in Section 4.7 and a comparison with the version for approximate search of List of Clusters [27] is given with a precision-recall analysis. In Section 4.8 we deal with the problem of the curse of dimensionality. Indeed in high dimension, linear scan for uniform data sets may become competitive with the best searching algorithms. However most of the real world data sets are non-uniform. We successfully compare our algorithm with linear scan in non-uniform data sets of very high dimensional Euclidean spaces. Next, in section 4.9 the secondary memory management is discussed and comparisons with M-Tree are showed. Finally, in section 4.10 we show how to deal with the dynamic updates.

4.2 Basic Definitions and Related Works

Let M be a non-empty set of objects and let $dist : (M \times M) \longrightarrow \mathbb{R}$ be a function such that the following properties hold:

1. $(\forall\, x,\, y \in M)\, dist(x, y) \geq 0$ (positiveness);
2. $(\forall\, x,\, y \in M)\, dist(x, y) = dist(y, x)$ (symmetry);
3. $(\forall\, x \in M)\, dist(x, x) = 0$ (reflexivity),
   $(\forall x,\, y \in M)\,(x \neq y \rightarrow dist(x, y) > 0)$ (strict positiveness);
4. $(\forall\, x,\, y,\, z \in M)\, dist(x, y) \leq dist(x, z) + dist(z, y)$ (triangle inequality);

then the pair $(M, dist)$ is called a *metric space* and *dist* is called its *metric function*. Well known metric functions include Manhattan distance, Euclidean distance, string edit distance, or the shortest path distance through a graph. Our goal is to build a low cost data structure for the range search problem and k-nearest neighbor searching in metric spaces.

Definition 4.2.1. *(Range query). Given a query object q, a database S, and a threshold t, the Range Search Problem is to find all objects $\{o \in D | dist(o, q) \leq t\}$.*

Definition 4.2.2. *(k-Nearest Neighbor query). Given a query object q and an integer $k > 0$, the k-Nearest Neighbor Problem is to retrieve the k closest elements to q in S.*

Our basic cost measure is the number of distance calculations since these are often expensive in metric spaces, e.g. when computing the editing distance among strings. Three main sources of ideas have contributed to our work. The FQ-Tree [140], an example of a structure using pivots (see [35] for an extended survey), organizes the items of a collection ranging over a metric space into the leaves of a tree data structure. Viewed abstractly, FQ-Trees consist of a vector of reference objects $r_1, \cdots, r_k$ and a distance vector v_o associated with each object o such that $v_o[i] = dist(o, r_i)$. A query object q computes a distance to each reference object, thus obtaining a v_q. Object o cannot be within a threshold distance t from q if for any i, $v_q[i] > v_o[i] + t$. That is, even if o is closer to q than r_i, q cannot be closer to o than t.

We use a similar idea except that our reference objects are the centroids of clusters. M-Trees [41, 39] are dynamically balanced trees. Nodes of an M-Tree store several items of the collection provided that they are "close" and "not too numerous". If one of these conditions is violated, the node is split and a suitable sub-tree originating in the node is recursively constructed. In the M-Tree, each parent node corresponds to a cluster with a radius and every child of that node corresponds to a subcluster with a smaller radius. If a centroid x has a distance $dist(x, q)$ from the query object and the radius of the cluster is r, then the entire cluster corresponding to x can be

discarded if $dist(x, q) > t + r$.

We take the idea that a parent node corresponds to a cluster and its children nodes are subclusters of that parent cluster from the M-Tree. The main differences between our algorithm and the M-Tree are the construction method, that clusters in the M-Tree must have a limited number of elements, and the search strategy as our algorithm produces a binary tree data structure.

VP-Trees [128, 141] organize items coming from a metric space into a binary tree. The items are stored both in the leaves and in the internal nodes of the tree. The items stored in the internal nodes are the "vantage points". To process a query requires the computation of the distance between the query point and some of the vantage points. The construction of a VP-Tree partitions a data set according to the distances that the objects have with respect to a reference point. The median value of these distances is used as a separator to partition objects into two balanced subsets (those as close or closer than the median and those farther than the median). The same procedure can recursively be applied to each of the two subsets.

The Multi-Vantage-Point tree [22] is an intellectual descendant of the vantage point tree and the GNAT [23] structure. The MVP-Tree appears to be superior to the previous methods. The fundamental idea is that, given a point p, one can partition all objects into m partitions based on their distances from p, where the first partition consists of those points within distance d_1 from p, the second consists of those points whose distance is greater than d_1 and less than or equal to d_2, etc. Given two points, p_a and p_b, the partitions $a_1, \cdots, a_m$ based on p_a and the partitions $b_1, \cdots, b_m$ based on p_b can be created. One can then intersect all possible a- and b-partitions (i.e. a_i intersect b_j for $1 \leq i \leq m$ and $1 \leq j \leq m$) to get m^2 partitions. In an MVP-Tree, each node in the tree corresponds to two objects (vantage points) and m^2 children, where m is a parameter of the construction algorithm and each child corresponds to a partition. When searching for objects within distance t of

query point q, the algorithm does the following: given a parent node having vantage points p_a and p_b, if some partition Z has the property that for every object $z \in Z$, $dist(z, p_a) < d_z$ and $dist(q, p_a) > d_z + t$, then Z can be discarded. There are other reasons for discarding clusters, also based on the triangle inequality. Using multiple vantage points together with pre-computed distances reduces the number of distance computations at query time. Like the MVP-Tree, our structure makes aggressive use of the triangle inequality.

Another relevant recent work, due to Chávez et al. [33], proposes a structure called List of Clusters. Such list is constructed in the following way: starting from a random point, a cluster with bounded diameter (or limited number of objects) centered in that random point is constructed. Then such a process is iterated by selecting a new point, for example the farthest from the previous one, and constructing another cluster around it. The process terminates when no more points are left. Authors experimentally show that their structure outperforms other existing methods when parameters are chosen in a suitable way.

Other sources of inspiration include [25, 43, 60, 74, 118, 116, 89, 101].

4.3 The Antipole Tree

Let (M, $dist$) be a finite metric space, let S be a subset of M and suppose that we aim to split it into the minimum possible number of clusters whose radii should not exceed a given threshold σ. This problem has been studied by Hochbaum and Maass [83] for Euclidean spaces. Their approximation algorithm has been improved by Gonzalez in [75]. Similar ideas are used by Feder and Greene [56] (see [109] for an extended survey on clustering methods in Euclidean spaces).

The Antipole clustering of bounded radius σ is performed by a recursive top-down procedure starting from the given finite set of points S and checking at each step if a given splitting condition Φ is satisfied. If this is not the case, then splitting is not

performed, the given subset is a cluster, and a centroid having distance approximatively less than σ from every other node in the cluster is computed by the procedure described in Section 3.1.

Otherwise, if Φ is satisfied then a pair of points $\{A, B\}$ of S called the Antipole pair is generated by the algorithm described in 3.3 and is used to split S into two subsets S_A and S_B obtained by assigning each point p of S to the subset containing the endpoint closest to p of the Antipole $\{A, B\}$. The splitting condition Φ states that $dist(A, B)$ is greater than the cluster diameter threshold corrected by the error coming from the Euclidean case analysis described in 3.3.2. Indeed the diameter threshold is based on a statistical analysis of the pairwise distances of the input set (see Section 4.6.2) which can be used to evaluate the intrinsic dimension [35] of the metric space. The tree obtained by the above procedure is called an Antipole Tree. All nodes are annotated with the Antipole endpoints and the corresponding cluster radius; each leaf contains also the 1-median of the corresponding final cluster. Its implementation is described in Section 4.3.1.

4.3.1 The Antipole Tree data structure in general metric spaces

The Antipole Tree data structure can be used in a generic metric space $(M, dist)$ where $dist$ is the distance metric. Each element of the metric space along with its related data constitutes a type called *object*. An *object* O (Fig. 4.1 (a)) in the Antipole data structure contains the following information: an element x, an array D_V storing the distances between x and all its ancestors (the Antipole pairs) in the tree, and a variable D_C containing the distance from the centroid C of x's cluster. A data set S is a collection of *objects* drown from M. Each cluster (Fig. 4.1 (b)) stores the following information:

- *centroid*, C, the element that minimizes the sum of the distances from the other

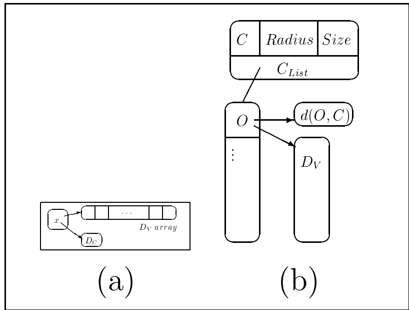


Figure 4.1: (a) A generic *object* in the Antipole data structure. (b) A generic cluster in the Antipole data structure.

cluster members;

- *radius*, *Radius*, containing the distance from C to the farthest *object*;
- *member list*, C_{List}, storing the catalogue of the *objects* contained in the cluster;
- *size of* C_{List}, *Size*, stored in the cluster.

The Antipole data structure has internal nodes and leaf nodes.

- An internal node stores (i) the identities of two Antipole objects A and B, called the Antipole pair of distance at least 2σ apart, (ii) the radii Rad_A and Rad_B of the two sub-sets (S_A, S_B obtained by splitting S based on their proximity to A and B respectively), and (iii) pointers to the left and right sub-trees *left* and *right*;
- A leaf node stores a cluster.

To build such a data structure, the procedure BUILD (see Fig. 4.2) takes as input the data set S, a target cluster radius σ, and a set Q (empty at the beginning). The algorithm starts by checking if Q is empty and if so, it calls the sub-routine

The Build Antipole Tree algorithm

BUILD(S, σ, Q)
1 **if** $Q = \emptyset$ **then**
2 $\quad Q \leftarrow$ ADAPTED_APPROX_ANTIPOLE(S,σ);
3 $\quad$ **if** $Q = \emptyset$ **then** // splitting condition Φ fails
4 $\quad\quad T.Leaf \leftarrow TRUE$;
5 $\quad\quad T.Cluster \leftarrow$ MAKE_CLUSTER(S);
6 $\quad\quad$ **return** T;
7 $\quad$ **end if**;
8 **end if**;
9 {A,B} $\leftarrow Q$;
10 $T.A \leftarrow A$;
11 $T.B \leftarrow B$;
12 $S_A \leftarrow \{O \in S \mid dist(O, A) < dist(O, B)\}$;
13 $S_B \leftarrow \{O \in S \mid dist(O, B) \leq dist(O, A)\}$;
14 **for each** $O \in S$;
15 $\quad O.D_V \leftarrow O.D_V \cup$
$\quad\quad \{(A, dist(O, A)), (B, dist(O, B))\}$;
16 **end for each**;
17 $T.Rad_A \leftarrow \max_{O \in S_A} dist(O, A)$;
18 $T.Rad_B \leftarrow \max_{O \in S_B} dist(O, B)$;
19 $T.left \leftarrow$ BUILD(S_A,σ,CHECK(S_A,σ,A));
20 $T.right \leftarrow$ BUILD(S_B,σ,CHECK(S_B,σ,B));
21 **return** T;
END BUILD.

MAKE_CLUSTER(S)
1 $Cluster.C \leftarrow$ APPROX_1_MEDIAN(S);
2 $Cluster.Radius \leftarrow \max_{x \in S} dist(x, Cluster.C)$
3 $Cluster.C_{List} \leftarrow S \setminus \{Cluster.C\}$;
4 **for each** $x \in Cluster.C_{List}$;
5 $\quad x.D_C \leftarrow dist(x, Cluster.C)$;
6 **end for each**;
7 **return** *Cluster*;
END MAKE_CLUSTER.

Figure 4.2: The algorithm Build Antipole Tree and routine MakeCluster.

ADAPTED_APPROX_ANTIPOLE,[2] which returns an Antipole pair. Then the Antipole pair is inserted into Q. Next, the algorithm checks if the splitting condition is true. If this is the case, the set S is divided into S_A and S_B, where the objects closer to A are put in S_A and symmetrically for B. Otherwise a cluster is generated. The other subroutine used in BUILD is CHECK which checks whether there is an object O in S_A (or S_B) that may become the Antipole of A (or B), by using the distances already computed and cached. If an Antipole is found, it is inserted into Q and then the recursive call in BUILD skips the computation of another Antipole pair.

The routine MAKE_CLUSTER (Fig. 4.2) creates a cluster of objects with bounded radius. This procedure computes the cluster centroid C with the randomized algorithm APPROX_1_MEDIAN and then computes the distance between each O in the cluster and C.

The data structure resulting from BUILD is a binary tree whose leaves contain a set of clusters, each of which has an approximate centroid and the radius, based on that centroid, is less than σ. Fig. 4.3 (a) shows the evolution of the data set during the construction of the tree. At the first step, the pair A, B is found by the algorithm ADAPTED_APPROX_ANTIPOLE, then the input data set is split into the subsets S_A and S_B. The second step proceeds as the first for the subset containing A while, for the subset containing B, it produces a cluster since its diameter is less than 2σ. The third and final step produce the final clusters for the subsets containing A_1 and B_1. Fig. 4.3 (b) shows the corresponding Antipole data structure.

Construction time analysis

Let us compute the running time of each routine. Building the Antipole Tree takes quadratic time in the worst case. For example, let us consider a metric space in which

[2] Notice that this algorithm is a variation of FIND_ANTIPOLE that stops when a pair of objects with distance greater than 2σ is found, otherwise it returns an empty set.

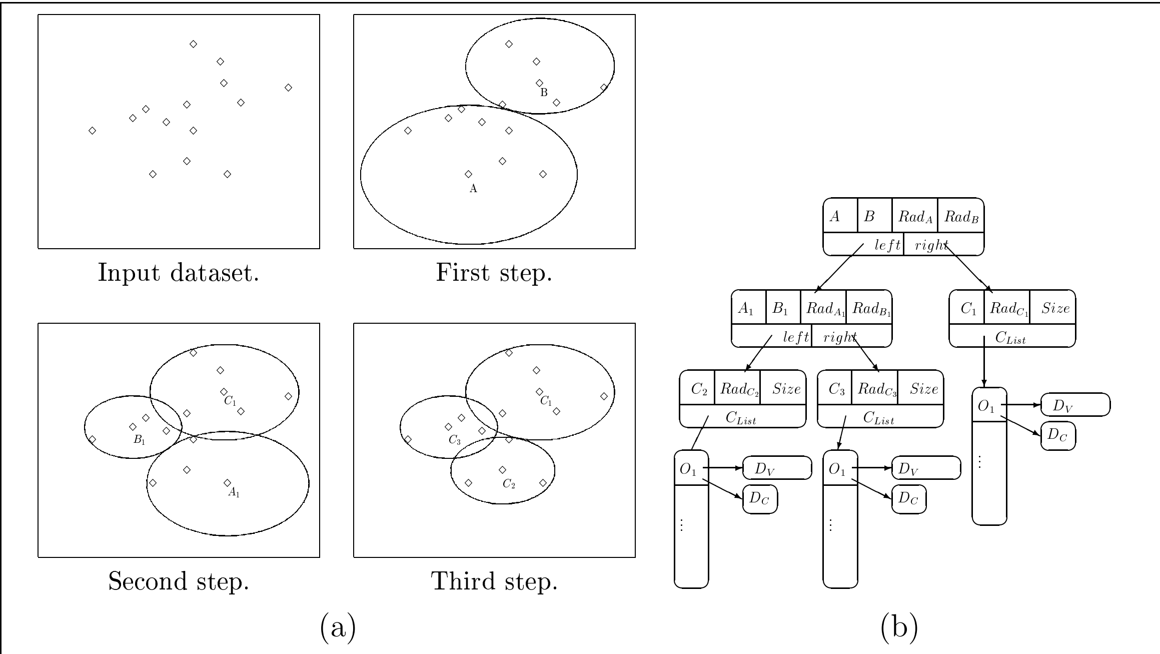


Figure 4.3: A clustering example (a) in a generic metric space and (b) the corresponding Antipole data structure.

the distance between any pair of distinct objects is $2\sigma + 1$. In this case, if the subsets S_A and S_B have size 1 and $|S| - i$ respectively, where i is the i-th recursive call, then the complexity becomes $O(n^2)$. Notice that ADAPTED_APPROX_ANTIPOLE will take constant computational time in this case because all the pairwise distances are supposed to be strictly greater than 2σ.

4.4 Range Search Algorithm

The range search algorithm takes as input the Antipole Tree T, the query object q, the threshold t, and returns the result of the range search of the database with threshold t. The search algorithm recursively descends all branches of the tree until either it reaches a leaf representing a cluster to be visited or it detects a subtree that is certainly out of range and therefore may be pruned out. Such branches are filtered by applying the triangle inequality. Notice that the triangle inequality is used both for exclusion and inclusion. The use for exclusion establishes that an object can be pruned, thus avoiding the computation of the distance between such an object and the query. The other usage establishes that an object must be inserted, because the object is close to its cluster's centroid and the centroid is very close to the query object (see Figs. 4.4 and 4.5 for the pseudocode).

4.5 K-Nearest-Neighbor Algorithm

The k-nearest neighbor search algorithm takes as input the Antipole Tree T, the query object q, and the k parameter indicating the number of objects requested. It returns the set of objects in S which are the k-nearest neighbors of q. Hjaltason and Samet in [81] propose a method called Incremental Nearest Neighbor to perform k-nearest neighbor search in spatial databases. Their approach uses a priority queue storing the subtrees that should be visited, ordered by their distance from the query

```
RANGE_SEARCH(T, q, t, OUT)
 1  if (T.Leaf = FALSE) then
 2     D_A ← dist(q, T.A);
 3     D_B ← dist(q, T.B);
 4     if (D_A ≤ t) then
 5        OUT ← OUT ∪ {T.A};
 6     end if;
 7     if (D_B ≤ t) then
 8        OUT ← OUT ∪ {T.B};
 9     end if;
10     q.D_V ← q.D_V ∪ {D_A, D_B};
11     if (D_A ≤ t + T.Rad_A) then
12        RANGE_SEARCH(T.left, q, t, OUT);
13     end if;
14     if (D_B ≤ t + T.Rad_B) then
15        RANGE_SEARCH(T.right, q, t, OUT);
16     end if;
17     q.D_V ← q.D_V \ {D_A, D_B};
18     return;
19  else // leaf case
20        OUT ← OUT ∪ {VISIT_CLUSTER(T.Cluster, q, t, OUT)};
21  end if;
    END RANGE_SEARCH.
```

Figure 4.4: The Range Search Algorithm.

```
VISIT_CLUSTER(Cluster, q, t, OUT)
 1  q.D_C ← dist(q, Cluster.C);
 2  if (q.D_C ≤ t) then
 3     OUT ← OUT ∪ {Cluster.C};
 4  end if;
 5  if (q.D_C ≥ t + Cluster.Radius) then
 6     return;
 7  end if;
 8  if (q.D_C ≤ t − Cluster.Radius) then
 9     OUT ← OUT ∪ Cluster;
10     return OUT;
11  end if;
12  for each O ∈ Cluster.C_List do
13       if (q.D_C ≥ t + O.D_C) then
14          continue;
15       end if;
16          if (q.D_C ≤ t − O.D_C) then
17             OUT ← OUT ∪ {O};
18             continue;
19          end if;
20          if (∄(d_q ∈ q.D_V and d_O ∈ O.D_V) |
               d_q ≥ t + d_O or d_q ≤ t − d_O) then
21             if (dist(q, O) ≤ t) then
22                OUT ← OUT ∪ {O};
23             end if;
24          else
25                if (d_q ≤ t − d_O) then
26                   OUT ← OUT ∪ {O};
27                end if;
28          end if;
29  end for each;
30  return OUT;
    END VISIT_CLUSTER.
```

Figure 4.5: The Visit Cluster algorithm.

object. The authors claim that their approach can be applied to all hierarchical data structures. Here we propose an application of such a method to Antipole Tree.

The algorithm described below uses two different priority queues. The first one stores the subtrees of the Antipole data structure which may be visited during the search (left sub-tree, right sub-tree or leaf); the second one keeps track of the objects that will be returned as output.

The incremental nearest neighbor algorithm starts by putting the root of the Antipole Tree in the priority queue *pQueue*. Then it proceeds by extracting the minimum from the priority queue. If the extracted node is a leaf (cluster) it visits it. Otherwise it decides to visit each of its subtrees on the basis of the subtree's radius, the distance of the Antipole endpoint from the query, and a threshold t by applying the triangle inequality. The threshold t, which is initialized to ∞, stores the largest distance from the query q to any of the current k-nearest neighbors. Subtrees which need to be visited will be put in the priority queue. All current k-nearest neighbors found are stored in another heap *outQueue* in order to optimize the dynamic operations (such as insertions, deletions and updates). Figs. 4.6 and 4.7 summarize the pseudocode.

4.6 Experimental Analysis

In this section we evaluate the efficiency of constructing and searching through an Antipole Tree. We have implemented the structure using the C programming language under Linux operating system. The experiments use synthetic and real data sets. The synthetic data sets are based on those ones used by [22]:

- uniform 10-dimensional Euclidean space (sets of 100000, 200000, ..., 500000 objects uniformly distributed in $[0, 1]^{10}$);

- clustered 20-dimensional Euclidean space. More precisely a set of 100000 objects

```
K_NEAREST_NEIGHBOR(T, q, t, outQueue, k, pQueue)
 1  Enqueue(pQueue, Tree, NULL);
 2  while NotEmpty(pQueue) do
 3          node = Dequeue(pQueue);
 4          if (node.leaf = TRUE) then
 5             KNN_VISIT_CLUSTER(node, q, t, outQueue, k);
 6          else
 7                 D_A ← CHECK(q, node.A, t, outQueue);
 8                 D_B ← CHECK(q, node.B, t, outQueue);
 9                 Enqueue(pQueue, node.left, D_A-node.Rad_A);
10                 Enqueue(pQueue, node.right, D_B-node.Rad_B);
11          end if;
12  end while;
    END K_NEAREST_NEIGHBOR.
```

Figure 4.6: The incremental k-nearest neighbor search algorithm.

```
CHECK(q, O, t, OUT)
 1  D_O ← dist(q, O);
 2  if (|OUT| < k) then
 3     HEAP_INSERT(O, OUT);
 4     t ← HEAP_EXTRACT_MAX(OUT);
 5  else
 6        if (D_O < t) then
 7           HEAP_INSERT(O, OUT);
 8           t ← HEAP_EXTRACT_MAX(OUT);
 9        end if;
10  end if;
11  return D_O;
    END CHECK.
```

Figure 4.7: A procedure for checking wheter the object O should be added to the OUT set.

obtained in the following way: by using uniform distributions, take 100 random spheres and select 1000 random points in each of them.

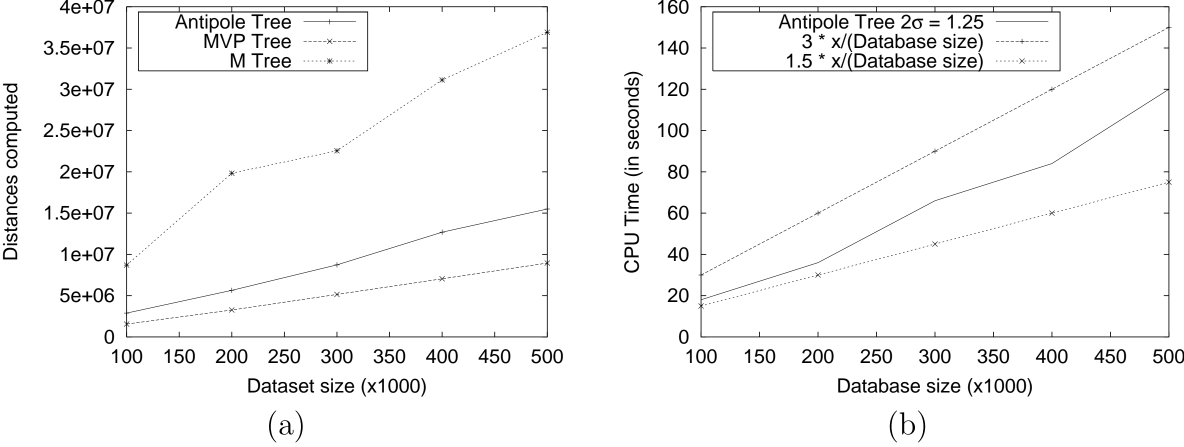
The real data sets are respectively:

- a set of 45000 strings chosen from the Linux dictionary with the editing distance;
- a set of 42000 images chosen from the Corel image database with the metric L_2;
- high dimensional Euclidean space sets of points corresponding to textures of VISTEX database [1] with the metric L_2.

For each experiment, we ran 100 random queries: half of them were chosen in the input set, the remaining ones in the complement.

4.6.1 Construction Time

We measure construction time in terms of the number of distance computations and CPU time on uniformly distributed objects in $[0,1]^{10}$, as described above. Fig. 4.8 (a) illustrates a comparison between the Antipole Tree, the MVP-Tree, and the M-Tree, showing the distances needed during the construction. Data were taken again in $[0,1]^{10}$ with size from 100000 to 500000 elements. The cluster radius σ used was $\sigma = 0.625$, as found by our estimation algorithm described below. We used the parameter settings for MVP-Trees and M-Trees suggested by the authors [22, 39]. Fig. 4.8 (a) shows also that building the Antipole Tree requires fewer distance computations than the M-Tree but more than the MVP-Tree. The difference is roughly a factor of 1.5. Fig. 4.9 shows that the difference in construction costs can be compensated by faster range queries on less than 0.2% of the entire input database. Thus, unless queries are very rare, the Antipole Tree recovers in terms of queries cost what it loses in construction. Experiments proving this fact are reported in Section 4.6.3.



(a) (b)

Figure 4.8: Construction complexity using uniformly generated data. (a) It is measured by the number of distance computations needed by the Antipole Tree with cluster diameter 1.25 vs M-Tree and MVP-Tree. (b) CPU time in seconds needed to build the Antipole Tree.

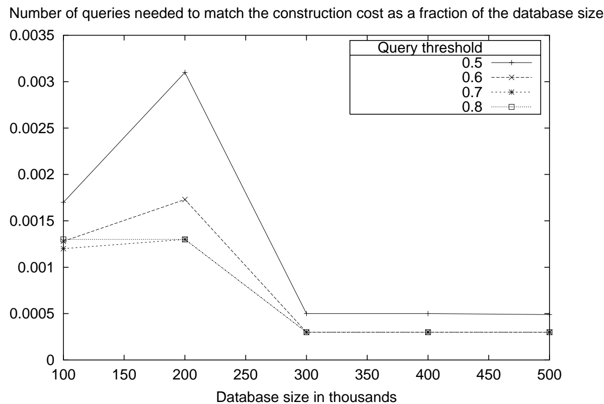


Figure 4.9: Number of range queries, as a fraction of the data set size, which are sufficient to recover the higher cost of Antipole Tree construction with respect to MVP-Tree construction.

Fig. 4.8 (b) shows the CPU time needed to bulk load the proposed data structure; it also shows that the CPU time needed to construct the Antipole Tree grows linearly in many cases. Because the MVP-Tree entails sorting, it requires at least $O(n \log n)$ operations (though not distance calculations) to build the data structure.

4.6.2 Choosing the best cluster diameter

In this section we discuss how to tune the Antipole Tree for range queries. We measure the cost by the number of distance calculations among objects of the underlying metric space.

Before the Antipole data structure can be used, it needs to be tuned. To tune the Antipole Tree, we must choose the radius σ of the clusters very carefully by analyzing the data set properties. In what follows we will show that optimal cluster radius depends on the intrinsic dimensionality of the underlying metric space.

We performed, as described before, our experiments in 10 and 20 dimensional spaces with uniform and clustered distributions having size 100000. However, the methodology of finding the optimal diameter can be applied to other dimensions and arbitrary data sizes.

Figs. 4.10 (Uniform) and (Clustered) show that across different values of the threshold t of the range search, the best choice of the cluster diameter is 0.625 for the uniform data set and 2.5 for the clustered one.

Experiments with real and synthetic data showed that choosing the cluster diameter 10% less than the median pairwise distance value gives, regardless of the range search threshold, a quite surprising result.

4.6.3 Range search analysis and comparisons

In this section we present an extensive comparison among the Antipole Tree, the MVP-Tree, the M-Tree, and List of Clusters in terms of the number of distance

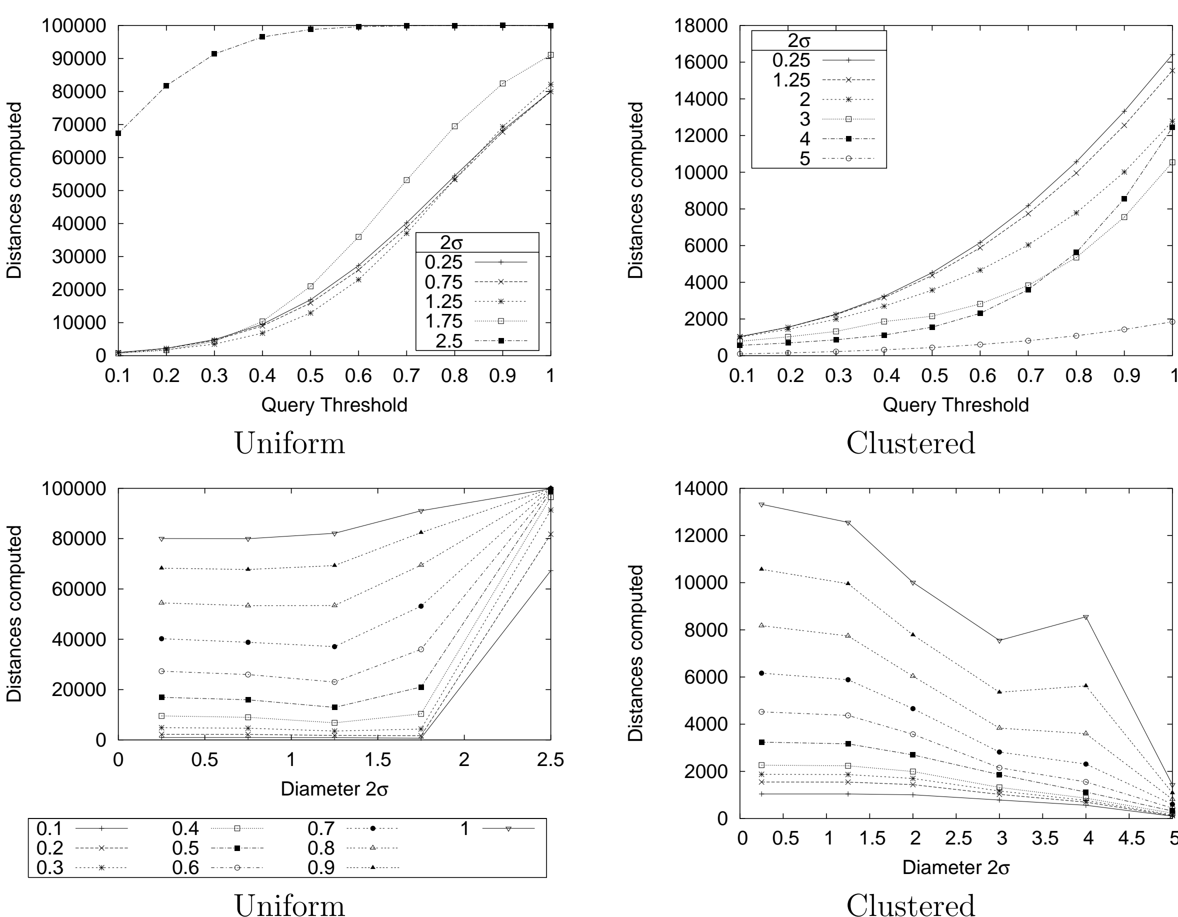


Figure 4.10: Diameter tuning using uniformly and clustered generated points in dimensions 10 and 20, respectively.

computations for range queries. The number of distance computations required by each query has been estimated as the average value in a set of 100 queries. In order to perform a fair comparison with the three competing data structures, MVP-Tree, M-Tree, and List of Cluster, we have set their implementation parameters to the best values according to the ones suggested by the authors. For the MVP-Tree, in [22] it is shown that its best performance is achieved by setting the parameters in the following way:

1. two vantage points in every internal node v_1 and v_2.
2. $m^2 = 4$ partition classes. Four children for each pair of vantage points.
3. $k = 13$ maximum number of objects in a leaf node.
4. p unbounded, the size of the vector storing the distances between the objects in a leaf and their ancestors in the tree (the vantage points). Such a vector is used during the range search to discard objects without having to compute their distance from the query object. Notice that the higher is the dimension of such a vector the more distances from vantage points can be used to prune candidates and this improves the performance of the MVP-Tree in terms of distance computations. For this reason, we have set this parameter to its maximum value: the height of the MVP-Tree.[3]

For the M-Tree implementation, we made use of the *BulkLoading*[4] algorithm [39]. The two parameters needed to tune the data structure in order to obtain better performance are the minimum node utilization and the secondary memory page size. The best performance observed during the search was obtained with minimum node utilization 0.2 and page size 8K.

[3]We are grateful to T. Bozkaya and M. Ozsoyoglu for providing us the program to generate the input for the clustered data set.

[4]We are grateful to P. Ciaccia, M. Patella, and P. Zezula for providing us the source code of the M-Tree.

Concerning List of Clusters, we used fixed bucket size according with heuristics $p3$ and $p5$ suggested by the authors in [33]. $p3$ consists of choosing the center of the i-th cluster as the furthest element from the $(i-1)$-th center, whereas $p5$ picks the element which maximizes the sum of distances from previous centers.

In the first experiment (Fig. 4.11) we compare the four data structures in a uniform data set taken from $[0, 1]^n$ with $n = 10$, varying the query threshold from 0.1 to 0.8, and using a data set of size 300000. For the Antipole, we used two different cluster radii σ: 0.5 and 0.625, respectively. Antipole Tree performs better than the other three data structures computing less distances during the search.

Notice that using a query threshold from 0.1 to 0.7, we capture in the output data set from 0% to 1% of the elements of the entire data set (0.8 captures the 3% of the entire set). Fig. 4.12 shows that with query thresholds from 0.4 to 0.6 we save between 10% and 70% of the distance computations, which in the figure is indicated as the gain percentage.

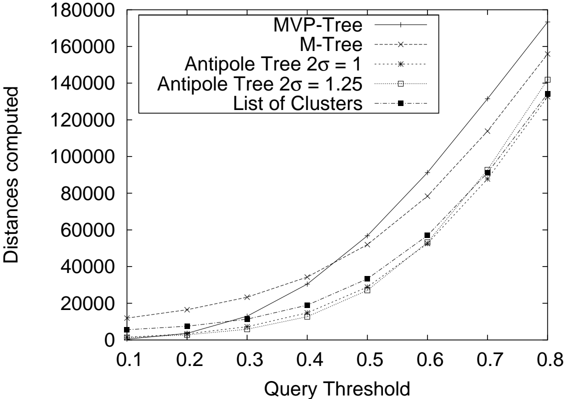


Figure 4.11: Comparisons in $\mathbb{R}^{10}$ using 300000 randomly generated vectors. The query threshold goes from 0.1 to 0.8.

The next set of experiments (see Fig. 4.13) was designed to compare the four data structures in different metric spaces: the clustered Euclidean space $\mathbb{R}^{20}$, a string

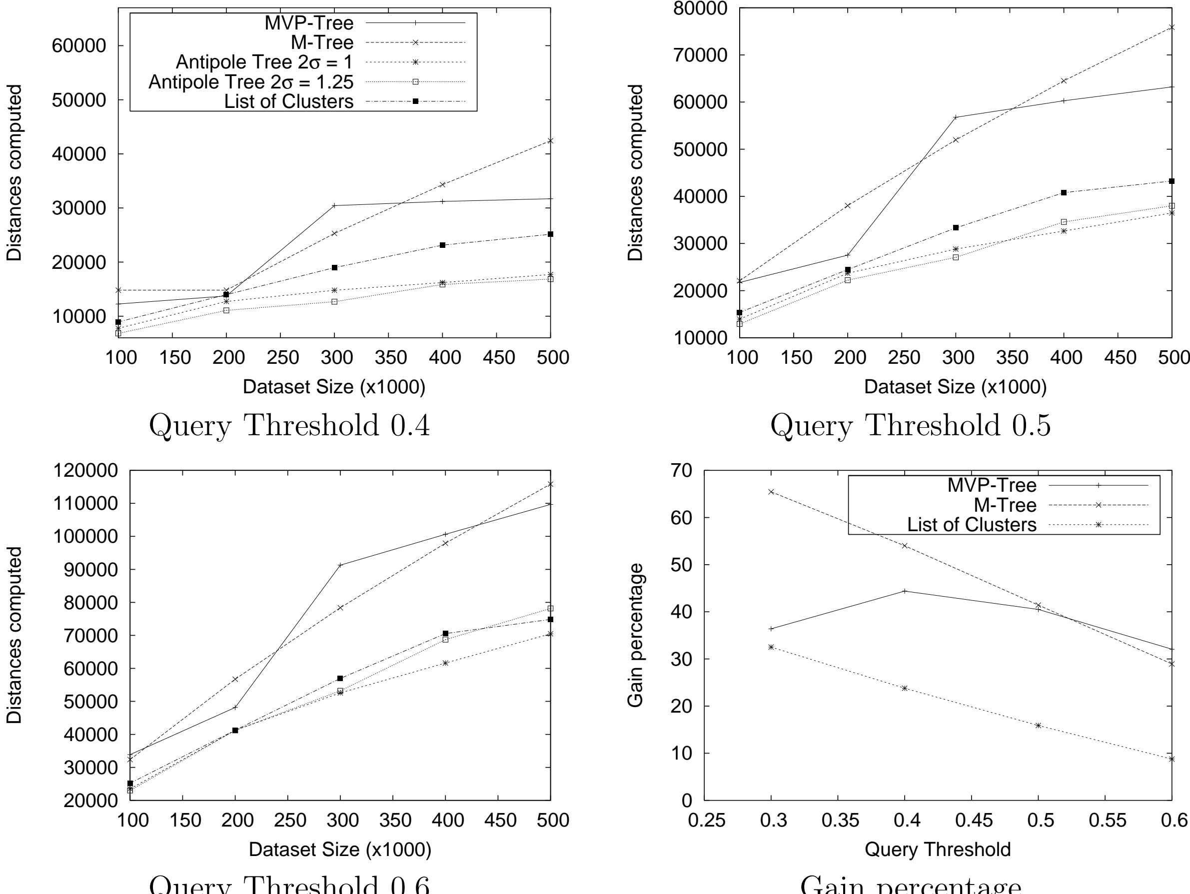


Figure 4.12: Each picture shows the number of distances computed by the compared data structures using threshold from 0.4 to 0.6. The respective gain percentage (percentage of distances saved) of the Antipole Tree w.r.t. the MVP-Tree, the M-Tree, and the List of Clusters is also plotted.

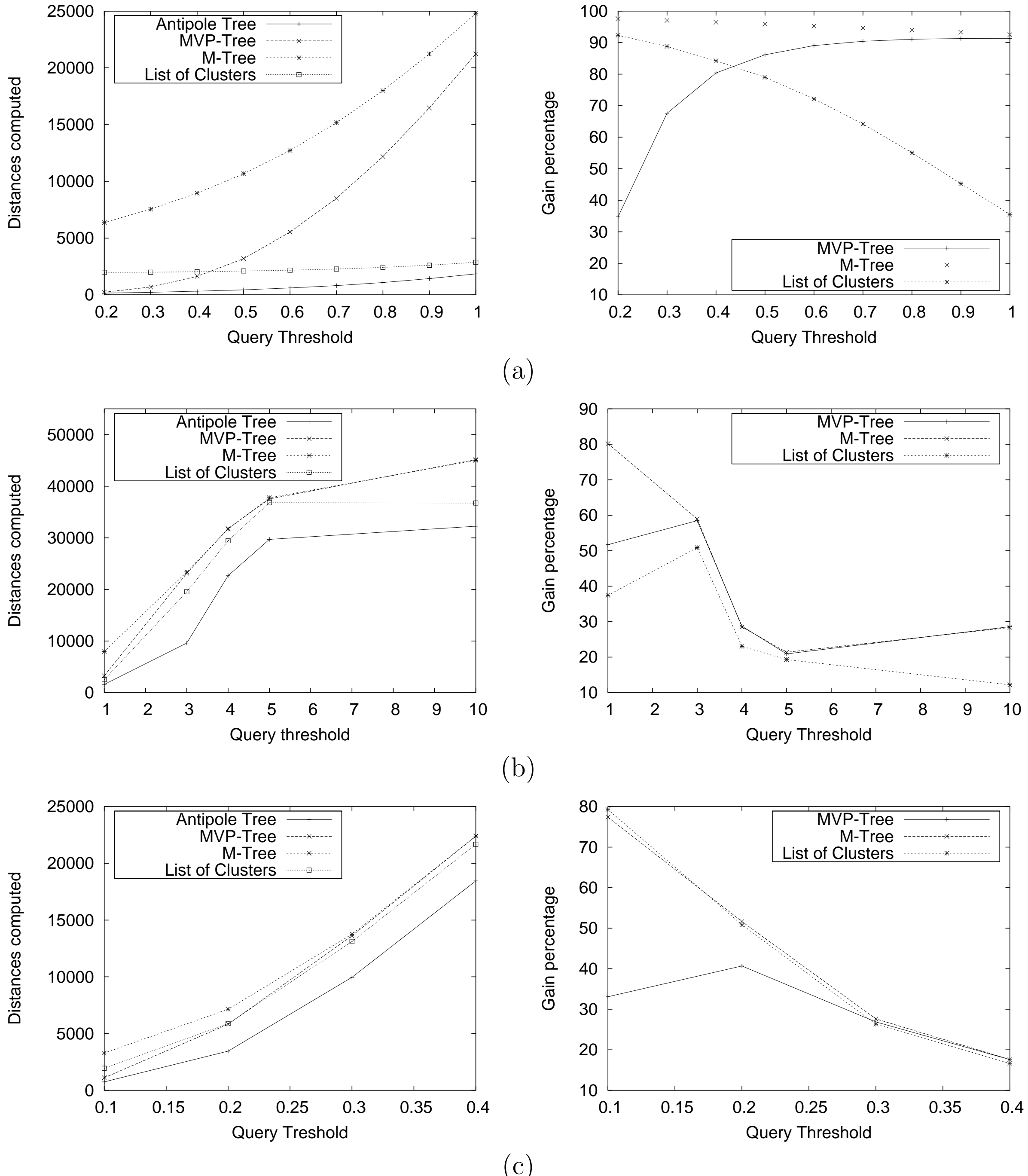


Figure 4.13: (top) Comparisons of Antipole Tree vs. MVP-Tree, M-Tree, and List of Clusters in a clustered space from $\mathbb{R}^{20}$ varying the query threshold from 0.1 to 1, with cluster radius 2. (middle) Antipole Tree vs. MVP-Tree, M-Tree, and List of Clusters using an editing distance metric with cluster radius 5. (bottom) Antipole Tree vs. MVP-Tree, M-Tree, and List of Clusters using a set of image histograms with cluster radius 0.4.

space under an editing distance metric, and an image histogram space with an L_2 distance metric. The corresponding data sets are: 100000 clustered points, 45000 strings from the Linux dictionary, and 42000 image histograms from the Corel image database,[5] respectively. Results show a 30% of savings in distance computations. Since List of Clusters reportedly works well in high dimension, in Fig. 4.14 we show

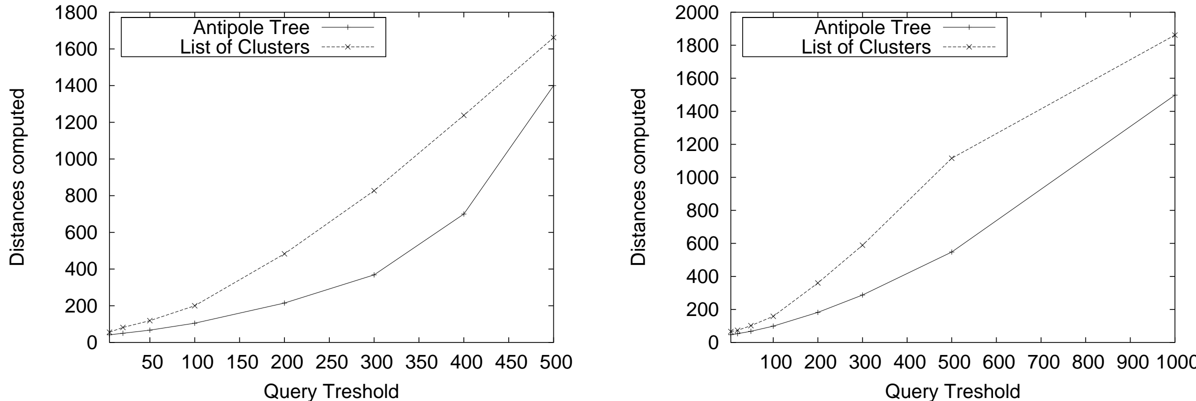


Figure 4.14: A comparison between Antipole Tree and List of Clusters using real database in $\mathbb{R}^{147}$ (left) and $\mathbb{R}^{267}$ (right).

a comparison in range search in very high dimension Euclidean Space $\mathbb{R}^{147}$ and $\mathbb{R}^{267}$, with a database size 3000 obtained from the VISTEX [1] texture database. Notice that by using the query thresholds depicted in Fig. 4.14 the output set captures from 0% to 5% of the elements of the entire data set in $\mathbb{R}^{147}$ and from 0% to 10% of the elements of the entire data set in $\mathbb{R}^{267}$. Antipole Tree shows a better behavior w.r.t. List of Clusters tuned with the best fixed bucket size we noticed.

4.6.4 K-Nearest Neighbor comparisons

In the Fig. 4.15 we present a set of experiments in which the K_NEAREST_NEIGHBOR algorithm is compared with the M-Tree and the List of Clusters. Notice that we

[5]Obtained from the UCI Knowledge Discovery in Databases Archive, http://kdd.ics.uci.edu

compared the Antipole Tree with just the M-Tree and List of Clusters because the k-nearest neighbor search is not discussed for the MVP-Tree (see [22]). As described in Section 4.6.3, we choose uniform and clustered data in $\mathbb{R}^{10}$ and $\mathbb{R}^{20}$. Each data set has size 100000. We run the K_NEAREST_NEIGHBOR algorithm with $k = 1, 2, 4, 6, 8, 10, 15, 20$ using one hundred queries for each experiment (half belonging to the data structure and half not). Using the Antipole Tree we save up to 85% of distance computations.

Concerning experiments in very high dimension, in Fig. 4.16 we show a comparison with List of Clusters using a data set of 3000 elements in Euclidean $\mathbb{R}^{147}$ and $\mathbb{R}^{267}$ from VISTEX [1]. Antipole Tree clearly outperforms List of Clusters.

4.7 Approximate K-Nearest Neighbor search Via Antipole Tree

When the dimension of the space becomes very high (say ≥ 50) all existing data structures perform poorly on range and k-nearest neighbor searches. This is due to the well known problem of the *curse of dimensionality* [87]. Lower bounds [36] show that the search complexity exponentially grows with the space dimension. For generic metric spaces, following [34, 35], we introduce the concept of *intrinsic* dimensionality:

Definition 4.7.1. Let $(M, dist)$ be a metric space, and let $S \subseteq M$. The intrinsic dimension of S is $\rho = \frac{\mu_S^2}{2\sigma_S^2}$, where μ_S and σ_S^2 are the mean and the variance of its histogram distances.

A promising approach to alleviate, at least, the curse of dimensionality is to consider approximate and probabilistic algorithms for k-nearest neighbor search. In some applications, such algorithms give acceptable results. Several interesting algorithms have been proposed in the literature [34, 40, 100, 66]. One of the most successful data structure seems to be the Tree Structure Vector Quantization (TSVQ). Here we will show how to use the Antipole Tree to design a suitable approximate

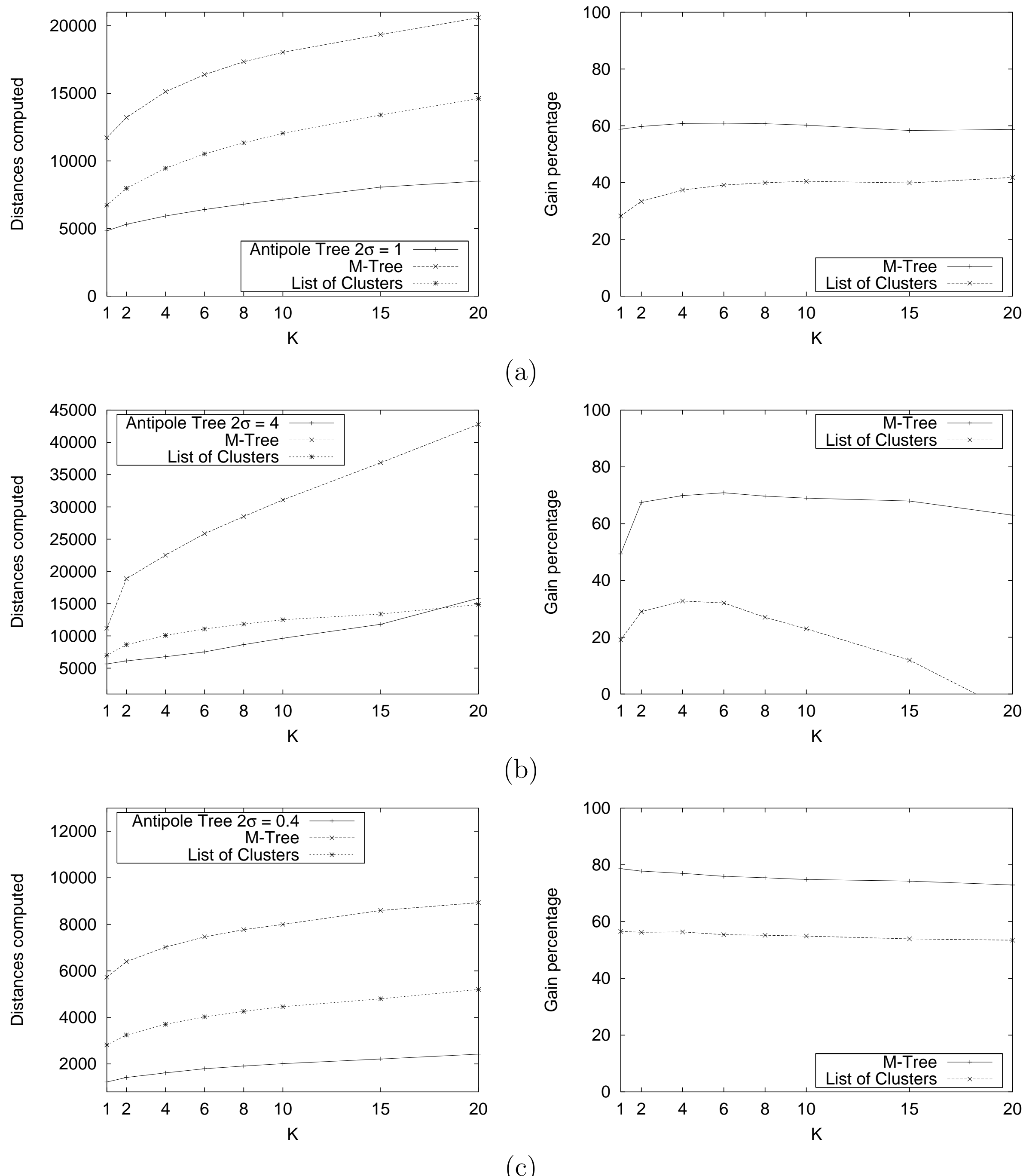


Figure 4.15: k-nearest neighbor comparisons. (a) 100000 uniformly generated points in $[0,1]^{10}$. (b) 100000 points from $\mathbb{R}^{20}$ generated in clusters. (c) Comparisons using the image histogram database.

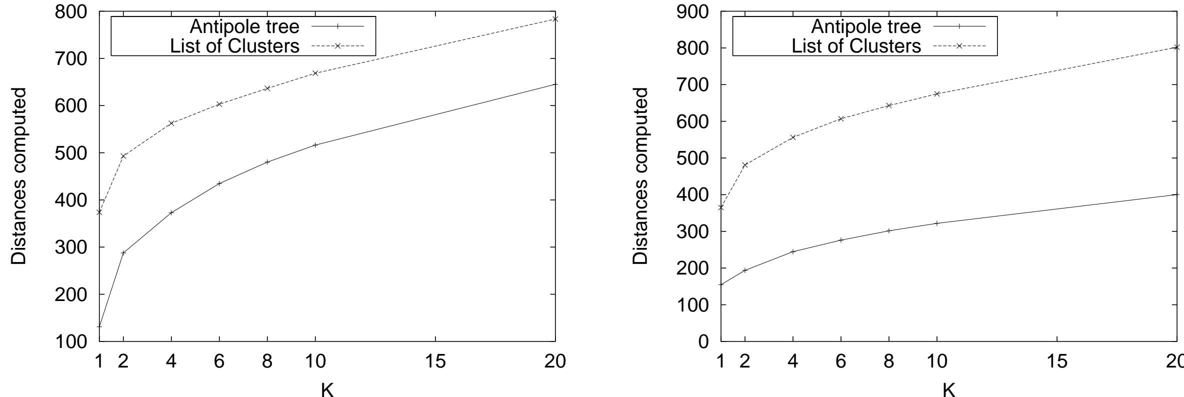


Figure 4.16: k-nearest neighbor search using real data from the VISTEX database in dimension $\mathbb{R}^{147}$ and $\mathbb{R}^{267}$.

search algorithm for the nearest neighbor search. A first simple algorithm, called BEST_PATH_SEARCH, follows the best path in the tree from the root to the leaf, and returns the centroid stored in the leaf node. This algorithm uses the same strategy of the TSVQ to find quickly an approximate nearest neighbor of a query object.

In what follows we present a set of experiments where TSVQ and Antipole Tree are compared. The experiments refer to uniformly generated objects in spaces whose dimension ranges from 10 to 50. For each input data set one hundred queries were executed. In order to evaluate the quality of the results, we run the exact search first. Then the error δ is computed in the following way:

$$\delta = \frac{|dist(O_{opt}, q) - dist(O_{TSVQ/Antipole}, q)|}{dist(O_{opt}, q)}.$$

In Fig. 4.17 (a) the errors introduced by the two approximate algorithms in uniformly generated set of points (upper figures) and clustered set of points (lower figures) are depicted. On the other hand, Figs. 4.17 (b), (d) show the number of distances computed by the two algorithms.

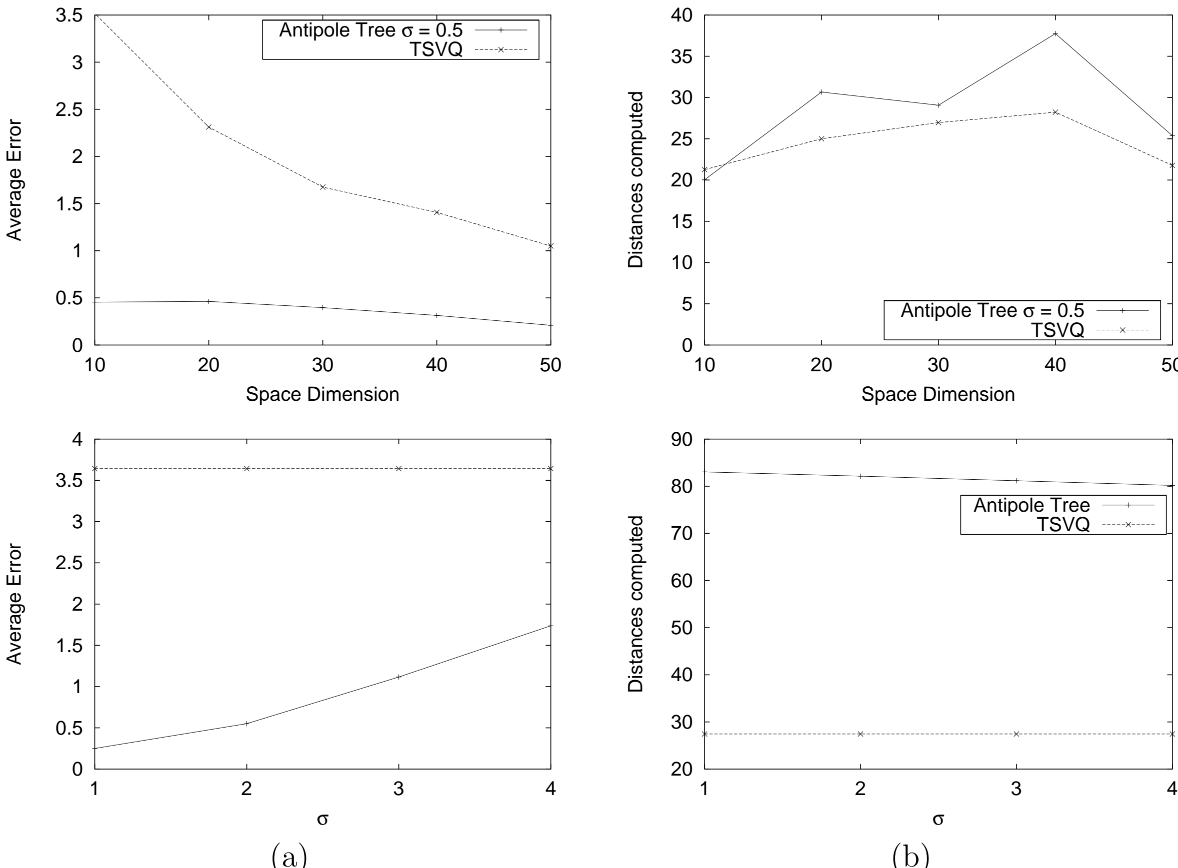


Figure 4.17: A comparison between the approximate Antipole search and TSVQ search. (a) shows the average error introduced by the two algorithms in uniformly generated points with $\sigma = 0.5$ varying the space dimension from 10 to 50. (b) shows the number of distances computed. (c) shows the average error introduced using points generated in clusters of space dimension 20 varying the cluster radius σ. (d) shows the corresponding number of distances needed.

The experiments clearly show that the Antipole Tree improves on TSVQ. We think that this is due to the better position of the Antipole pairs.

A more sophisticated approximation algorithm to solve the k-nearest neighbor problem can be obtained by using the K_NEAREST_NEIGHBOR algorithm. The idea is the following: for each cluster reached during the search, the algorithm compares the query object with the cluster centroid without taking into consideration the objects inside it.

This search is slower than the BEST_PATH_SEARCH but is more precise and can be used to perform k-nearest neighbor search. Fig. 4.18 (a) shows a set of experiments done in uniform spaces in dimension 30 with radius σ set to 1 and 1.5.

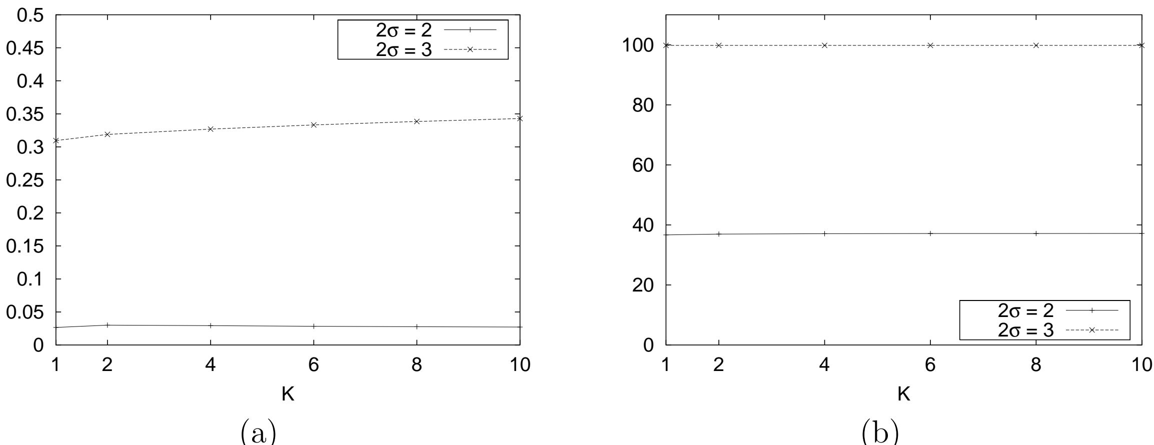


Figure 4.18: An experiment with the approximate k-nearest neighbor algorithm in dimension 30. In (a) the average error is showed. (b) depicts the gain percentage in the number of distance computations.

In approximate matching, precision and recall [93] are important metrics. Following [93], we call the k-nearest neighbor elements of a query q: the k golden results. Then, the *recall* after quota distances can be defined as the fraction of the k top golden elements retrieved fixing a bound, called quota, in the number of distances

that can be computed during the search. The *precision* is the number of golden elements retrieved over the number of distances computed. On the other hand if

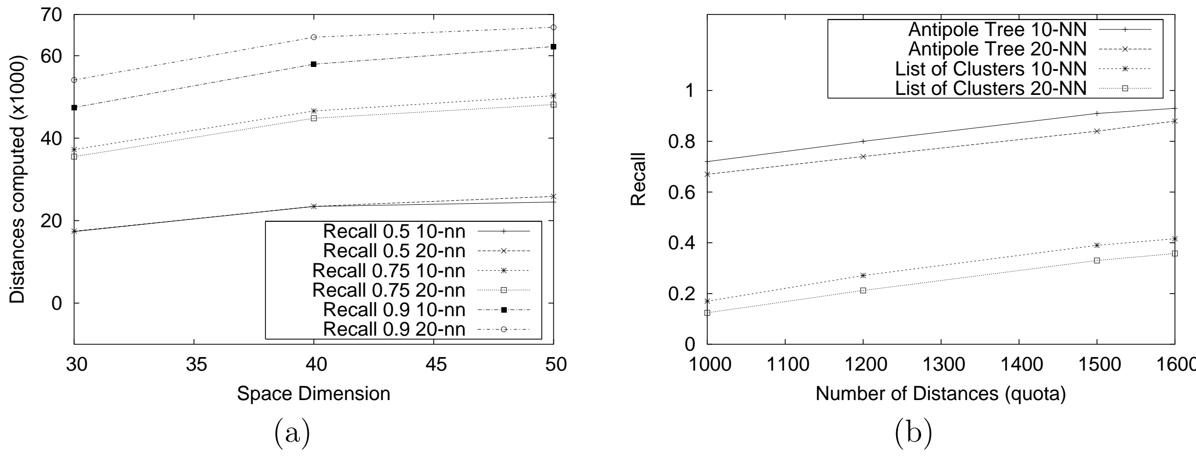


Figure 4.19: (left) Analysis of curse of dimensionality using Antipole Tree from dimension 30 to 50. Number of distances needed fixing the recall. (right) Comparisons using the image histogram database between the Antipole Tree and List of Clusters w.r.t. approximated k-nearest neighbor. The recall varying the quota is depicted.

the recall R is fixed (i.e. 50%), the *R-precision* (precision after R recall) gives the number of distances which must be computed to obtain such recall. We performed precision-recall analysis between Antipole Tree and the approximate version of List of Clusters [27]. Experiments in Fig. 4.20 made use of 100000 elements of dimension 30. We fixed several quotas and recalls ranging from 7000 to 42000 and from 0.5 to 0.9 respectively. Results clearly show that Antipole Tree gives precision-recall factors better than List of Clusters (with fixed bucket size). Fig 4.19 (left) makes the same comparison but using Image histogram database, also it illustrates (right) the effect of curse of dimensionality in precision-recall factor analysis for the Antipole Tree using uniformly distributed objects in Euclidean spaces of dimension ranging from 30 to 50.

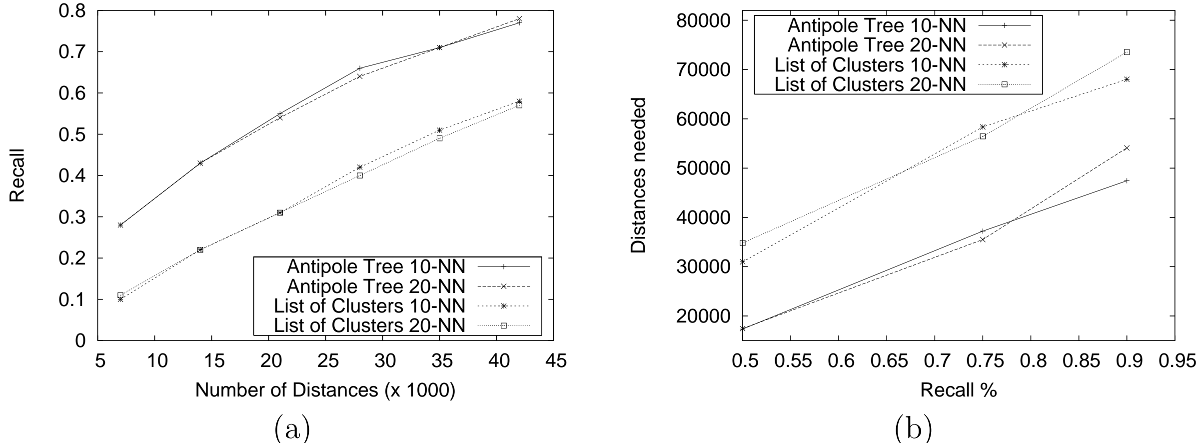


Figure 4.20: Comparing Antipole Tree and List of Clusters w.r.t approximated K-nearest neighbor. In (a) Recall varying the *quota*. (b) Number of distances computation with fixed recall.

4.8 A comparison with linear scan

In this section we present a set of experiments in which we compare the proposed data structure with a naive linear scan. We used a set of very high dimensional Euclidean data sets. Such data sets were obtained from a set of textures taken from the VISTEX database [1]. Starting from a given texture, the data sets of tuples were built in the following way: for each pixel p in the texture we considered, per color channel, half of its $h \times h$ neighborhood (see [133] for more details). We obtained data sets of dimension ranging from 63 to 267. Results, which are plotted in Fig. 4.21, show that the proposed data structure outperforms the linear scan in such high dimensional data sets. We have also noticed that the intrinsic dimension of these spaces goes from 5 to 10.

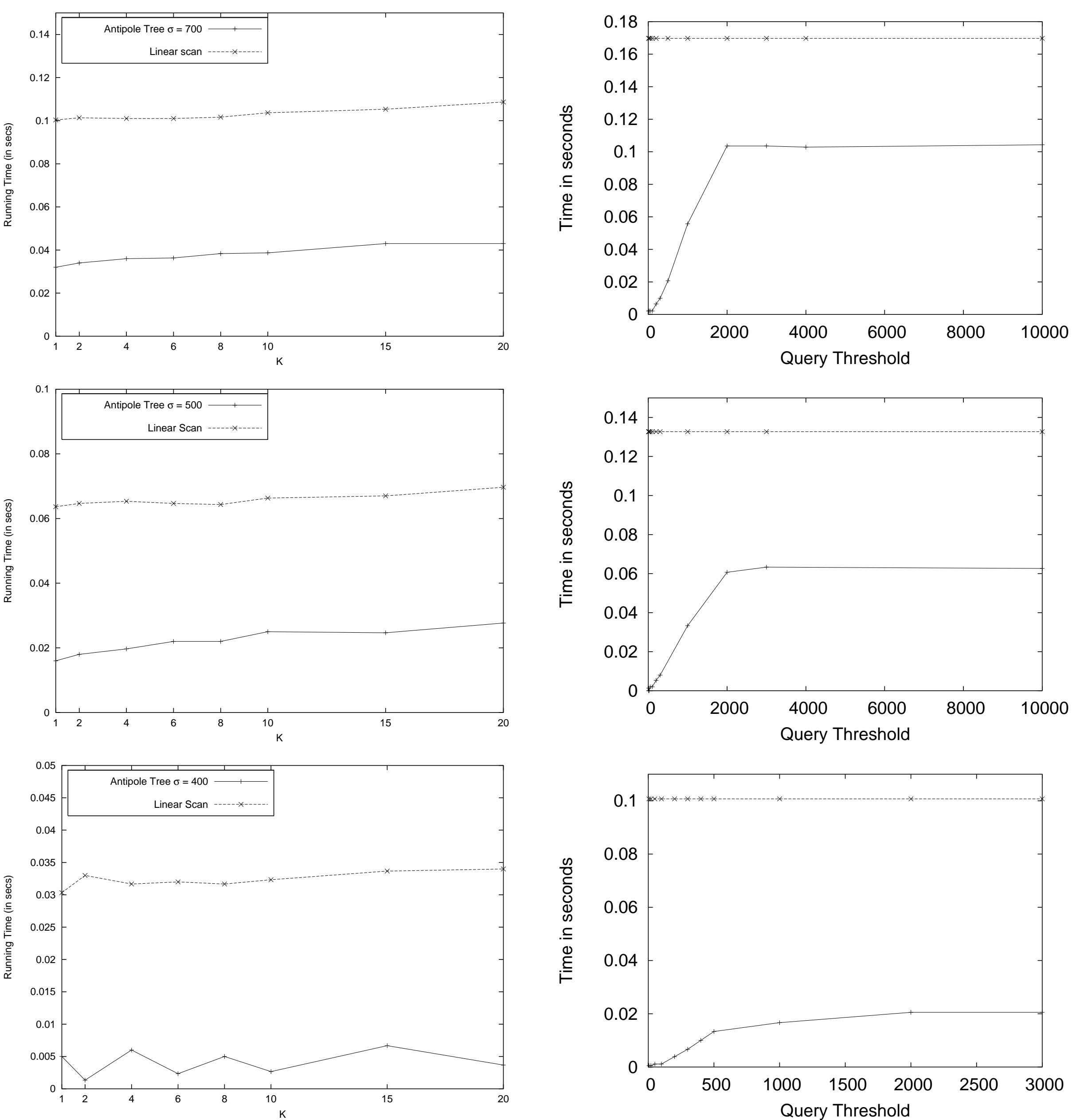


Figure 4.21: Comparing Antipole Tree and linear scan w.r.t. k-nearest neighbor (left side) and range search (right side) in $\mathbb{R}^{267}$ top, $\mathbb{R}^{147}$ middle, and $\mathbb{R}^{63}$ bottom.

Field	size (in byte)
A_1 Antipole Node	$sizeof(Object)$
A_2 Antipole Node	$sizeof(Object)$
R_{A_1} Subset radius	4
R_{A_2} Subset radius	4
$Left$ pointer to the left subtree	4
$Right$ pointer to the right subtree	4

Table 4.1: Internal node information together with the byte needed by each field.

Field	size (in byte)
C Centroid	$sizeof(Object)$
C_{List} Member List	$sizeof(Object) \times \|C_{List}\|$
D_v Ancestor's distance vector for each object	$4 \times \|D_v\|$
R Cluster radius	4

Table 4.2: Leaf node information together with the byte needed by each field.

4.9 Secondary Memory management

As reported in [110], despite growing main memories, it is often not possible to hold the entire database in main memory. Access methods need to integrate secondary and tertiary storage in the same way. For multidimensional spaces several papers have been presented (see [62] for an extended survey). On the other hand for generic metric spaces just the M-Tree [41] and next the SLIM-Tree [89] deal with the secondary storage problem.

Even if the Antipole Tree data structure is a binary tree designed for main memory in this section we present a secondary memory representation of such a data structure. In the Antipole Tree data structure we have two kind of nodes: internal nodes and leaf nodes.

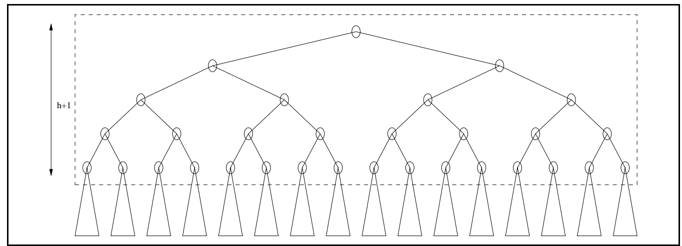


Figure 4.22: Representation of an Antipole Tree in main memory.

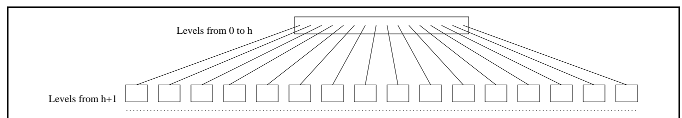


Figure 4.23: Representation of an Antipole Tree in secondary memory.

The internal nodes as reported in table 4.1 (taken from [110]) store information relative to the antipole objects. Each Antipole internal node needs $2 \times sizeof(Object) + 16$ byte. A leaf node (see table 4.2 (taken from [110])) of the Antipole Tree store a cluster of objects. The size of a leaf node is $(C_{List}+1) \times sizeof(Object) + C_{List} \times |D_V| \times 4 + 4$.

The strategy we use to represent the tree in secondary memory was to represent the Tree by level in disk page. Fig. 4.22 (taken from [110]) depicts an Antipole Tree where the first h levels are captured inside a dashed box. These h levels of the tree are relative to internal nodes of the tree and in our representation will be stored in a suitable disk page.

In order to evaluate how many levels can be stored in a single disk page, suppose that each page allow us to store a certain amount of kilobyte m. We can observe we do not need to store the pointer of each node in the page but we need to store the children's pointers for the last level of the Antipole. We call h the maximum depth of the tree stored which can be stored in a single disk page. Using this information we want to obtain the depth h as function of the page size m. Notice that the number

of pointers we need to store in a page are at most 2^{h+1} and given that an address requires 4 byte, we can write:

$$m - 4 \times 2^{h+1} = \sum_{i=0}^{h} 2^i \times [2 \times sizeof(Object) + 8] = (2^{h+1} - 1) \times [2 \times sizeof(Object) + 8];$$

then we obtain:

$$2^{h+1} \times [8 + 2 \times sizeof(Object) + 4] = m + 8 + 2 \times sizeof(Object);$$

$$2^h = \frac{m + 8 + 2 \times sizeof(Object)}{24 + 4 \times sizeof(Object)}$$

$$h = \lg\left(\frac{m + 8 + 2 \times sizeof(Object)}{24 + 4 \times sizeof(Object)}\right)$$

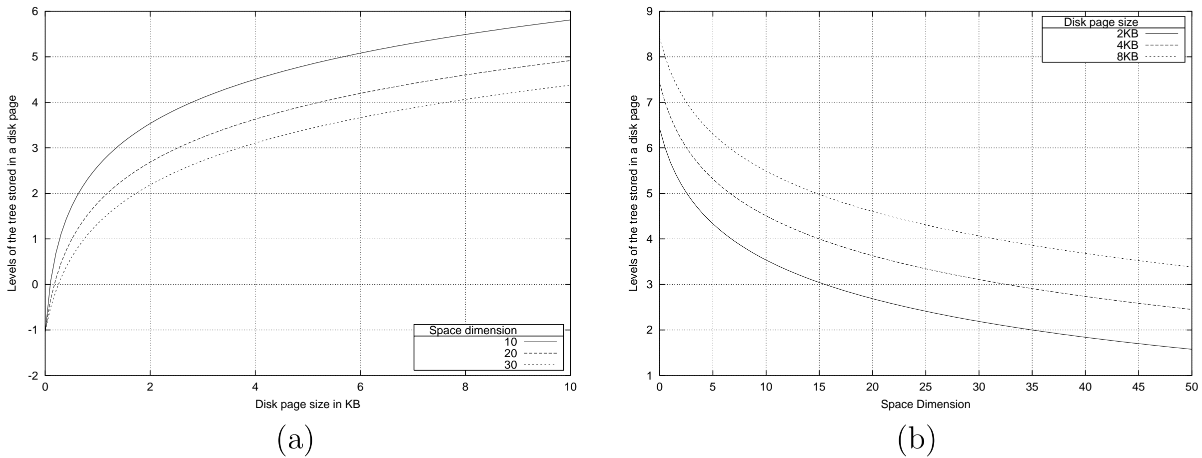


Figure 4.24: Tree levels inside a disk (internal) node function of (a) page size (b) space dimension.

If we suppose that the objects are in a multidimensional space $\mathbb{R}^k$ we obtain that $sizeof(Object) = n*4$ and Fig. 4.24 (a) (taken from [110]) shows the maximum depth which can be stored in a single page while Fig. 4.24 (b) (taken from [110]) shows the maximum depth with respect to the space dimension.

For the leaf nodes of the Antipole Tree in the secondary memory representation instead of store all the distance from the ancestor for each database object O we can maintain just the distances between the object and the cluster centroid. Each cluster will need $(1 + C_{List}) \times (sizeof(Object) + 4)$. Furthermore in a disk page of size m will be store more clusters in order to reduce the fragmentation.

```
MAKE_TREE(Dataset S, Diameter σ, AntipoleFile fp)
 1  Q ← RAND_ANTIPOLE(S,σ);
 2  {A,B} ← Q;
 3  if dist(A,B) ≤ σ then
 4    return MAKE_CLUSTER(S,fp);
 5  end if;
 6  MAKE_NODE(S, σ,0,V_set,fp);
 7  for each X ∈ V_set do
 8      MAKE_TREE(X, σ, fp);
 9  end for each;
10  return;
    END MAKE_TREE.
```

Figure 4.25: Antipole Tree construction on secondary memory.

To build the Antipole Tree on secondary memory we used the recursive procedure of Fig. 4.25(taken from [110]). It makes use of the function MAKE_NODE (see Fig. 4.26 (taken from [110])) to construct an internal page (I_PAGE) recursevely until the maximum depth (say $MAXDEPTH$) is reached. MAKE_NODE returns a set of subsets of S (say V_{set}) and for each subset $X \in V_{set}$ is called recursively MAKE_NODE. The MAKE_CLUSTER function builds a disk page storing the all cluster informations.

Fig. 4.27 (taken from [110]) shows an Antipole Tree and its secondary memory representation.

```
MAKE_NODE(Dataset S, Diameter σ, int C_depth, set V_set, AntipoleFile APfile, int I_PAGE)
 1  Q ← RAND_ANTIPOLE(S,σ);
 2  {A,B} ← Q;
 3  if dist(A,B) ≤ σ then
 4     V_set ← V_set ∪ S;
 5     return;
 6  end if;
 7  if C_depth ≤ MAXDEPTH then
 8     S_l ← {O ∈ S | dist(O,A) < dist(O,B)};
 9     S_r ← S \ S_l;
10     ADD_DATA(Q, I_PAGE);
11     MAKE_NODE (S_l, σ,C_depth + 1,V_set,APfile, I_PAGE);
12     MAKE_NODE (S_r, σ,C_depth + 1,V_set,APfile, I_PAGE);
13  else
14        V_set ← V_set ∪ S;
15  return;
    END MAKE_NODE.
```

Figure 4.26: Internal disk page node construction.

4.9.1 Searching in Secondary memory via Antipole Tree

As widely discussed in [110], the Antipole Tree in secondary memory allows range search as the one presented for the main memory. The visit starts from the root page of the tree. The internal structure of a disk page is an Antipole Tree and comparisons based on triangle inequality can be applied. Once the page is accessed a catalog containing the pages which need to be visited is returned and the search proceeds starting from the first element in the catalog.

A preliminary experimentation of this technique (see Fig. 4.28 (taken from [110])) was done using a uniformly generated data set from $\mathbb{R}^{10}$ with 100 queries with disk page size $4kb$. The average number of I/O needed (the number of disk pages) to perform the search were computed and the results obtained were compared with the one obtained by using the M-Tree. As can be seen from the picture the Antipole Tree

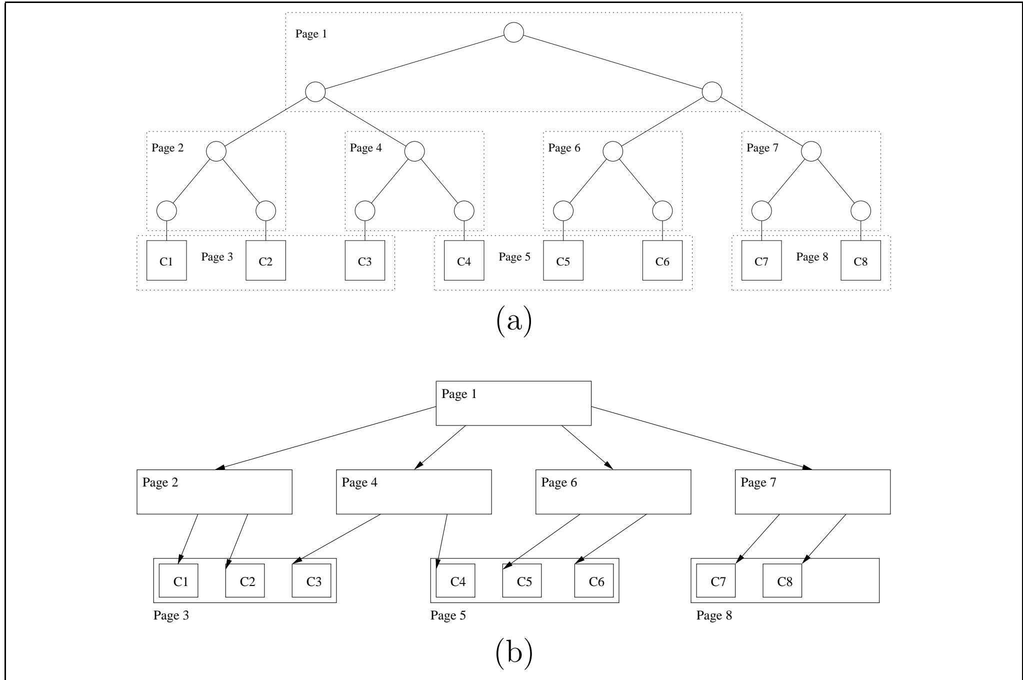


Figure 4.27: (a) An Antipole Tree main memory representation with 8 clusters. (b) Antipole Tree secondary memory representation with $MAXDEPTH = 1$, storing up to 3 clusters in each leaf.

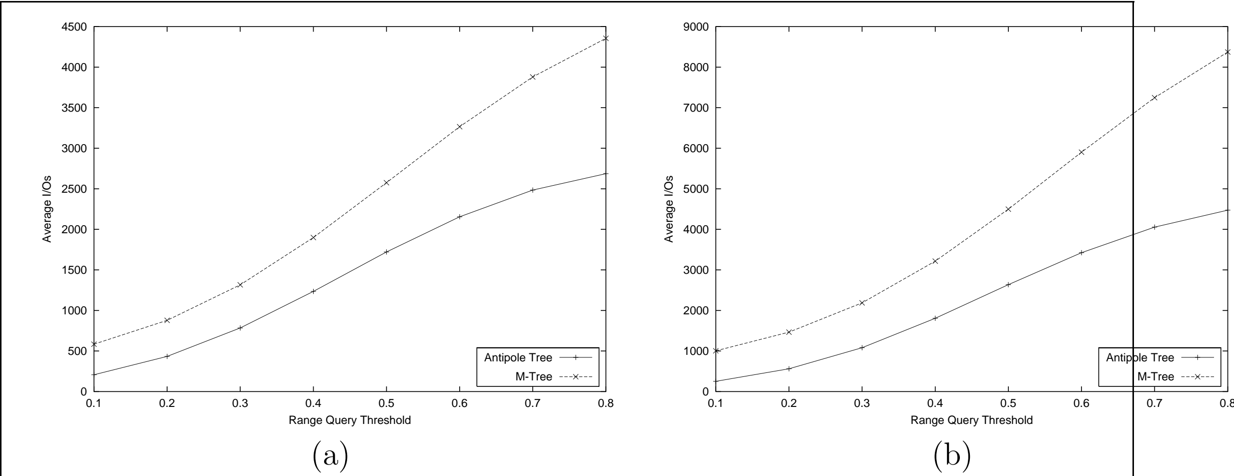


Figure 4.28: Range search queries in secondary memory using Antipole Tree and M-Tree: (a) Database size 100000, (b) Database size 200000.

needs much less I/O accesses than the M-Tree to perform range search.

4.10 Dealing with Dynamic Updates

The construction of the Antipole Tree we presented builds the data structure from scratch. Thus it is suitable when data is static. In several applications databases change dynamically due to insertions and deletions of objects. As reported in [110], here we discuss an approach to make the Antipole Tree able to deal with dynamic insertions. Notice that we do not deal with the deletion problem because we use the assumption that objects will be never removed from the database.

The process to insert an object into the Antipole Tree proceeds as follows. Starting from the root it follows the best path in the tree as performed during the approximate search seen in section 4.7. During the search process the object O will be closer to one of the antipole objects (say A) but can have distance from A greater than Rad_A, in such a case we update $Rad_a = dist(A, O)$. The search goes until a cluster is reached, then the distance between the object O and the cluster centroid C is computed. Depending on this distance two cases might occur:

1. the distance is less than σ then the object is inserted into the cluster;
2. the distance is greater or equal than σ and the cluster is splitted into two new clusters introducing one new internal node and one new leaf node.

For the secondary memory version of the Antipole Tree the insertion follows the same approach. The algorithm looks for the leaf disk page containing the cluster by traversing the tree. When the page is reached wether the cluster needs to be splitted or the its dimension exceeds the available size in the disk page two cases are possible:

1. if the father of such a page is full, then a new internal disk page must be allocated, the addresses stored inside the father page must be updated and this new internal page will contain the address of a new leaf page that will be inserted to store the cluster;
2. if the page is not full, then the new antipole obtained by splitting the cluster will be inserted into it and the address pointing the leaf nodes will be updated.

Chapter 5

Applications

This chapter will introduce four successful applications of the Antipole Clustering. These applications lie in different computer science areas. The first one deals with the BioInformatic area; given a protein network, we use the Antipole clustering to group the proteins according to their connections. Next we will present an innovative technique to cluster a set of labeled graphs.

The two final applications lie in the field of computer graphics. We use the Antipole clustering and its nearest neighbor algorithm to solve the two problems of *texture synthesis* and the *image colorization*. A texture is an image which has a pattern characterizing it. The pattern of the texture or the texture sample can be used to synthesize a new image that, when perceived by a human viewer, seems to be generated by the same underlying stochastic process. Texture synthesis has applications in image compression and image zooming. In the image colorization application the Antipole clustering is used to transfer colors from a source image (colored) to a destination image (gray-scaled).

5.1 Protein Interaction Network

Recent advances in proteomics technologies such as two-hybrid, phage display and mass spectrometry have created the possibility of developing a detailed map of biomolecular interaction networks. Initial mapping efforts have already produced a wealth of data. As the size of the interaction set increases, databases and computational methods will be required to store, visualize, and analyze the information in order to effectively aid in knowledge discovery. In this section, the Antipole Clustering will be used as a graph clustering algorithm that detects densely connected regions in large protein-protein interaction networks that may represent molecular complexes. Recent papers published in Science and Nature, among others, describe large-scale proteomics experiments that have generated large data sets of protein-protein interactions and molecular complexes [57, 127]. Protein structure [38] and gene expression data [90] is also accumulating at a rapid rate. Bioinformatics systems for storage, management, visualization and analysis of this new wealth of data must keep pace.

Currently, most proteomics data is available for the model organism Saccharomyces cerevisiae, by virtue of the availability of a defined and relatively stable proteome, full genome clone libraries [136], established molecular biology experimental techniques and an assortment of well designed genomics databases [37, 97, 48]. Predicting molecular complexes from protein interaction data is important because it provides another level of functional annotation above other guilt-by-association methods. Since sub-units of a molecular complex generally function towards the same biological goal, prediction of an unknown protein as part of a complex also allows increased confidence in the annotation of that protein.

Algorithms for finding clusters, or locally dense regions, of a graph are an ongoing research topic in computer science and are often based on network flow/minimum cut theory [59, 72] or more recently, spectral clustering [102]. We present a simple

scheme to find such regions. In a protein network, the nodes of the graph (protein) are the objects of the metric space where the metric is the shortest path distance. In our discussion we will assume that the protein network will be a connected graph.

The Antipole algorithm, given the radius σ, starts to create clusters of proteins in which nodes inside will be at distances no greater than σ. Now, executing the Antipole algorithm again with different radii we can obtain clusters that are bigger or smaller according to σ. In this way we will be able to understand how far apart the proteins are and if there is any relationship between them.

We have just described the case which occurs when one considers only one kind of edges, where there are only hops, not weights; if x is the distance between two nodes a and b, it means that x edges exist to go from a to b following the shortest path. Of course, in a protein network, each protein may have several kinds of edges. In this situation we simply weight the edges according to their importance and the Antipole will be built by considering the weights of the edges rather than the hops. For example if nodes a and b have three different edges T_1, T_2 and T_3, the edges will be removed and replaced by a new edge with weight $\sum_{i=0}^{3} w(T_i)$. Now, if a and b are x distance apart, it means that a shortest path of weight x from a to b exists.

This work is still in progress with NYU bioresearchers, and no report is ready yet; however, thanks to the speed of the Antipole clustering building, interesting results have already shown the possibility of predicting important proteins functions.

5.2 Graphs Clustering

In the last few years, developing algorithms for clustering data represented by graphs has been recognized as a problem in the pattern recognition community [24]. Nevertheless, graph clustering is still an open problem for two reasons. First, exact graph matching problems, i.e. subgraph isomorphism, maximum common subgraph, etc., are NP-complete. So exact graph clustering algorithms using graph matching are

extremely time consuming. Second, the proper distance metric between graphs is a matter of debate.

Spectral methods try to represent the most interesting properties of the input graphs using vectors, thus reducing the graphs clustering problem to a problem in a vector space [18, 19, 91]. In this paper, a new *spectral* method taking advantage of text retrieval concepts is presented.

Text retrieval (see [94, 51, 13, 92]) has focussed on the need to locate textual information efficiently. Widely-researched text searching method (see [51]) involves modelling a text collection in document-term matrix, and evaluating a document's relevance to a query using a linear algebraic dot product. In a term-document matrix A, $A[i, j]$ gives the number of occurrences of term j in document i. Queries are normally represented as a bit vector over the same set of terms. The similarity between document vectors (the rows of document-term matrices) can be found by their inner product. This corresponds to determining the number of term matches (weighted by frequency) in the respective documents. Another commonly used similarity measure is the cosine of the angle between the document vectors. This can be achieved computationally by first normalizing (to 1) the rows of the document-term matrices before computing inner products. Singular Value Decomposition (SVD) has been shown to work well for text retrieval in several recent works [52, 84]. Singular Value Decomposition achieves rank reduction as follows.

Large document-by-term matrices have a significant amount of redundant data. Removing this information allows a more precise and efficient search. However, given that in this paper we will only show a tecnique to map graphs in a k-dimensional space, without performing any kind of search, the Singular Value Decompositon, will not be used. Latent Semantic Indexing (LSI, [85]) attempts to project term and document vectors into a lower dimensional space spanned by the true "factors" of the collection. This uses a truncated Singular Value Decomposition (SVD) of the

term-document matrix.

Subdue is another method to capture essential structure information from graphs. The Subdue substructure discovery system ([2]) discovers repetitive subgraphs in a labeled graph representation by using the minimum description length principle. Experiments show Subdue's applicability to several domains, such as molecular biology, image analysis and computer-aided design.

In this section, GraphClust, a new algorithm for clustering labeled graphs, will be presented. The problem of mapping the graphs as feature vectors is solved by creating some substructures such that the frequency of substructure j in the graph i is stored at $A[i, j]$. After this, the rows of the matrix A are finally clustered.

5.2.1 Design

GraphClust assumes that the nodes of the database graphs have an identification number and a label. Edges are unlabeled (for purpose of this paper).

GraphClust deals with either directed or undirected graphs. The substructures can be discovered in two ways:

- by using the AllPairShortestPath algorithm; in this case, for each graph of the dataset and for each vertex v, all the shortest paths of length 1 up to a small constant l_p are generated from v.
- by using the Subdue substructure discovery system ([2]); in this case, for each graph g of the dataset, Subdue finds common or approximately common substructures of g.

A matrix having a number of columns equal to the number of the found substructures and a number of rows equal to the number of the graphs in the dataset is created. Each entry $A[i, j]$ equals the number of times in which the substructure j is contained in the graph i.

```
                    Basic Algorithm

GRAPHCLUST_BASIC(DataBase)
       - - - Start first step - - -
   1   Creates substructures of the data graphs;
       - - - End first step - - -

       - - - Start second step - - -
   2   Creates a matrix A having as number of rows, the number
       of data graphs, and as number of columns, the number of
       substructures;
   3   for each graph i
   4       Fills the entry A[i, j] with the number of occurrences
           of substructure j in the graph i;
   5   end for each;
       - - - End second step - - -

       - - - Start third step - - -
   6   Clusters the rows of A;
       - - - End third step - - -
   7   end GRAPHCLUST_BASIC.
```

Figure 5.1: GraphClust. the three steps of the basic algorithm.

Subdue is more suitable when the graphs in the database have few labels compared to the number of nodes. In that case, there is a high chance of common substructures. If AllPairShortestPath is run, it finds for each graph all the paths of length 1 up to a small constant l_p and therefore it creates more columns in the matrix A than Subdue. For example, on chemical compounds, where usually there are not so many nodes and edges, Subdue provides a better solution. For time-critical applications or graphs with many edges, AllPairShortestPath should be used because it takes less time.

Once the matrix A is built, there are two possible clustering algorithms to use: one is the k-means algorithm in which the user chooses the number of clusters k to create; the other is the Antipole Clustering [30] in which the user chooses a "tightness" measure (an integer value in the range 1 to 4) where the higher the measure the smaller the cluster radius and hence the larger the number of generated clusters. Antipole clustering is much faster than k-means even if it is not possible to know a-priori the number of clusters that will be created. The metric distance used in both clustering algorithms just described can be either the Euclidean distance or the inner product. The Euclidean distance has an intuitive appeal as it is useful for evaluating the similarity of objects in a multidimensional space.

The three steps of the basic GraphClust algorithm are shown in Fig. 5.1. In table 6.2, a matrix obtained from the patterns generated by applying the AllPair-ShortestPath algorithm, with $l_p = 3$ to the dataset in Fig. 6.1, is shown.

5.2.2 Algorithms

It turns out that GraphClust consists of 16 different algorithms broken down along the four binary dimensions described in the section 5.2.1. The main concept of Graph-Clust is the mapping of the data graphs into k-dimensional vectors. To perform this step we have introduced the concept of substructures and the methods used to find these substructures.

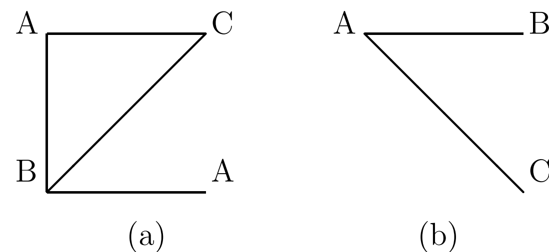


Figure 5.2: Dataset of two graphs.

Graph (a)		Graph (b)	
Initial Node	Substructures generated	Initial Node	Substructures generated
top-left A	{A,AC,AB,ABA}	A	{A,AB,AC}
B	{B,BC,BA,BA}	B	{B,BA,BAC}
bottom-right A	{A,AB,ABC,ABA}	C	{C,CA,CAB}
C	{C,CA,CB,CBA}		

Table 5.1: Patterns generated from the dataset of Fig. 6.1 using AllPairShortestPath with $l_p = 3$.

	A	AC	AB	ABA	B	BC	ABC	BAC
graph (a)	2	2	4	2	1	2	2	0
graph (b)	1	2	2	0	1	0	0	2

Table 5.2: Matrix generated from the pattern of Table 6.1.

In this section, the algorithms used by GraphClust in the three steps of its main procedure will be discussed.

Subdue discovers interesting and repetitive subgraphs in a labeled graph representation using the minimum description length principle; Subdue discovers substructures that compress the original data and represent structural concepts in the data. By replacing previously-discovered substructures in the data, multiple passes of Subdue produce a hierarchical description of the structural regularities in the data. Subdue uses a computationally-bounded inexact graph match that identifies similar, but not identical, instances of a substructure and finds an approximate measure of closeness of two substructures when under computational constraints. In addition to the minimum description length principle, other background knowledge can be used by Subdue to guide the search towards more appropriate substructures. Once the substructures and the matrix A have been created, the clustering is performed by the k-means or Antipole clustering method. In our implementation of k-means, k centroids are computed by using the Gonzalez (see [74]) algorithm, and, then, the remaining vectors are assigned to the closest centroid.

As already seen in 4 the randomized algorithms used by Antipole clustering makes its construction much faster than k-means's.

5.2.3 Complexity

Here is a description of the worst case complexity for the three steps of GraphClust. Let $|D|$ the number of graphs in a database D. The first and the second steps of the algorithm depend on which algorithm is used to create the patterns. If AllPairShortestPath is used, then the complexity of the first step is $\mathcal{O}(\sum_i^{|D|}({V_i}^3))$, where V_i are the nodes of the graph i; in this case the complexity of the second step is $\mathcal{O}(|D||words|\sum_i^{|D|}(n_i m_i^{l_p}))$, where $|words|$ is the number of patterns generated. If the Subdue algorithm is used, then the complexity of the first step becomes

$\mathcal{O}(\sum_i^{|D|}(\sum_{j=1}^{nsubs}(ninst_j \times gm_j)))$, where $ninst_i$ is the maximum possible number of non-overlapping instances for substructure j and gm_j is the user-defined maximum number of partial mappings that are considered during a graph match between substructure definition j and a potential instance of the substructure. In this case the complexity of the second step is $\mathcal{O}\ (|D||words|\sum_i^{|D|}(\sum_{j=1}^{nsubs}(ninst_j \times gm_j)))$, where $ninst_i$ and gm_j have already been described above. Details of the Subdue complexity analysis can be found in [111].

The third step depends on the clustering algorithm used. K-means takes time $\mathcal{O}(tkn)$, where n is the number of objects, k is the number of clusters, and t is the number of iterations. Normally, $k, t \leq n$. The Antipole algorithm has a worst-case complexity of $\frac{\tau(\tau-1)}{2}n + o(n)$ in the input size n, where τ is the bounded radius (see [30] for further details).

Hence, given the higher complexity of Subdue, GraphClust by using Subdue for finding the substructures should be used with small datasets having graphs with not so many edges. In this case it is highly probable that Subdue is able to find at least a meaningful substructure for each graph and, thus, the quality of the clustering will be of higher quality. As mentioned above, Subdue is more suitable than AllPairShortestPath when GraphClust is used to cluster a dataset of chemical compounds.

Conversely, when GraphClust is used to cluster big datasets having large numbers of edges, the speed of AllPairShortestPath makes it preferable to Subdue.

5.2.4 Performance Studies

GraphClust is written in C language and it is freely downloadable to [112]. In this section, a systematic evaluation of the quality of the clusters will be discussed. The Silhouette method [20] is used to show how good the clustering obtained is. Moreover, we want to show how well the final clustering captures the graphs present in different categories known *a priori*. For this, we have used an artificial graph benchmark to

create a database containing five different categories of undirected graphs. The five categories includes randomly graphs, regular 2D-meshes, regular 3D-meshes, irregular 2D-meshes, irregular 3D-meshes. For each category we have generated 1000 graphs with 30 nodes and 1000 graphs with 80 nodes. The number of edges vary from 48 to 189. Each group of 1000 graphs differs for the 20% of the edges. Thus, our artificial dataset contains 10000 graphs. An optimal clustering creates 10 clusters, each one containing one structural group of graphs. Tables 5.3 and 5.4 depict the global silhouette values, GSu, for each partition, and the silhouette values, S_i, for each number of clusters c, for $c = 10$ to 15. Clustering in table 5.3 is based on GraphClust with the Antipole Tree data structure whereas clustering in table 5.4 uses GraphClust with the k-means algorithm. For both clustering algorithms, the substructures have been discovered by using the AllPairShortestPath fixing the constant $l_p = 3$. In both tables $c = 10$ is suggested as the best clustering configuration for the examined data set and this is also the optimal number of clusters known for when the data set has been created.

c	GSu	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}	S_{15}
10	0.712	1	0.86	1	0.82	0.05	0.71	1	0.88	0.12	0.66					
11	0.665	1	0.86	1	0.82	0.03	0.81	0.11	1	0.88	0.12	0.66				
12	0.693	1	0.86	1	0.82	0.03	0.81	0.11	1	1	0.88	0.12	0.66			
13	0.607	1	0.86	1	0.82	0.03	0.81	0.11	1	1	0.21	0.24	0.12	0.66		
14	0.518	1	0.12	0.10	1	0.82	0.03	0.81	0.11	1	1	0.21	0.24	0.12	0.66	
15	0.489	1	0.05	0.13	0.12	1	0.82	0.03	0.81	0.11	1	1	0.21	0.24	0.12	0.66

Table 5.3: Global Silhouette values for clustering obtained by using GraphClust with Antipole Tree data structure.

Now, we have to show that this clustering is also coherent with the *a priori* classification of the data set. Recall that in the optimal clustering each cluster contains a single structural group where in each group the graphs differ by 20% of their edges. Table 5.5 shows the similarity in percent between the best clustering obtained in table 5.3 and 5.4 for $c = 10$ and the optimal clustering. A value $x\%$ for the cluster

c	GSu	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}	S_{15}
10	0.886	1	1	0.66	0.88	1	0.86	0.82	0.81	0.81	1					
11	0.830	1	1	0.66	0.89	1	0.86	0.82	0.81	0.81	1	0.25				
12	0.714	1	1	0.66	0.20	1	0.87	0.82	0.81	0.81	1	0.25	0.12			
13	0.678	1	1	0.66	0.20	1	0.11	0.82	0.81	0.81	1	0.25	0.12	1		
14	0.643	1	1	0.66	0.20	1	0.14	0.82	0.81	0.81	1	0.25	0.16	1	0.10	
15	0.660	1	1	0.66	0.20	1	0.09	0.82	0.81	0.81	1	0.25	0.11	1	0.10	1

Table 5.4: Global Silhouette values for clustering obtained by using GraphClust with k-means algorithm.

C_i obtained with GraphClust means that C_i is equal to $x\%$ of the optimal cluster C_i. To measure the robustness of the clustering obtained, a pair of graphs g_1 and g_2 are considered to be consistent in the two clusterings if they are in the same cluster in both cases or in different clusters in both cases. Otherwise they are inconsistent. In table 5.6, for $c = 10$, the output clustering generated by GraphClust has been compared with the optimal clustering. The value *consistent_value* shows the number of graphs pairs that are consistent divided by the total number of graphs pairs for the output obtained.

Clustering Algorithm	C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}
Antipole Tree	100%	100%	100%	100%	66.0%	95.14%	100%	100%	28.9%	50%
K-means	100%	54.4%	50%	100%	100%	100%	100%	100%	100%	45.6%

Table 5.5: Similarity in percentual between the best clustering found ($c = 10$) in Tables 5.3, 5.4 and the optimal clustering.

Clustering Algorithm	Number of consistent pairs	Total number of pairs	*consistent_value*
Antipole Tree	48704861	49995000	0.97
K-means	48746936	49995000	0.97

Table 5.6: Robustness between the best clustering found in Tables 5.3, 5.4 and the optimal clustering.

5.3 Texture synthesis

The exciting world of "texture", with its different applications and results has been for years a challenging research area [12]. Texture classification, discrimination, retrieval, mapping and/or rendering represent only a partial view if the various lines of research and application fields. Among others, to be able to realize fast and effective algorithm for texture synthesis, with high performance both in term of real time generation and perceived quality is a fascinating goal. Two different strategies or line of research have been followed in the literature. The more ambitious one tries to "learn", by using properly filtering, the underlying stochastic model (e.g. Markov Random Fields [50]) of an input texture; the synthesis is then obtained by a suitable sampling (see Fig. 5.3). Main drawbacks of these methodologies are related with the computation time that

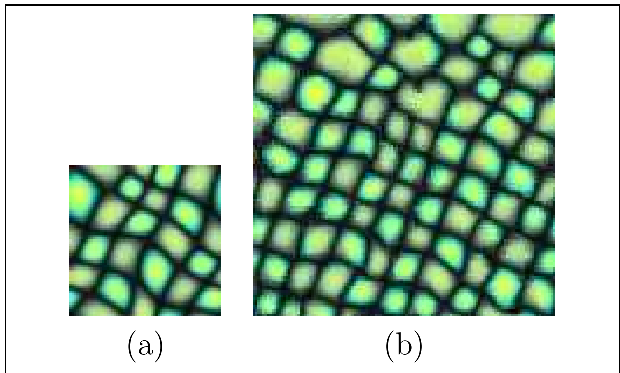


Figure 5.3: A synthesis example.

tends to be impractical for real-time applications ([137, 144, 143]). More efficient techniques tend to properly match texture features ([107]), measured at different resolution levels (see [26, 105]). In some sense a series of heuristics are used without explicitly derive a real mathematical model.
In [78, 21] impressive results using respectively marginal histograms of image pyramids and maintaining cross-scale dependencies were obtained (see also [Bat00a], [Bat01b]).

More recently [54] and [132] pointed out to a series of simple but effective techniques showing excellent results on large class of textures. In particular the work presented in [Wey00] has been furtherly generalized in [79] to realize a computational framework where analogies between pairs of images can be deduced. Other techniques such as those presented in [138] and [139] combine together smart patch merging. This section describes a series of possible solutions trying to improve existing algorithmic solutions by making use of advanced approximated search data structures. The procedural approach described in [132] applies a multiresolution technique tracking neighborhood dependance level by level. The synthesis is realized using a classical sampling strategy over the data collected in the analysis phase. The entire process is then accelerated using the TSVQ (Tree Structure Vector Quantization) [66] which introduces some approximation but speeds-up the overall process. The TSVQ considers the input neighborhoods as vectors in a multi-dimensional space in order to replace them by a codebook of few vectors. The goal is to find the nearest neighborhood from the current neighborhood as fast as possible. The time is reduced by more than one order of magnitude. As a drawback such approximation introduces, in some cases, undesired artifacts. We claim that the overall computation time needed to perform a full-search sampling strategy can be avoided using suitable advanced data structures and searching strategies. In our approach, image pixels are grouped into clusters of bounded radius by the Antipole Tree Clustering and the synthesis is performed by using the nearest neighbor algorithm (see chapter 4 for the description of the data structure).

The clustering probability model of spatial neighborhoods derived from a texture was introduced for the first time by [106].

5.3.1 Texture synthesis via Antipole Tree Clustering

The Wei-Levoy algorithm [132] uses the locality and stationarity properties of the textures to synthesize an image by a raster scan order. Each pixel in the input sample is mapped into the pixel of the nearest neighborhood vector. The input consists of an example texture patch (Fig. 5.3 left side picture) together with a random noise image having the desired size of the output image. The algorithm modifies this random noise to make it looks like the given example. The algorithms consists in a raster scan order synthesis of the pixels. The process starts from the upper left side corner pixel and proceeds from left to right line after line. The picture in Fig. 5.4 shows an example of such a synthesis process.

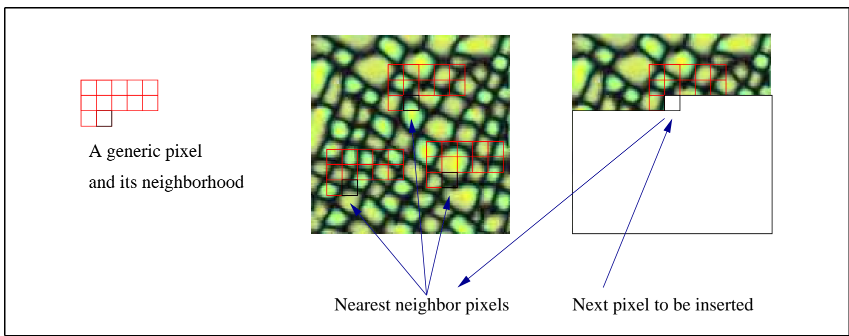


Figure 5.4: The right scan order synthesis method.

This technique is flexible and easy to use, since only an example texture patch is required. Since this process is quite computationally expensive, multi-resolution pyramids and quantization acceleration are used (see [7, 15, 131]). In [132, 131] a speed-up is obtained using TSVQ [66] which takes as input a set of training vectors and generates a binary tree of codebooks having a depth specified by he user, which will be representative of the dataset. First the process finds a centroid c of the training vector and uses it as root of the tree. After that, the same centroid c and a properly

perturbated centroid are chosen as children of the root. The process proceeds recursively until the specified depth is reached. The approximation introduced considers a variable number of training codebooks allowing also a limited backtracking in the tree traversal to trade-off between computation time and final image quality.

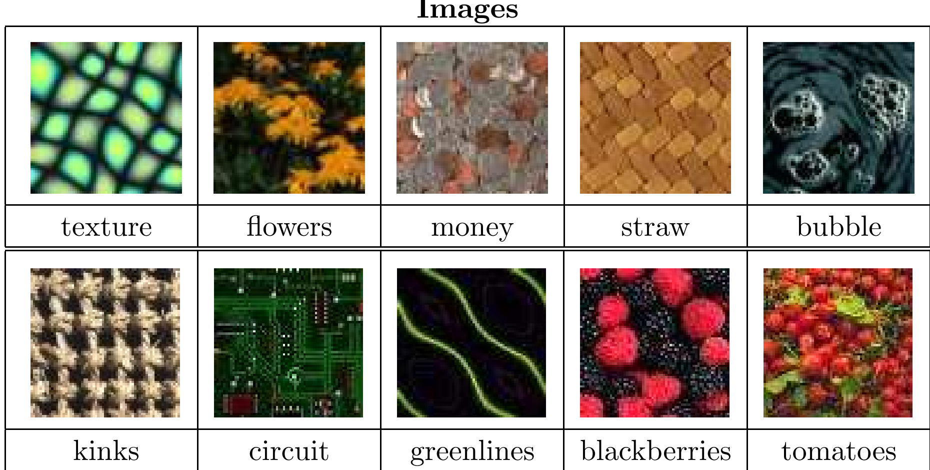
In our approach the synthesis process is performed by using the Antipole Tree, as reported in [10]. The positive cluster radius σ, is used to guarantee that pixels with a similar neighborhood lie in the same cluster. Once the Antipole Tree is built the nearest neighbor procedure is used to perform the search needed to synthesis the image.

The results obtained with the proposed Antipole Clustering were compared with the work of [132]. The notation $\{R_1 \times C_1, 1\}, \cdots, \{R_i \times C_i, k_j\}, \cdots, \{R_m \times \{C_m, k_n\}$ indicates multi-resolution n levels each with neighbor size $R_i \times C_i$ at the top level merged with the previous $k_j - 1$ levels each one having neighborhood size $R_i - 2 \times C_i - 2, \cdots, R_i - 2 * (k_j - 1) \times C_i - 2 * (k_j - 1)$. For example the expression $\{7 \times 7, 1\}\{9 \times 9, 2\}\{11 \times 11, 3\}$ means: synthesize 3 levels multi-resolution with the first level neighbor size 7×7, the second level neighbor 9×9 merged with the previous level with neighbor size 7×7 and the third level 11×11 merged with 9×9 and 7×7. The other term of comparison of our approach with respect to the classical full-search strategy is based on timing.

We implemented our algorithm in standard C (GNU gcc compiler v2.96) and all experiments were carried out on a PC Pentium III 900Mhz with Linux Operating System (Mandrake 8.2), while the input texture database used was the VisTex [1]. Each time as reported in [132] is referred to the synthesis process obtained starting from a random equalized noise image. Furthermore to stress the synthesis process the size of the output image is always two times the size of the input image. The timing comparison has been realized, using different neighborhood size (e.g. 3×3, 5×5, 7×7, ...), single level and/or multi-resolution. In all cases the computation time of the

Antipole strategy was better than the classical full search.

Table 5.7 reports some results, showing the average time and the corresponding percentage gain obtained over several textures. One crucial point of the proposed method is to find a suitable clustering radius to perform a fast synthesis. The results presented in Table 5.7 show that the Antipole Clustering running time over different textures with same levels and neighborhood size may be different. This means that the underlying vector space distribution generated during the synthesis process affects the performance of the proposed method. The running time of the Antipole Tree reported in Table 5.7 includes the building time of the tree which takes only few seconds with respect to the overall synthesis process. The TSVQ (Tree Structure Vector Quantization) acceleration used by [132], whose details are better discussed in [131], is able to run two orders of magnitude faster. It works well over a large data set of texture [1], but introduces some approximation. As shown in [7], the texture that are composed of various small objects (and many edges) do not give output images of good quality as in the case of most natural textures (e.g. leaves, flowers, etc.). In many cases only the full neighborhood search guarantees satisfactory results. But Fig. 5.8 shows a series of examples and comparisons where TSVQ acceleration produces noticeable artifacts while our proposed method provides satisfactory results. For each experiment the TSVQ tree is constructed using the maximum number of codewords and the search is implemented as suggested in [132]. Fig. 5.8 shows that our proposed acceleration technique seems to be more robust. This can be associated with the experiments done in chapter 3 section 4.7 were the approximated search with the Antipole Tree returns objets nearer to the exact nearest neighbor with respect to the TSVQ. Furthermore the time of the approximate Antipole search are fully comparable to the one of the TSVQ.



Images	Neighbor	Full	Antipole	%
texture	$\{5 \times 5, 1\}$	728	44	93,96
	$\{7 \times 7, 1\}$	1386	69	95,02
	$\{5 \times 5, 1\}\{5 \times 5, 2\}$	1260	84	93,33
	$\{7 \times 7, 1\}\{7 \times 7, 2\}$	2580	206	92,02
flowers	$\{3 \times 3, 1\}$	221	38	82,81
	$\{9 \times 9, 1\}$	2820	427	84,86
	$\{5 \times 5, 1\}\{5 \times 5, 2\}$	1260	205	83,73
	$\{5 \times 5, 1\}\{7 \times 7, 2\}$	2520	428	83,02
money	$\{5 \times 5, 1\}$	728	179	75,41
	$\{3 \times 3, 1\}\{5 \times 5, 2\}$	1221	374	69,37
straw	$\{9 \times 9, 1\}$	2820	965	65,78
bubble	$\{9 \times 9, 1\}$	2820	783	72,23
	$\{13 \times 13, 1\}$	4020	2340	41,79
	$\{7 \times 7, 1\}\{9 \times 9, 2\}$	3900	2038	47,74
kinks	$\{9 \times 9, 1\}\{9 \times 9, 2\}\{9 \times 9, 3\}$	4800	1001	79,14
	$\{9 \times 9, 1\}\{9 \times 9, 2\}$	4200	829	80,26
	$\{7 \times 7, 1\}$	1386	264	80,95
circuit	$\{5 \times 5, 1\}\{5 \times 5, 2\}$	1573	480	69,48
	$\{9 \times 9, 1\}\{9 \times 9, 2\}\{9 \times 9, 3\}$	5400	1155	78,61
greenlines	$\{7 \times 7, 1\}\{7 \times 7, 2\}$	3120	446	85,70
	$\{7 \times 7, 1\}\{7 \times 7, 2\}\{7 \times 7, 3\}$	3480	381	89,05
	$\{9 \times 9, 1\}$	3240	365	88,73
blackberries	$\{7 \times 7, 1\}$	1724	302	82,48
	$\{9 \times 9, 1\}\{9 \times 9, 2\}$	4897	907	81,47
	$\{13 \times 13, 1\}\{13 \times 13, 2\}$	8086	1407	82,59
tomatoes	$\{7 \times 7, 1\}$	1723	370	78,52
	$\{9 \times 9, 1\}\{9 \times 9, 2\}$	4896	2651	45,83

Table 5.7: Running time comparison between full search and Antipole data structure. The second column describes the size of each neighbor level by level. The third and fourth columns show the running time (in seconds) needed by the full search and the Antipole search.

Input image	Antipole	Approx Antipole	TSVQ

Table 5.8: A comparison between the Antipole exact search (second column), approximate Antipole search (third column), TSVQ search (last column). The examples show, for complex textures, that the synthesis using the approximated Antipole search is able to capture more details than TSVQ.

5.4 Image Colorization

The general problem of inverting a gray palette to a color palette is an undetermined problem and generally has no unique solution. For this reason to accomplish this task, for example in restoration of old photos, the (costly!) semantic knowledge of an expert is required. In this section a semi-automatic method to minimize the amount of human work required for this task is proposed. The early published methods to perform the image colorization rely on heuristic techniques for choosing RGB colors from a palette and applying them to regions of the target gray-scaled image. The proposed method, instead, transfers the color from a source image to a target image by matching luminance information between the images. This approach, inspired to a recently published algorithm by Welsh et al [135], hence inscribes itself among the similarity based image enhancing techniques [80]. With the adoption of the Antipole data structure we fastly retrieve color words from a very large vocabulary (see [17]).

One wishes to add colors to a gray-scaled image for many reasons: colors increase the visual appeal of an image such as an old black and white photo; they make an old movie nicer, and help to make a scientific illustration more attractive (example: a scanning electron microscopy image, SEM). The problem of colorization of a gray-scaled image involves assigning three-dimensional (RGB) pixel values to an image whose elements (pixels) are characterized only by one feature (luminance). Since different colors may carry the same luminance in spite of differences in hue and/or saturation, the problem of gray-scaled images colorization has no inherently correct solution. Due to these ambiguities, human interaction usually plays a large role in the colorization process [121]. Even in the case of pseudo-colorization [73], where the mapping of luminance values to color values is automatic, the choice of the colormap is commonly determined by human decision. Detailed technical documents describing the colorization process are generally not publicly available because of the economical

relevance of such applications for the movie industries. There also exist a number of applications for the use of color in information visualization. Further, color can be added to a range of scientific images for illustrative and educational purposes. Our concept of transferring color from one image to another is inspired by work of Welsh et al. [135]. In their work, colors from a source image are transferred to a target gray-scaled image using a simple procedure. Their basic method matches the one-dimensional distribution of luminance values between the images and then transfers the other components from the source image to the target image. To perform the matching they use the pixel luminance and the standard deviation of the luminance in a pixel neighborhood with size of 5x5 pixels. The matching is performed on a sample set of pixels of the source image using a sequential search. We claim that the overall computation time needed to perform a full-search sampling strategy can be avoided using a suitable advanced data structure and a more refined searching strategy. To use the Antipole Data Structure we map the hole space of pixels as the set X and the normal Euclidean distance between pixels as the metric function d. In our approach, as detailed in [17], image pixels are grouped into clusters of bounded radius by the Antipole Tree Clustering. In this section we report and discuss three different variations over the base idea reported above: in a first approach (named in the following LPN) we use the luminance values of the pixel neighborhood (using a 5x5 size), in a second approach (named in the following L&S) we use the pixel luminance and the standard deviation of the pixel neighborhood (using a 5x5 size), and in the last approach (named in the following UnA) we unify the two previous methods. Observe that the L&S technique is essentially the same proposed by [135], but our implementation of it takes advantage of the Antipole Tree. Experimental results show the improvement in term of computation time with respect to full-search strategy while maintaining the same final quality.

5.4.1 Image Colorization using Antipole Tree Clustering

In this section, we describe the algorithm for transferring color, as described in [17]. The general procedure for color transfer requires a few simple steps. First RGB source image is converted into the YUV color space. This color space has been chosen because it promptly provides the luminance value (channel Y) which is a crucial datum for our procedure. It also grants a more faithful modeling of human perception. Next the Antipole tree is constructed, each vector contains the information necessary to perform the Antipole search and the UV components of the pixel color (see chapter 4 for details). After the data structure has been completed, in scan-line order, for each pixel in the gray-scaled image we construct its vector and perform the Antipole search to select the best matching vector in the Antipole tree. The UV components of the best matching vector are then transferred to the gray-scaled image to form the final image, while the Y component (luminance) of the pixel in the gray-scaled image is retained to its original value. Although this procedure is very simple and direct the experimental results show that it works very well on a large set of images. Even if at this stage of research we focused on homogeneous images it is likely to imagine that the algorithm will also work well on nonhomogeneous (segmented) images.

This section reports all the experimental results obtained with the proposed method. Performances of the proposed approach with respect to [135] work are compared. Figures 5.11 show some examples of colored images obtained with the Antipole strategy. Table 5.9 reports the timing results and the corresponding percentage gain obtained. The running time of the Antipole Tree reported in Table 5.9 includes the building time of the tree which takes only few seconds with respect to the overall synthesis process. Table 5.10 reports the Peak Signal Noise Ratio (PSNR) obtained with the three different methods. The PSNR is defined as:

$$PSNR = -10 \times log(\frac{MSE}{S^2})$$

where

$$MSE = \frac{1}{nRow \times nCol} \times \sum_{x,y} [I(x,y) - J(x,y)]^2$$

and S=255, $nRow$ is the row number, $nCol$ is the column number, $I(x,y)$ is the pixel value of the target image and $J(x,y)$ is the pixel value of the recolored image. Firstly, we compute the PSNR on the tree (RGB) channels and, secondly, we evaluate the mean value. As reported in the previous section the Antipole strategy speeds-up the process without quality loss. We implemented our algorithm in standard C (GNU gcc compiler v3.2) and all experiments were carried out on a PC dual Athlon XP 2000+, 1 GB RAM, with Linux Operating System (Red Hat 8.0). In all cases the computation time of the Antipole strategy was better than the classical full search.

IMAGES		LPN		L&S		UnA	
	Time	229s	44s	27s	11s	235s	64s
	Gain	81%		59%		73%	
	Time	157s	49s	18s	9s	160s	69s
	Gain	69%		50%		57%	
	Time	766s	144s	90s	39s	788s	159s
	Gain	81%		57%		80%	
	Time	739s	130s	86s	38s	762s	146s
	Gain	82%		56%		81%	
	Time	3900s	724s	448s	194s	3942s	796s
	Gain	81%		57%		80%	
	Time	953s	93s	98s	48s	978s	105s
	Gain	90%		51%		89%	
	Time	2700s	542s	280s	133s	2980s	627s
	Gain	80%		53%		79%	
	Time	951s	149s	112s	48s	982s	205s
	Gain	84%		57%		79%	
	Time	946s	153s	111s	48s	974s	190s
	Gain	84%		57%		80%	

Table 5.9: Running time comparison (expressed in seconds) between full search and Antipole search.

IMAGES	LPN	L&S	UnA
	38,31	36,45	38,27
	25,85	24,69	26,20
	21,32	21,45	21,42
	26,98	25,21	27,42
	27,60	22,66	27,61
	29,15	24,06	28,25
	32,69	30,15	31,42
	26,18	25,07	25,68
	27,19	27,44	27,70
Mean Value	28,36	26,35	28,22

Table 5.10: PNSR (mean on the RGB channels) comparison between the LPN, L&S and UnA methods described above using the Antipole search, the last row shows the mean values obtained.

Source Image	Target Image	Recolored Image	Original Colored Image

Table 5.11: Some examples of colored images obtained with the Antipole strategy.

Part II

Graph Searching Based on Indexing Techniques

Chapter 6

GraphGrepVF: a new efficient method for exact and inexact graph matching

Graphs are data structures widely used for representing information both in low-level and high level vision tasks. One of the problems of interest, with graphs, is matching a sample graph against a reference graph. Depending on the nature of the considered vision task and on the characteristics of the graphs, either exact or inexact matching may be required. In both cases, different types of morphism are possible for the matching: we will mainly consider graph isomorphism.

A relevant problem when matching graphs is that of limiting the computational cost of the process. Purpose of this second part of the thesis is to illustrate a novel graph matching algorithm and to compare its performance with two of the most commonly used algorithms performing the same task by taking into account the problem of reducing the matching time. The algorithms considered for comparison are GraphGrep [71, 68, 117] and VF [46]. For a detailed description of GraphGrep see [68].

In real pattern recognition applications, the variability of the samples is such that they are seldom identical to prototypes so that pattern recognition can only be

achieved by inexact graph matching methods.

6.1 Introduction

The increasing size of application databases requires efficient structure searching algorithms [68]. Examples of such database and substructure searching methods can be found in computational chemistry [88, 130], vision [53], and web exchange data (XML) [95, 42]. Finding occurrences of a subgraph in a set of graphs is known to be NP complete [65]. Although graph-to-graph matching algorithms [44, 129] can be used, efficiency considerations suggest the use of special techniques to reduce the search space and the time complexity. There is an extensive literature on graph (or substructure) searching. For a review see [130, 123, 117]. Most of the existing methods however, are designed for specific applications. For example several querying methods for semistructured databases, and in particular for XML databases, have been proposed ([95, 42, 120, 63, 123, 114]). These methods use different data models, query languages and indexing strategies. The data objects used in XML databases are viewed as rooted labeled graphs. Regular path expressions are used to address substructures in the database. Cycles are searched by evaluating recursion functions or by formulating complex queries. To avoid unnecessary traversals of the database during the evaluation of a path expression, indexing methods are introduced in [95] and [98]. Daylight [88] proposes a searching system for a database of molecular graphs. It finds all the molecules that contain, as a subgraph, at least one occurrence of the query. Daylight uses fingerprints consisting of bit vectors, where each position represents a small path. It also provides a graph expression language based on the SMILES [134] molecule representation to formulate queries. Frowns [16] is another chemoinformatics toolkit based on PyDaylight geared toward rapid development of chemistry related algorithms. Messmer and Bunke [96] propose an application independent method. The method indexes the graphs in a database and computes a graph

isomorphism. Both indexing and matching are based on all possible permutations of the adjacent matrices of the graphs. This algorithm works extremely well on small graphs, but does not scale well to larger graphs or large databases of graphs.

The quite few approaches to inexact matching proposed in literature, try to extend the applicability of exact matching methods, by introducing criteria allowing matching in presence of syntactic and/or semantic deformations. In [125], a pattern deformational model is proposed, while a generalization of the method, including the possibility of deleting nodes and branches, is discussed in [126]. The algorithm, though powerful enough for some practical applications, is not effective when large variations among the members of a same class may exist. In these cases inexact matching approaches based on the definition of a distance measure between graphs, seem more appropriate [115, 113]. An extension of an ARG matching algorithm which uses a set of feasibility rules and taking into account deformations on syntactic and semantic parts of the graphs is described in [44].

In this section we present an application-independent method to perform exact and inexact subgraph queries in a database of graphs. Our system, GraphGrepVF, finds all the occurrences of a graph in a database of graphs. GraphGrepVF is a merging of two powerful method for graph matching: GraphGrep [71] and VF [46] algorithms. Both GraphGrep and VF are available online [70, 3, 45]. In the section 6.4 the new method will be explained in detail.

To formulate queries we use a graph query language which we term GLIDE: Graph LInear DEscription language [68, 70]. GLIDE descends from two query languages Xpath [42] for XML documents and SMART [88] for molecules. In Xpath, queries are expressed using complex path expressions where the filter and the matching conditions are included in the notation of the nodes. GLIDE uses graph expressions instead of path expressions. SMILES is a language designed to code molecules and SMART is a query language to discover components in a SMILES databases. GLIDE borrows

the cycle notation from SMILES and generalizes it to any graph application.

In this chapter we will explain the two basic algorithms we have started with; in particular, GraphGrep [71] will be discussed in the section 6.2 while VF [46] will be discussed in section 6.3. Finally in section 6.4 the new method will be explained. In the two final sections, Graph LInear DEscription language (GLIDE) [68, 69], will be described together with the extension of GraphGrepVF to inexact matching.

6.2 GraphGrep

As reported in [71], GraphGrep is a general method to find all the occurrences of a query graph in a database of graphs. It is focused on undirected graphs whose edges are unlabeled but it generalizes to directed graphs with labeled edges. Due to the intractable complexity of the graph searching problem, GraphGrep is based on the idea to reduce the space of the possible matches. Its main algorithmic component is the storage of all paths up to a fixed length. These paths are used to perform the filtering and the matching. More precisely, GraphGrep filters out graphs of the dataset that do not contain the query graph. Moreover, for each candidate graph, it filters out the parts of the graph that do not contain the query. Once we have the set of candidate paths, the matching is performed by combining those. GraphGrep is divided into three basic step:

- Building the index to represent the database of graphs as sets of paths (this step is done only once).
- Filtering the database based on the submitted query and the index to reduce the search space.
- Performing the exact matching.

In subsections 6.2.1, 6.2.2 and 6.2.3 the three steps will be dealt and in subsection 6.2.5 the complexity analysys of the method will be presented.

6.2.1 Building the sets of paths

GraphGrep, as reported in [71], assumes that vertices of the data graphs have an identification number (*id-vertex*) and a label (*label-vertex*) (string of any length). An *id-path* of length n is a list of n id-vertices with an edge between any two consecutive vertices. A *label-path* of length n is a list of n label-vertices.

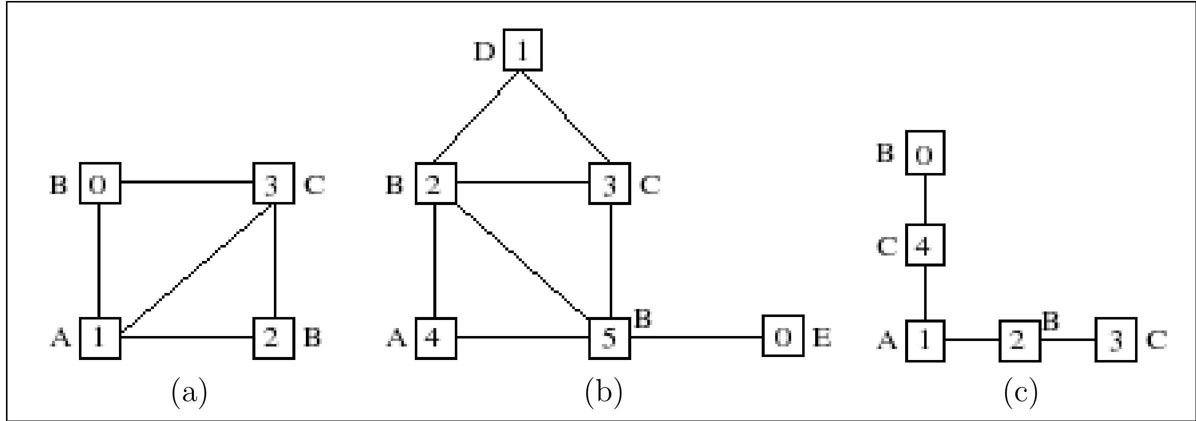


Figure 6.1: A database of three graphs.

In Fig. 6.1(taken from [71]), A-C is a label path of graph (a) and 1-3 is the id-path corresponding to it. The index of the database is constructed by using the label-paths and id-paths of the graphs in the following way: for each graph and for each vertex, we find all paths that start at this vertex and have length one (single vertex) up to a small fixed constant value (l_p). l_p is used as sample for all graphs of the database. Several paths may contain the same label sequence, so we group the id-paths associated with the same label-path in a hash table. The keys of this hash table are the hash values of the label paths. Each row contains the number of id-paths

associated with a key in each graph. We will refer to the hash table as the fingerprint of the database (see Table 6.2 (taken from [71])). In Table 6.1 (taken from [71]) there is a path representation of the graph in Fig. 6.1(a).

The query graph is decomposed in a set of intersection paths. The branches of a depth-first traversal tree of the query graph are decomposed into sequences of overlapping label-paths, called *patterns*, of length l_p or less. (see Fig. 6.2 (taken from [71])).

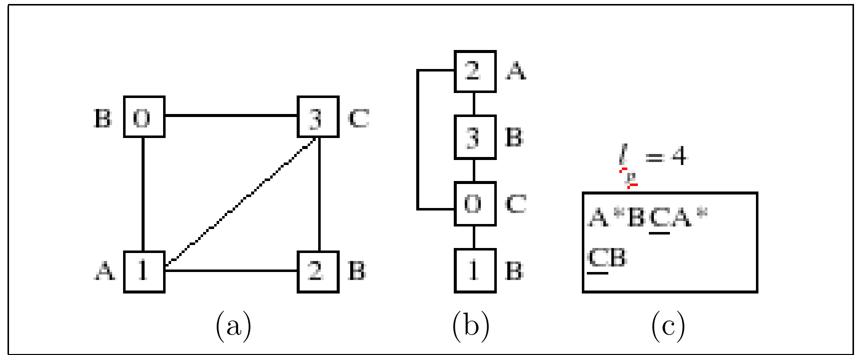


Figure 6.2: (a) A query graph. (b) The depth first tree of the graph in (a). (c) Patterns obtained with $l_p = 4$. Overlapping labels are marked with asterisks or underlining. Labels with same mark represent the same vertex

.

Overlaps may occur in the following cases:

- for consecutive label-paths, the last vertex of a pattern coincides with the first vertex of the next pattern (e.g. A/B/C/B/ with $l_p = 3$ is decomposed into two patterns: A/B/C/ and C/B/);
- if a vertex has branches, it is included in the first pattern of every branch (see vertex C in Fig. 6.2(c));
- the first vertex visited in a cycle appears twice: in the beginning of the first

Label-Path	Id-Path
A/	$\{(1)\}$
A/B/	$\{(1,0)(1,2)\}$
A/C/	$\{(1,3)\}$
A/B/C/	$\{(1,0,3)(1,2,3)\}$
A/C/B/	$\{(1,3,0)(1,3,2)\}$
A/B/C/A	$\{(1,0,3,1)(1,2,3,1)\}$
A/B/C/B/	$\{(1,0,3,2)(1,2,3,0)\}$
A/C/B/A/	$\{(1,3,0,1)(1,3,2,1)\}$
B/	$\{(0)(2)\}$
B/A/	$\{(0,1)(2,1)\}$
B/C/	$\{(0,3)(2,3)\}$
B/A/B/	$\{(0,1,2)(2,1,0)\}$
B/A/C/	$\{(0,1,3)(2,1,3)\}$
B/C/A/	$\{(0,3,1)(2,3,1)\}$
B/C/B/	$\{(0,3,2)(2,3,0)\}$
B/A/B/C/	$\{(0,1,2,3)(2,1,0,3)\}$
B/A/C/B/	$\{(0,1,3,0)(2,1,3,2)(2,1,3,0)(0,1,3,2)\}$
B/C/B/A/	$\{(0,3,2,1)(2,3,0,1)\}$
B/C/A/B/	$\{(0,3,1,0)(2,3,1,2)(2,3,1,0)(0,3,1,2)\}$
C/	$\{(3)\}$
C/B/	$\{(3,0)(3,2)\}$
C/A/	$\{(3,1)\}$
C/B/A/	$\{(3,0,1)(3,2,1)\}$
C/A/B/	$\{(3,1,0)(3,1,2)\}$
C/B/A/B/	$\{(3,0,1,2)(3,2,1,0)\}$
C/B/A/C/	$\{(3,0,1,3)(3,2,1,3)\}$
C/A/B/C/	$\{(3,1,0,3)(3,1,2,3)\}$

Table 6.1: Path representation of the graph in Fig. 6.1(a) with $l_p = 4$.

Key	Graph g_1	Graph g_2	Graph g_3
h(C/A/)	1	0	1
h(C/B/)	2	2	2
h(A/B/C/A/)	2	0	0
...			
h(A/B/C/B/)	2	2	0

Table 6.2: The fingerprint of the database.

pattern of the cycle and at the end of the last pattern of the cycle (the first and last pattern can be identical, as in Fig. 6.2(c)).

6.2.2 Filtering the database

This section shows the filtering method, as reported in [71]. The query graph is parsed to build its fingerprint (hashed set of paths). The fingerprint of the query is compared with the fingerprint of the database in order to filter the database. When the exact graph matching is performed, a graph for which at least one value in its fingerprint is less than the corresponding value in the fingerprint of the query is discarded. For example, with the query graph of Fig. 6.2 with $l_p = 4$ and the dataset of Fig. 6.1, the graphs (b) and (c) would be filtered out because they do not contain the label-path A/B/C/A/. The remaining graphs represent the candidates and they may contain one or more subgraphs matching the query. We continue filtering out parts inside of the candidate graphs in the following way: we decompose the query in patterns and only the parts of each candidate graph whose label-path sets correspond to the patterns of the query are selected and then compared with the query. The above method requires finding all the label-paths up to a length l_p starting from each node in the query graph.

Although this fingerprint produced a better filtering result, sometimes the its

size is too big while we have many graphs in the database. Moreover, the filtering construction running time is not linear to the number of graphs in the database. The file size of fingerprint was depending on two variables, the number of the graphs in the database and the number of rows in the hash table. For big complex database graphs, if we want to have better filtering results, the hash rows must be bigger enough to hold all possible patterns without collisions. The size of the hash table can be huge. Thus, another filtering tecnique, useful for such a kind of graphs, is the one used in Frowns and Daylight [16, 88]. In this method, we store a fingerprint as an fix-length array of integers (the default length is 256). Each integer has a certain number of bits (usually 32 in most platform) that can be flipped. The process of adding a path to a fingerprint is simply choosing the index and bit position for a path. The above procedure does this in a deterministic fashion. The advantage of this fingerprint is that each fingerprint is fix sized (1K bytes in this case), and we can produce and compare each fingerprint in procedure.

6.2.3 Matching

This section shows how to match a query graph against a graphs database taken from [71]. After filtering, we look for all matching subgraphs in the remaining graphs. We use the path representation of the graphs to look for occurrences of the query. Only the parts of each (candidate) graph whose id-path sets correspond to the patterns of the query are selected and compared with the query. After the id-path sets are selected, we identify overlapping id-path lists and concatenate them (removing overlaps) to build a matching subgraph. For overlapping cases (1) and (2) a pair of lists is combined if the two lists contain the same id-vertex in the overlapping position. In overlapping case (3), a list is removed if it does not contain the same id-vertex in the overlapping positions; finally, lists are removed if equal id-vertices are not found in overlapping positions.

Example. Let us consider the steps to match the query in Fig. 6.2(a) with the graph g_1 in Fig. 6.1(a).

1. Select the set of paths in g_1 (Fig. 6.1) matching the patterns of the query (Fig. 6.2(c)): A/B/C/A/= f{1, 0, 3, 1), (1, 2, 3, 1)} C/B/= {(3, 0), (3, 2)}.

2. Combine any list l_1 from A/B/C/A/ with any list l_2 of C/B/ if the third id-vertex in l_1 is equal to the first id-vertex of l_2 and the first id-vertex in l_1 is equal to the fourth id-vertex of l_1: A/B/C/A/C/B/= {(1, 0, 3, 1), (3, 0)), ((1, 0, 3, 1),(3, 2)), ((1, 2, 3, 1), (3, 0)), ((1, 2, 3, 1), (3, 2))}.

3. Remove lists from A/B/C/A/ C/B/ if they contain equal id-vertices in non-overlapping positions (the positions in each list not involved above). The two substructures in g_1 whose composition yields A/B/C/A/ C/B/ are ((1, 0, 3, 1), (3, 2)) and ((1, 2, 3, 1), (3, 0)).

The matching algorithm depends on the number of query graph patterns p that need to be combined; p is somewhat difficult to determine for the average case. Roughly speaking, it is directly proportional to the query size and to the maximum valence of the vertices in the query. The larger l_p, the smaller p, though this relationship is data-dependent.

6.2.4 Data storage with BerkeleyDB

Berkeley DB is an open-source embedded database library that provides scalable, transaction-protected data management services to applications. It provides a variety of storage/access methods including dynamic hash tables, B+trees, persistent queues, and numbered records. In short, it is a toolkit for writing customized databases. Unlike standard databases that function as standalone servers, embedded databases such

as Berkeley DB are software libraries that developers can embed into their applications. The database functions in the application's process. The application itself can be a server and can use the embedded database library to implement custom database logic. Berkeley DB has a performance advantage over general-purpose databases because it does not have interprocess communication (IPC) overhead with application servers. Nor does Berkeley DB provide a generic complex query language like SQL. Instead, a developer can customize the database for specific access patterns. In GraphGrep, we use BerkeleyDB to store the mass data of the result of index construction process.

6.2.5 Complexity Analysis

Here is a description of the worst case complexity for GraphGrep as given in [71]. Let $|D|$ be the number of graphs in a database D. Let n, e and m be the number of nodes, the number of edges and the maximum valence (degree) of the nodes in a database graph, respectively. The worst case complexity of building a path representation for the database is $\mathcal{O}(\sum_i^{|D|}(n_i m_i^{l_p}))$, whereas the memory cost is $\mathcal{O}(\sum_i^{|D|}(l_p n_i m_i^{l_p}))$. Given a query with n_q nodes, e_q edges and m_q maximum valence, finding its patterns takes $\mathcal{O}(n_q + e_q)$ time; building its fingerprint takes $\mathcal{O}(n_q m_q^{l_p})$. Filtering the database takes linear time in the size of the database. The matching algorithm depends on the number of query graph patterns p, that need to be combined; p is somewhat difficult to determine for the average case. Roughly speaking, it is directly proportional to the query size and to the maximum valence of the nodes in the query. The larger l_p, the smaller p, though this relationship is datadependent. In general if $\tilde{n}$ is the maximum number of nodes having the same label, the worst case time complexity for the matching is $\mathcal{O}(\sum_i^{|D_f|}((\tilde{ni} m_i^{l_p})^p))$ with $|D_f|$ the size of the database after the filtering. For a query containing w pairs of nodes connected with wildcards the complexity for the matching is $\mathcal{O}(\sum_i^{|D_f|}((\tilde{n}_i m_i^{l_p})^p + w e_i))$.

6.3 VF

Purpose of this section is to illustrate VF [44, 46, 47], a graph matching algorithm of Attributed Relational Graphs (ARG) which, using a set of feasibility rules, allows to reduce the computational cost of the matching process.

The matching process is carried out by using a State Space Representation: a state represents a partial solution of the matching between two graphs, and a transition between states corresponds to the addition of a new pair of matched nodes. The feasibility rules prune states corresponding to partial matching solutions not satisfying the required graph morphism.

6.3.1 The Algorithm

This section shows the VF algorithm as reported in [46]. A matching process between two graphs $G_1 = (N_1, B_1)$ and $G_2 = (N_2, B_2)$ consists in the determination of a mapping M which associates nodes of the graph G_1 to nodes of G_2 and viceversa. As it is well known, different constraints can be imposed to M and consequently different mapping types can be obtained: monomorphism, strict isomorphism and graph-subgraph isomorphism are the most frequently used. We will consider the monomorphism known also as graph matching.

Generally, the mapping is expressed as the set of ordered pairs (n, m) (with $n \in G_1$ and $m \in G_2$) each representing the mapping of a node n of G_1 with a node m of G_2. Each pair is here denoted as component m_i of the mapping M. The State Space Representation (SSR) can be effectively used to describe a graph matching process, if each state s of the matching process represents a partial mapping solution. A partial mapping solution $M(s)$ is a subset of M, i.e. it contains only some components of M. In the adopted SSR representation a transition between two states corresponds to the addition of a new pair of matched nodes. In principle, the solutions to the

matching problem could be obtained by computing all the possible partial solutions and selecting the ones satisfying the wanted mapping type (Brute Force approach). In order to reduce the number of paths to be explored during the search, for each state on the path from s_0 to a goal state, we impose that the corresponding partial solution verifies some coherence conditions, depending on the desired mapping type. The rationale of our algorithm is that of introducing, given a state s, criteria for foreseeing if s has no coherent successors after a certain number of steps. It is clear that these criteria (feasibility rules) should allow to detect as soon as possible conditions leading to incoherence. States which do not satisfy a feasibility rule can be discarded from further expansions.

The VF Matching Algorithm

```
VF(G1,G2)
 1  C(s0) ← ∅;
 2  S(0) ← {s0};
 3  k ← 0;
 4  repeat
 5          S(k + 1) ← ∅;
 6          for each s in S(k)
 7            Compute the set P(s) of all possible pair of nodes
              of G1 and G2 not yet included in C(s), and set
              Q(s) ⊆ P(s) of the pairs that, if inserted into C(s),
              produce a coherent partial mapping;
 8            for each (n, m) in Q
 9              Add to S(k + 1) the state obtained adding (n, m) to C(s);
10            end for each;
11          end for each;
12          k ← k + 1;
13  until k = Card(N1) or S(k) = ∅;
14  end VF.
```

Figure 6.3: The VF Matching Algorithm.

In Fig. 6.3 (taken from [46]) the proposed algorithm is outlined. At each iteration of the outer loop, the algorithm considers the set $P(s)$ of node pairs that can be added to the state s, discarding those pairs which does not satisfy the feasibility rules. There are two kinds of feasibility rules, respectively regarding the syntax and the semantics of the graphs. The syntactic feasibility rules defined for an exact isomorphism has been described in [44]. Semantic compatibilities can be introduced very easily in the matching process: each time a node of the sample is compared to a node of the prototype to determine if a new pair can be added to the current partial solution, the attributes of the two nodes and of the branches linking them to the nodes already in s are tested for semantic compatibility (which obviously has to be defined with reference to the specific application domain). The exact matching algorithm can be extended by considering both transformations on the structure of the graph and on nodes and branches attributes. The considered syntactic transformations are the split of a node into a subgraph, the merge of a subgraph into a node and the insertion or deletion of a branch. Syntactic transformations are taken into account, during the expansion process of the search graph, by generating new states. When examining a state in the search process, the algorithm checks if there is a syntactic transformation that can be applied to it. For each applicable transformation, a new state s is added to the SSR graph. The nodes involved in a syntactic transformation are marked, in order to avoid reconsidering them in successive transformations. In this way we avoid the possible generation of infinite length paths in the search graph and prevent the possibility that a prototype is matched with a too different sample, as a consequence of the repeated application of some transformation.The conditions a state s has to meet in order that a transformation can be applied, depend on the transformation type. For each type of transformation, only the nodes in $P(s)$ are considered as candidates so as to ensure that, in the next step of the algorithm, the nodes generated by the transformation (or at least some of them) will be tested using the feasibility rules. In this way, if the

new paths are fruitless, they will be pruned as soon as possible.

To evaluate the computational complexity of the VF algorithm, we have to consider both the best and the worst case. The computational complexity of the matching algorithms depends on two factors: the number of the SSR states currently visited and the time needed for visiting each state. In [47] it is proven that the total cost for exploration of a single state is $\Theta(N)$ where N is the number of the nodes of the larger graph.

The best case happens when in each state only one of the potential successors satisfies the feasibility predicate (in the hypothesis that an isomorphism exists). In this situation, the number of explored states is equal to N, and thus the computational complexity in the best case is $\Theta(N^2)$.

In the worst case, in each state the predicate will not be able to avoid the visit of any of the successors, and the algorithm will have to explore all the states before reaching a solution. In [47] it is shown how the number of the states in this case is proportional to $N!$ and the computational complexity is $\Theta(N!N)$

6.4 GraphGrepVF for exact graph matching

In this section we will present GraphGrepVF, a new graph matching method, able to filter much more than GraphGrep, hence reducing the matching time needed to VF algorithm to perform the match. GraphGrepVF uses GraphGrep's filtering and VF's matching algorithm but it also introduces a new technique to prune paths of candidates graphs which will not contain for sure a solution. The idea is to use GraphGrep to construct the fingerprint of the database, a new hash table where it saves for each pattern of each graphs in the database of length 1 up to l_p, the edges of the idx-paths referred to that pattern. This hash table is stored into the BerkeleyDB for the same reasons of the index construction process built by GraphGrep. For example, if we have a dataset of Fig. 6.4 we will have the hash table like in table 6.3.

G1-ab	0 2
G1-ba	2 0
G1-baa	2 0
	0 1
G1-aab	1 0
	0 2
G1-aa	0 1
	1 0
G2-aa	0 1
	1 0
G2-ab	1 2
G2-ba	2 1
G2-aab	0 1
	1 2
G2-baa	2 1
	1 0
G3-ab	0 1
G3-bc	1 2
G3-ba	1 0
G3-cb	2 1
G3-abc	0 1
	1 2
G3-cba	2 1
	1 0

Table 6.3: New hash table of GraphGrepVF.

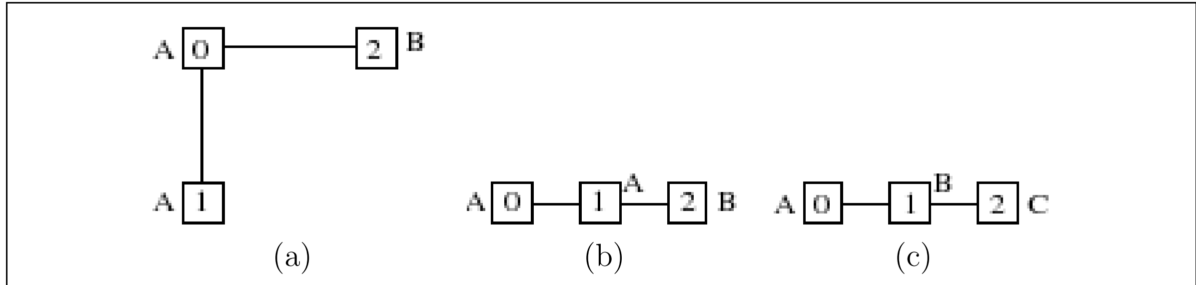


Figure 6.4: Dataset with three graphs.

When the query is parsed and it is decomposed in patterns, only among the graphs not pruned by the filtering, we retrieve the edges of the idx-paths corresponding to the query patterns. In this way the size of edges of the new graphs is much less than originals.

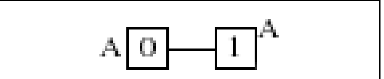


Figure 6.5: A graph query.

For example if we have the query of Fig. 6.5 and the graphs (a) and (b) of Fig. 6.4 are the candidates after the filtering, we will retrieve only the edge $(0-1)$ for the graph (a) and the edge $(0-1)$ for the graph (b) reducing the size of both graphs.

By experiments we have seen that even when no one graph is filtered, with this further technique, we are able to prune many parts of graphs obtaining reduced graphs, that is the same graphs but with a smaller number of edges.

6.5 Performance Analysis and Results

GraphGrepVF is written in C++ and it is an extension of GraphGrep whose source code is freely downloadable in [70, 3]. GraphGrepVF will be soon available for

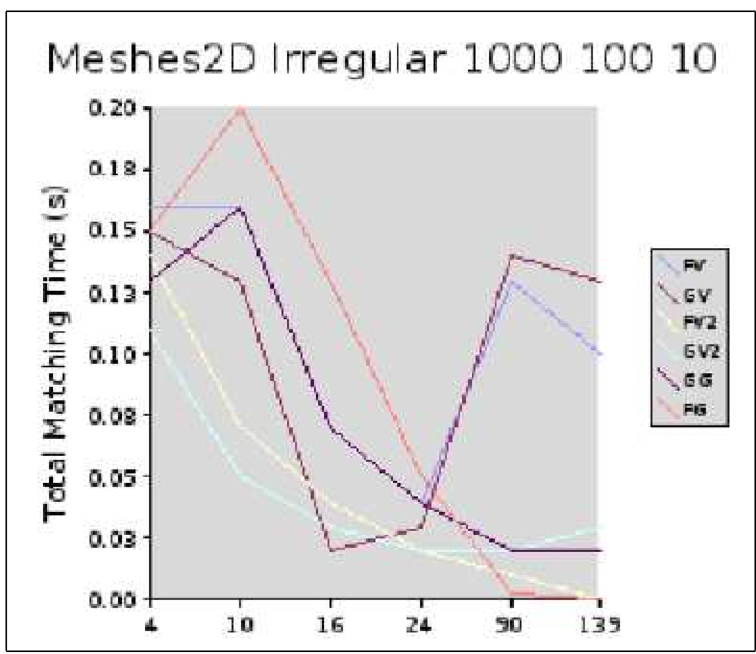


Table 6.4: Comparisons between FG,GV,GV2,FG,FV,FV2 methods for 1000 irregular meshes2D with 100 nodes and 10 labels.

free download. The machine used for the experiments is a Mobile Intel Pentium 4 with 512MB of RAM. We performed a set of numerical experiments on two kinds of datasets. The first one has been generated by using a graph benchmark created by us. The following kinds of graphs have been considered: regular and irregular meshes2D, regular and irregular meshes3D, random, valence with val 3,5,7.

For each category we have generated datasets of 1000 and 5000 graphs with 100 nodes for each graph; the number of labels varies from 4 to 60. The number of nodes for the queries is 4 up to 75. The second kind of dataset is the NCI databases. We have generated two datasets, one having 10.000 molecules and one having 50.000 molecules. Graphs in both datasets have an average number of 20 nodes; several graphs have up to 190 nodes. We have varied the query size for the NCI databases (23 to 189 nodes). For all the experiments, the l_p constant has been fixed to 4. We used a PC equipped

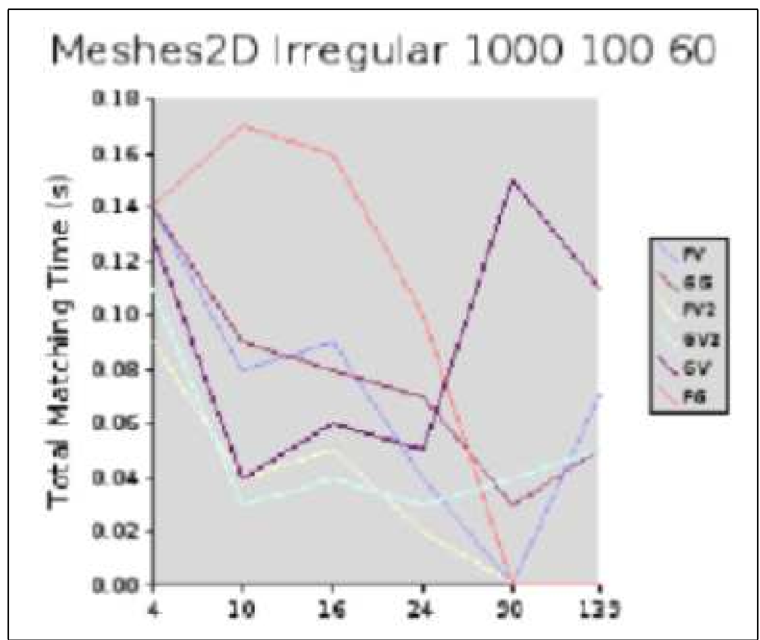


Table 6.5: Comparisons between FG,GV,GV2,FG,FV,FV2 methods for 1000 irregular meshes2D with 100 nodes and 60 labels.

with 2.30 GHz Pentium 4 Processor. We have compared six different methods. The six methods differ for the filtering tecnique, and for the matching tecnique. They are:

- GG : it performs the filtering and the matching by using GraphGrep.
- GV : it performs the filtering by using GraphGrep and the matching by using VF algorithm.
- GV2 : it performs the filtering by using GraphGrep and the matching by using VF algorithm. Moreover it uses the new filtering tecnique to reduce the dimension of the graphs to visit.
- FG : it performs the filtering by using Frowns and the matching by using GraphGrep.

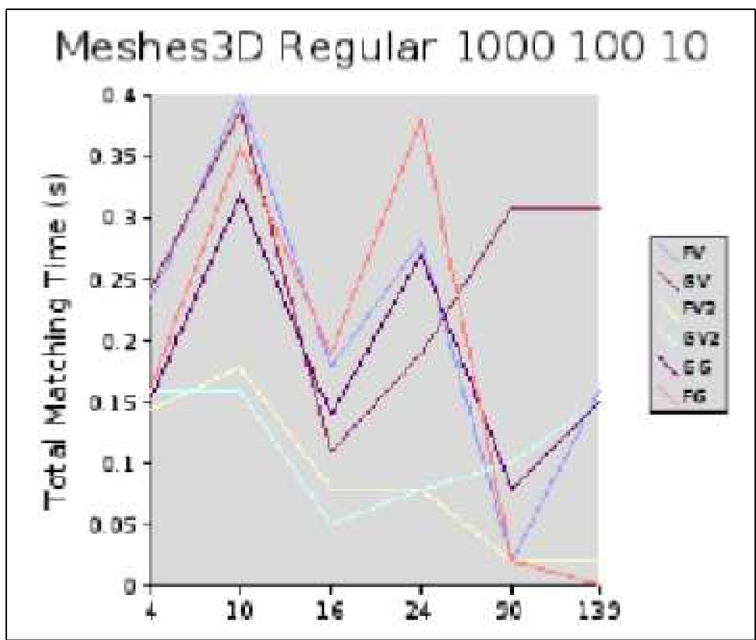


Table 6.6: Comparisons between FG,GV,GV2,FG,FV,FV2 methods for 1000 regular meshes3D with 100 nodes and 10 labels.

- FV : it performs the filtering by using Frowns and the matching by using VF algorithm.

- FV2 : it performs the filtering by using Frowns and the matching by using VF algorithm. Moreover it uses the new filtering tecnique to reduce the dimension of the graphs to visit.

As shown into the tables 6.4, 6.5, 6.6, 6.7 , 6.8, 6.9, 6.10, the GV2 or FV2 method, that is the ones which use the new filtering tecnique, performs always better in terms of time for every kind of query.

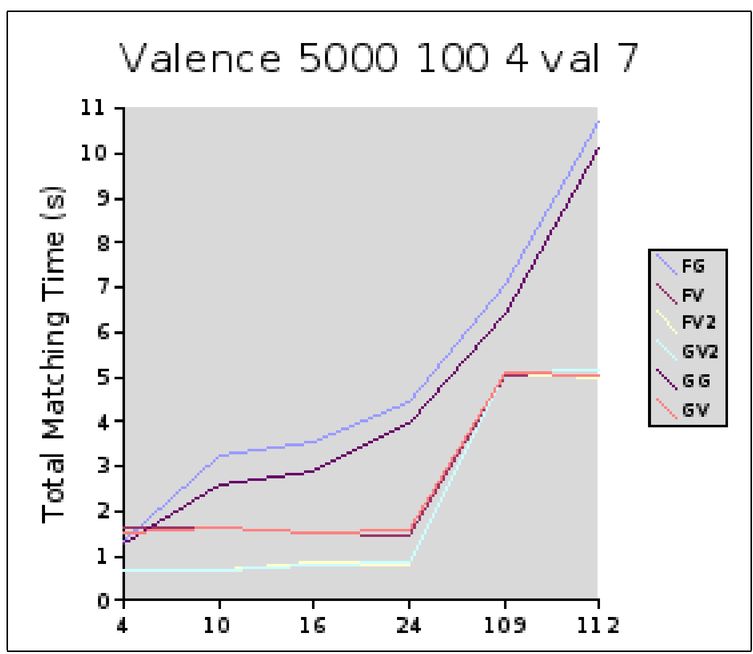


Table 6.7: Comparisons between FG,GV,GV2,FG,FV,FV2 methods for 5000 valence with 100 nodes, 4 labels and valence equal to 7.

6.6 Graph LInear DEscription language (GLIDE)

In this section we present GLIDE (Graph LInear DEscriptor), (see [68, 69]), a query language for a database of undirected graphs. The design of GLIDE has been influenced by two query languages: SMART [134, 88], and Xpath [42]. SMART is a query language for molecule databases coded using SMILES (Simplified Molecular Input Line Entry Specification) which is a nomenclature to represent a molecule. SMILES describes atoms and bonds of a molecule using their properties (element identity, isotope, formal charge, and implicit hydrogen count for the atoms; single, double, triple and aromatic for the bonds). SMART enriches SMILES's syntaxis including wildcards symbols to match any sequence of atoms and bonds. XML Path Language (XPath) is a query language to address parts of an XML document as a tree of nodes

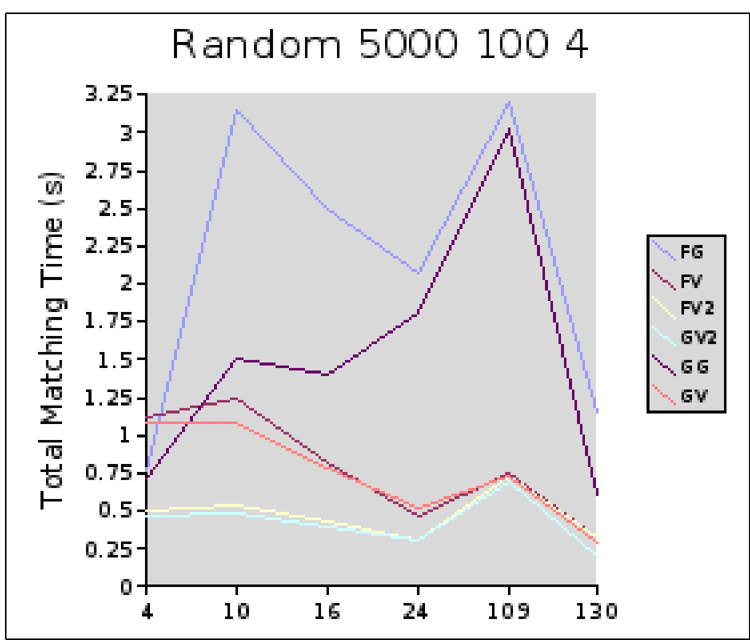


Table 6.8: Comparisons between FG,GV,GV2,FG,FV,FV2 methods for 5000 random with 100 nodes and 4 labels.

where adjacent nodes specifications are separated by the symbol '/' used in Unix file systems to describe the location of a file. It also contains wildcards to match unspecified paths. GLIDE uses graph expressions instead of path expressions, it represents the vertices with their labels (strings) and it uses the symbol '/' to separate two vertices.

6.6.1 Syntaxis and semantic of GLIDE

The main idea in GLIDE is to represent a graph, in linear notation, as a set of branches where each vertex is presented only once. Vertices are represented using their labels (see Table 6.11(a) (taken from [68, 69])) and they are separated using slashes (Table 6.11(b) (taken from [68, 69]); branches are grouped using nested parentheses '(' and ')' (Table 6.11(c)-(d) (taken from [68, 69])) and cycles are broken by cutting

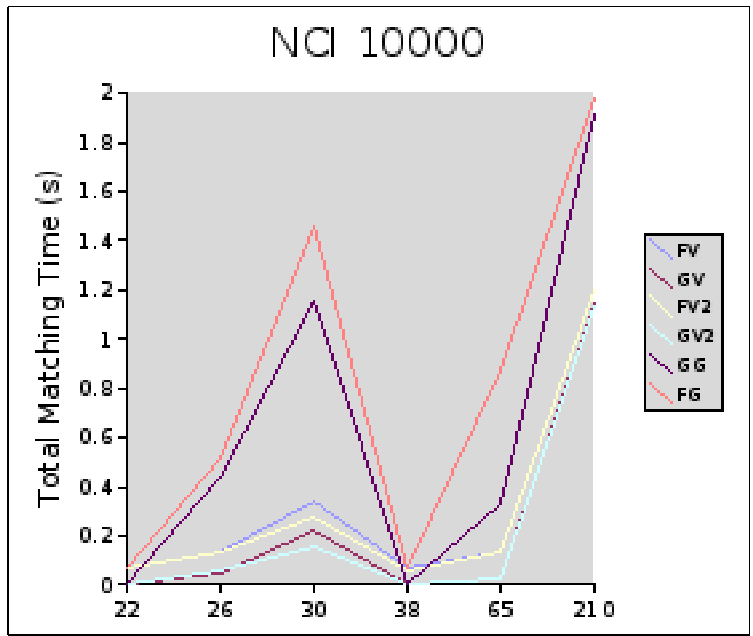


Table 6.9: Comparisons between FG,GV,GV2,FG,FV,FV2 methods for NCI database with 10000 molecule.

an edge and labeling it with an integer (Table 6.11(c)-(d) (taken from [68, 69])). The vertices of the cut edge are represented by their labels followed by '%', the integer and '/'. If the same vertex is a vertex of several cut edges the label of the vertex is followed by a list of '%' and integers (Table 6.11(g) (taken from [68, 69])). Non specified components in a graph are described using wildcards $\{* + .?\}$ (see Fig. 6.12 (taken from [68, 69])). The wildcards represent single vertices or paths. The semantic of the wildcards is given in based on the elements in a graph that during a search they can match: '.' matches any single vertex; '*' matches zero or more vertices; '?' matches zero or one vertex; '+' matches one or more vertices.

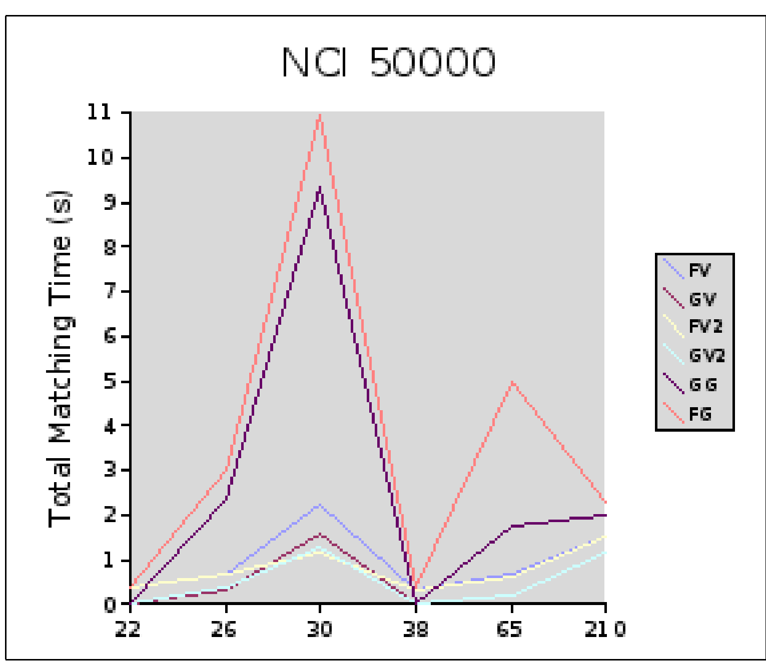


Table 6.10: Comparisons between FG,GV,GV2,FG,FV,FV2 methods for NCI database with 50000 molecule.

6.7 Extension of GraphGrepVF for inexact graph matching

In this section we will explain how GraphGrepVF is used to perform inexact matching with query expressed in GLIDE query language.

When a query in GLIDE is given, a parser computes all the wildcards and prepares a table of them (parser table) for the next phase, the matching. The idea is to try to filter the dataset using only the piece of query graph which for sure will be in the solution: each graph in the dataset not cointaning these pieces will be pruned out. Each entry of the parser table is composed by 4 objects: two indexes of the two nodes which share the wildcard (-1 if only one node is involved), another integer value L and the operator. The integer value L counts how many times the operator must be

Graph	GLIDE Representation
a	a/
a b	a/b/
a h c f	a/h/c/f/
a b h c	a/(h/c/)b/
a b l d l h c	i/(b/a/(l/)h/c)d/
d i c	i%1/c/d%1/
c i a h d	a%1/h/c%1%2/d/i%2

Table 6.11: GLIDE representation of graphs. (a) A vertex. (b) An edge. (c) A path. (d)-(e) Branches. (f)-(g) Graph with cycles. The dashed edges are the cut edges.

Wildcard	Semantic (matching with)	Example	
		GLIDE Expression	Graph
.	any single vertex	a/./b/	a b
*	zero or more vertices	a/*/b/	a b
?	zero or one vertex	a/?/b/	a b
+	one or more vertices	a/+/b/	a b

Table 6.12: Wildcards in GLIDE.

Query in GLIDE	Parser Table
a/./././+/*/*/b/	w[0]=[0 1 4 G]
a/*/*/*/b/	w[0]=[0 1 0 G]
a/././*/*/b/	w[0]=[0 1 2 G]
a/././b/	w[0]=[0 1 2 E]
a/././?/b/	w[0]=[0 1 3 L]
a/(./b/)(*/d/e/f/)(./)	w[0]=[0 1 1 E] w[1]=[0 2 0 G] w[2]=[0 -1 1 E]
a(./)(+/./c/)	w[0]=[0 -1 1 E] w[1]=[0 1 2 G]

Table 6.13: Examples of parser tables for some queries.

applied. The operator can assume the following values under the hypothesis that the two indexes are a and b:

- 'G' when there is some '+' or '*' and it means that we are looking for a path from 'a' to 'b' with the number of internal nodes greater or equal than the value L associated with this entry;
- 'L' when there is some '?' and it means that we are looking for a path from 'a' to 'b' with the number of internal nodes less or equal than the value L associated with this entry;
- 'E' when there is some '.' and it means that we are looking for a path from 'a' to 'b' of with the number of internal nodes equal to the value L associated with this entry.

In the table 6.13 there are some examples of query in GLIDE with the associated list of entries.

Once that the filtering process has been completed for the pieces of the query, the matching step is performed on each entry of the list generated by the parser.

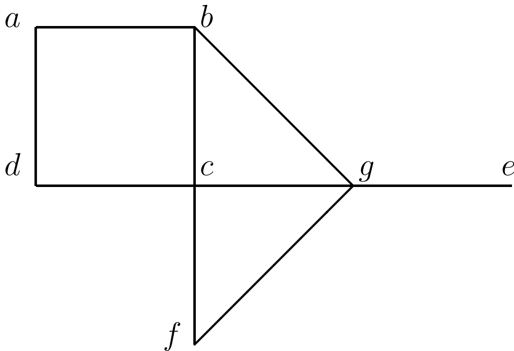


Figure 6.6: A graph.

Let us consider the query and its parser table in the last row of table 6.13. The filtering will prune all the graphs which do not contain neither 'a' nor 'c' vertices. Let us suppose that we have only one graph remained after the filtering, the one in figure 6.6. The next step is to perform the matching on every entry w[i] of the parser table. Thus first of all we check if there is any paths satisfying w[0]. w[0] asks for paths of length 1 starting from a node 'a' to anything else. In the graph in figure 6.6 the paths ab and ad satisfy w[0]. For w[1] we check if there is some path from 'a' to 'c' of length ≥ 2. Valid local solutions paths are $abgc$ of length 3 and $abgfc$ of length 4.

At this point we have a list of local solutions for every w[i] of the parser table. All the local solutions of different w[i] must be merged together and checked to see if they can be considered valid global solutions.

The rules we have decided to introduce to decide when global solutions are valid wheter not are the following:

1. if one solution x for w[i], with $i = 1, \ldots, n$, is contained in another one y for

w[j], with $j = 1, \ldots, n$ and $j \neq i$, then xy will not belong to any valid global solutions;

2. if one solution x for w[i], with $i = 1, \ldots, n$, shares the same initial node of a solution y for w[j], with $j = 1, \ldots, n$ and $j \neq i$, then xy will not belong to any valid global solutions;

3. if one solution x for w[i], with $i = 1, \ldots, n$, shares the same final node of a solution y for w[j], with $j = 1, \ldots, n$ and $j \neq i$, then xy will not belong to any valid global solutions.

Reminding that in the example above we had as solutions for w[0], the paths ab and ad, and as solutions for w[1] we had paths $abgc$ and $abgfc$; now we have to make the cartesian product of all local solutions to see which global solution satisfies the rules just introduced.

The four possible global solutions are $\{ab, abgc\}$, $\{ab, abgfc\}$, $\{ad, abgc\}$ and $\{ad, abgfc\}$: $\{ab, abgc\}$ and $\{ab, abgfc\}$ are not two valid global solutions because in both the local solution ab is contained in the other local solution. Instead, the other two global solutions $\{ad, abgc\}$ and $\{ad, abgfc\}$ are valid and they are the output of the inexact match of the graph in Fig. 6.6 and query in the last row of the table 6.13.

Of course, dealing with inexact matching with the above procedure could be very expensive in terms of time complexity. However a simple scheme of optimization without losing solutions is possible but not for all the wildcards. For '.' and '?' we can reduce the number of nodes to visit for each graph in the dataset to be processed. Infact, during a visit, if the current path length is greater than the value L of the parser table entry, then we can skip the visit of this path and continue with others. Hence, performing inexact matching in presence of '.' and '?' lets us prune many vertexes and the overall process is enough fast. However, for '*' and '+', we do not

have any optimization step yet and the inexact matching could be very expensive; of course it depends on the structure of the graphs in the dataset to be matched.

Chapter 7

Conclusions

In the first part of this thesis, we have extended the ideas of the most successful best match retrieval data structures, such as M-Tree, MPV-Tree, FQ-Tree, and List of Clusters, by introducing pivots based on the farthest pairs (Antipoles) in a data set. The resulting Antipole Tree is a bisector tree using pivot-based clustering with bounded diameter. Both range and k-nearest neighbor searches are performed by eliminating those clusters which cannot contain the result of the query. Antipoles and clusters centroids are found by playing a linear time randomized tournament among the elements of the input data set.

A data-dependent aspect of the algorithm is to control the proliferation of clusters through the introduction of a suitable diameter threshold. In order to properly define such a threshold we propose a statistical analysis on the set of pairwise distances. To decide when to split, we need an estimate of the ratio between the pseudo-diameter (Antipole length) and the real diameter. Since no guaranteed approximation algorithm for diameter computation in general metric spaces can exist, we use the approximation ratio given by a very efficient algorithm for diameter computation in Euclidean spaces together with the intrinsic dimension of the given metric space.

By using the tournament size equal to 3 or $d-1$, where d is the intrinsic dimension of the metric space, we obtained good experimental results. However, we are currently

investigating from a theoretical point of view how to determine an optimal value for the tournament size parameter. Extensive experimentations have been performed on both synthetic and real data sets, with normal and clustered distributions. All the experiments have shown that our proposed structure outperforms the most successful data structures for best match search by a factor ranging between 1.5 and 2.5.

This thesis has also shown how the Antipole Tree has been applied successfully over four different applications, each showing the Antipole Tree's strength and efficiency.

In the second part of the thesis, we presented a new graph search algorithm, GraphGrepVF, which is able to filter many more graphs than the previous algorithms, GraphGrep and VF. GraphGrepVF is realized by introducing a new filter method that can filter paths that do not contain the solution. Experiments have shown that GraphGrepVF performs faster than GraphGrep and VF.

This thesis also discussed how to deal with inexact matching. For inexact matching, we used GLIDE, a graph query language, to express the graph queries and the kind of inexactness the user wants. GraphGrepVF, speeded up during inexact matching by a new optimization step, returns all the subgraphs which satisfied the query.

There are numerous directions for future research. First of all, an improved version of the Antipole Tree is under study. Also it has been already applied in networks as a dynamic clustering algorithm. Furthermore, a first version of parallelized Antipole tree will be released soon. New algorithms to find an approximated solution to 1-median centroid are being explored; one new idea to use an approximated algorithm to solve the k-median problem and to transform the Antipole Tree as a tree with k children. Finally, new clustering algorithm over graphs who clusters regions of nodes will be investigated; the idea is to use the approximated 1-median algorithm seen in section 3.2 to find a centroid of such regions and hence to create the clusters according to such nodes.

As far as graph matching is concerned, other filtering tecniques to speed-up the

process are under study. We are exploring other ways to make the inexact matching faster.

Bibliography

[1] http://graphics.stanford.edu/projects/texture/demo/synthesis_VisTex_192.html, Texture Synthesis: VisTex Texture.

[2] http://cygnus.uta.edu/subdue, The SUBDUE Knowledge Discovery System.

[3] http://alpha.dmi.unict.it/ ctnyu/. University of Catania and New York University.

[4] P. Agarwal, J. Matousek, and S. Suri. Farthest neighbors, maximum spanning trees and related problems in higher dimensions. *In Comput. Geom.: Theory and Appls*, 1:189–201, 1992.

[5] C. Aggarwal, J.L. Wolf, P.S. Yu, and M. Epelman. Using unbalanced trees for indexing multidimensional objects. *Knowledge and Information Systems*, 1(3):157–192, 1999.

[6] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. *Proceedings of ACM SIGMOD*, pages 94–105, 1998.

[7] M. Ashikhmin. Synthesizing natural textures. *ACM Symposium on Interactive 3D Graphics*, pages 217–226, 2001.

[8] G. Barequet and S. Har-Peled. Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 82–91, 1999.

[9] S. Battiato, D. Cantone, D. Catalano, G. Cincotti, and M. Hofri. An efficient algorithm for the approximate median selection problem. *Proceedings of the 4th Italian Conference on Algorithms and Complexity (CIAC 2000), volume 1767 of Lecture Notes in Computer Science, Springer-Verlag*, pages 226–238, 2000.

[10] S. Battiato, A. Pulvirenti, and D. Reforgiato. Antipole clustering for fast texture synthesis. *Proceedings of Winter School of Computer Graphics (WSCG)*, 2003.

[11] S. Berchtold, D.A. Keim, and H.P. Kriegel. The x-tree: An index structure for high-dimensional data. *Proceedings of the 22th Intl. Conference on Very Large Database*, pages 28–39, 1996.

[12] J.R. Bergen and B. Julesz. Rapid discrimination of visual patterns. *IEEE Transactions on Systems Man and Cybernetics*, 13:857–863, 1993.

[13] M.W. Berry, Z. Drmac, and E.R. Jessup. Matrices, vector spaces, and information retrieval. *SIAM Review*, 41(2):335–362, 2003.

[14] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is nearest neighbor meaningful? *Proceedings of the 7th Intl. Conference on Database Theory*, 1540:217–235, 1999.

[15] P. Billault. Texture synthesis algorithms. Image Signal Departement de Mathematiques Appliquees, Technical report, 2001.

[16] M. Birgmeier. Frowns chemoinformatics system. http://frowns.sourceforge.net/.

[17] G. Di Blasi and D. Reforgiato Recupero. Fast colorization of gray images. *Eurographics Italian Chapter*, 2004.

[18] B.Luo, R.C.Wilson, and E.R.Hancock. Spectral feature vectors for graph clustering. *Proceedings of joint Syntactical and Structural Pattern Recognition and Statistical Pattern Recognition*, 2396:83–93, 2002.

[19] B.Luo, R.C.Wilson, and E.R.Hancock. Spectral clustering of graphs. *Proceedings of 4th IAPR-TC15 Graph based Representations in Pattern Recognition*, pages 190–201, 2003.

[20] N. Bolshakova and F. Azuaje. Improving expression data mining through cluster validation. *Information Technology Applications in Biomedicine, 2003*, pages 19–22, 2003.

[21] J.S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. *Computer Graphics, ACM SIGGRAPH*, pages 361–368, 1997.

[22] T. Bozkaya and M. Ozsoyoglu. Indexing large metric spaces for similarity search queries. *ACM Transaction on Database Systems*, 24(3):361–404, 1999.

[23] S. Brin. Near neighbor search in large metric spaces. *Proceedings of the 21th International Conference on Very Large Data Bases*, pages 574–584, 1995.

[24] H. Bunke. Graph-based tools for data mining and machine learning. *Proceedings of Machine Learning and Data Mining in Pattern Recognition*, pages 7–19, 2003.

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

[26] P.J. Burt and E.H. Adelson. The laplacian pyramid as a compact image code. *IEEE Transactions on Communications*, 31:532–540, 1983.

[27] B. Bustos and G. Navarro. Probabilistic proximity searching algorithms based on compact partitions. *Proceedings of SPIRE*, pages 284–297, 2002.

[28] I. Calantari and G. McDonald. A data structure and an algorithm for the nearest point problem. *In IEEE Transaction on Software Eng.*, 9(5), 1983.

[29] D. Cantone, G. Cincotti, A. Ferro, and A. Pulvirenti. An efficient algorithm for the 1-median problem. *SIAM Journal on Optimization (to appear)*, 2004.

[30] D. Cantone, A. Ferro, A. Pulvirenti, D. Reforgiato, and D. Shasha. Antipole tree indexing to support range search and k-nearest-neighbor search in metric spaces. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2004.

[31] T.M. Chan. Approximating the diameter, width, smallest enclosing cylinder, and minimum-width annulus. *In International Journal of Computational Geometry and Applications*, 12(1-2):67–85, 2002.

[32] M. Charikar, S. Guha, E. Tardos, and D. Shmoys. A constant factor approximation for the k-median problem. *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 1–10, 1999.

[33] E. Chavez and G. Navarro. An effective clustering algorithm to index high dimensional metric spaces. *Proceedings of SPIRE*, pages 75–86, 2000.

[34] E. Chavez and G. Navarro. A probabilistic spell for the curse of dimensionality. *Proceedings of Third Workshop on Al-gorithm Engineering and Experimentation (ALENEX01) volume 2153 of Lecture Notes in Computer Science*, pages 147–160, 2001.

[35] E. Chavez, G. Navarro, R. BaezaYates, and J.L. Marroquin. Searching in metric spaces. *ACM Computing Surveis*, 33(3):273–321, 2001.

[36] B. Chazelle. Computational geometry: a retrospective. *Proceedings of the 26th Annual ACM Symposium on Theory of Computing*, pages 75–94, 1994.

[37] S.A. Chervitz, E.T. Hester, C.A. Ball, K. Dolinski, S.S. Dwight, and M.A. Harris. Using the saccharomyces genome database (sgd) for analysis of protein similarities and structure. *Nucleic Acids Res*, 27:74–78, 1999.

[38] D. Christendat, A. Yee, A. Dharamsi, Y. Kluger, A. Savchenko, and J.R. Cort. Structural proteomics of an archaeon. *Nat Struct Biol 2000*, 7:903–909, 2000.

[39] P. Ciaccia and M. Patella. Bulk loading the m-tree. *Proceedings of the* 9th *Australasian Database Conference (ADC)*, 1998.

[40] P. Ciaccia and M. Patella. Pac nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. *Proceedings of the IEEE 16th International Conference on Data Engineering*, pages 244–255, 2000.

[41] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. *Proceedings of the 23th International Conference on Very Large Data Bases*, pages 426–435, 1997.

[42] J. Clark and S. DeRose. http://www.w3.org/TR/xpath, 1999.

[43] K.L. Clarkson. Nearest neighbor queries in metric spaces. *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, 1997.

[44] L. Cordella, P. Foggia, C. Sansone, and M. Vento. An efficient algorithm for the inexact matching of arg graphs using a contextual transformational model. *In proceedings of the 13th ICPR. IEEE Computer Society Press*, 3(180-184), 1996.

[45] L.P. Cordella, P. Foggia, C. Sansone, F. Tortorella, and M. Vento. Vf algorithm. http://amalfi.dis.unina.it/graph/.

[46] L.P. Cordella, P. Foggia, C. Sansone, F. Tortorella, and M. Vento. Graph matching: a fast algorithm and its evaluation. *Proceeding of the IEEE International Conference in Pattern recognition (ICPR)*, 1998.

[47] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. Performance evaluation of the vf graph matching algorithm. *Proceeding of the IEEE International Conference in Pattern recognition (ICPR)*, 1999.

[48] M.C. Costanzo, M.E. Crawford, J.E. Hirschman, J.E. Kranz, P. Olsen, and L.S. Robertson. Ls ypd, pombepd and wormpd: model organism volumes of the

bioknowledge library, an integrated resource for protein information. *Nucleic Acids Res 2001*, 29:75–79, 2001.

[49] T.M. Cover and P.E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, 13(1):21–27, 1967.

[50] G.R. Cross and A.K. Jain. Markov random field texture models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5:25–39, 1983.

[51] D.R. Cutting, D.R. Karger, J.O. Pedersen, and J.W. Tukey. Scatter / gather: A cluster-based approach to browsing large document collections. *Proc. ACM SIGIR 92*, pages 318–329, 1992.

[52] S. Deerwester, S.T. Dumais, T.K. Landauer, G.W. Furnas, and R.A. Harshman. Indexing by latent semantic analysis. *Journal of the Society for Information Science*, 41(6):391–407, 1990.

[53] S. Dickinson, M. Pelillo, and R. Zabih. Introduction to the special section on graph algorithms in computer vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 23(10), 2001.

[54] A. Efros and T. Leung. Texture synthesis by a non-parametric sampling. *Proceedings of the IEEE International Conference on Computer Vision*, 2:1033–1038, 1999.

[55] M. Ester, H.P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. *Proceedings of the 2nd International Conference on Knowledge Discovery in Databases and Data Mining*, 1996.

[56] T. Feder and D.H. Greene. Optimal algorithms for approximate clustering. *Proceedings of the 20st Annual ACM Symposium on Theory of Computing*, pages 434–444, 1988.

[57] S. Fields. Proteomics. proteomics in genomeland. *Science 2001*, 291:1221–1224, 2001.

[58] E. Fix and J. Hodgens. Discriminatory analysis: Nonparametric discrimination: Consistency properties. Technical Report 4, Project Number 21-49-004, School of Aviation Medicine, Randoplh Air Force Base, TX, 1951.

[59] G.W. Flake, S. Lawrence, C.L. Giles, and F.M. Coetzee. Self-organization of the web and identification of communities. *IEEE Computer 2002*, 35:66–71, 2002.

[60] A. Wai Chee Fu, P. M.S. Chan, Y.L. Cheung, and Y.S. Moon. Dynamic vp-tree indexing for n-nearest neighbor search given pair-wiewdistances. *VLDB Journal*, pages 311–321, 1999.

[61] K. Fukunaga. Introduction to statistical pattern recognition. *Academic Press*, 1, 1990.

[62] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[63] L. Galanis, E. Viglas, D.J. DeWitt, J.F. Naughton, and D. Maier. Following the paths of xml data: An algebraic framework for xml query evaluation. *Submitted*, 2001.

[64] V. Ganti, R. Ramakrishnan, J. Gehrke, A. Powell, and J. French. Clustering large datasets in arbitrary metric spaces. *Proceedings of the IEEE 15th International Conference on Data Engineering*, pages 502–511, 1999.

[65] M. Garey and D. Johnson. Computers and intractability: A guide to the theory of np-completeness. *Freeman and Company*, 1979.

[66] A. Gersho and R. Gray. *Vector Quantization and Signal Compression.* Kluwer Academic, 1991.

[67] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. *Proceedings of 25th VLDB Conference*, pages 518–529, 1999.

[68] R. Giugno. Searching algorithms and data structures for combinatorial, temporal and probabilistic databases. PhD Thesis. University of Catania 2002.

[69] R. Giugno and D. Shasha. Glide, a graph linear query language. http://www.cs.nyu.edu/shasha/papers/graphgrep/graphgrep_2002.html.

[70] R. Giugno and D. Shasha. Graphgrep matching tool. http://www.cs.nyu.edu/shasha/papers/graphgrep/.

[71] R. Giugno and D. Shasha. Graphgrep, a fast and universal method for querying graphs. *Proceeding of the IEEE International Conference in Pattern recognition (ICPR)*, 2002.

[72] A.V. Goldberg. Finding a maximum density subgraph. Technical Report UCB/CSD University of California, Berkeley, CA, 1984.

[73] R.C. Gonzales and P. Wintz. Digital image processing. *Addison-Wesley Publishing, Reading MA*, 1987.

[74] T.F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[75] T.F. Gonzalez. Covering a set of points in multidimensional space. *Information Processing Letter*, 40:181–188, 1991.

[76] S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. *Proceedings of ACM SIGMOD*, pages 73–84, 1998.

[77] S. HarPeled. A practical approach for computing the diameter of a point set. *Proceedings of the 17th Symposium on Computational Geometry*, pages 177–186, 2001.

[78] D.J. Heeger and J.R. Bergen. Pyramid-based texture analysis/synthesis. *Computer Graphics, ACM SIGGRAPH*, pages 229–238, 1995.

[79] A. Hertzmann, C.E. Jacobs, N. Oliver, B. Curless, and D.H. Salesin. Image analogies. *Proceedings of ACM-SIGGRAPH*, pages 327–340, 2001.

[80] A. Hertzmann, C.E. Jacobs, N. Oliver, B. Curless, and D.H. Salesin. Image analogies. *In Proceedings of ACM SIGGRAPH 2001*, 2001.

[81] G.R. Hjaltsson and H. Samet. Distance browsing in spatial database. *ACM Transaction on Information Systems*, 24(2):265–318, 1999.

[82] D.S. Hochbaum. *Approximation Algorithms for NP-Hard Problems.* PWS Publishing Compagny, 1997. D.S. Hochbaum editor.

[83] D.S. Hochbaum and W. Maass. Approximation schemes for covering and packing problems in image processing and vlsi. *Journal of the ACM*, 32:130–136, 1985.

[84] David Hull. Improving text retrieval for the routing problem using latent semantic indexing. *In Proceedings of the 17th ACM/SIGIR Conference*, pages 282–290, 1994.

[85] P. Husbands, H. Simon, and C. Ding. On the use of singular value decomposition for text retrieval. *Proc. of SIAM Comp. Info. Retrieval Workshop*, pages 145–156, 2001.

[86] P. Indyk. Sublinear time algorithms for metric space problems. *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 428–434, 1999.

[87] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 604–613, 1998.

[88] C.A. James, D. Weininger, and J. Delany. Daylight theory manual-daylight 4.71. *Daylight Chemical Information Systems, www.daylight.com*, 2000.

[89] C.Traina Jr, A. Traina, D. Seeger, and C. Faloutsos. Slim-trees: High performace metric trees minimizing overlap between nodes. *Proceedings of the 7th International Conference on Extending Database Technology*, pages 51–56, 2000.

[90] S.K. Kim, J. Lund, M. Kiraly, K. Duke, M. Jiang, and J.M. Stuart. A gene expression map for caenorhabditis elegans. *Science 2001*, 293:2087–2092, 2001.

[91] S. Kosinov and T. Caelli. Inexact multisubgraph matching using graph eigenspace and clustering models. *Proceedings of joint Syntactical and Structural Pattern Recognition and Statistical Pattern Recognition*, 2002.

[92] G. Kowalski. *Information retrieval systems: Theory and implementation.* Boston: Kluwer Academic Publishers, 1997.

[93] Chen Li, E. Chang, and H. Garcia-Molina G. Wiederhold. Clustering for approximate similarity search in high-dimensional spaces. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):792–808, 2002.

[94] Steinbach M., Karypis G., and Kumar V. A comparison of document clustering techniques. *Proc. Text Mining Workshop, KDD 2000*, pages 1–11, 2000.

[95] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26:54–66, 1997.

[96] B.T. Messmer and H. Bunke. Subgraph isomorphism detection in polynomial time on preprocessed model graphs. *ACCV, Lecture Notes in Computer Science. Springer*, pages 383–392, 1996.

[97] H.W. Mewes, D. Frishman, C. Gruber, B. Geier, D. Haase, and A. Kaps. Mips: a database for genomes and protein sequences. *Nucleic Acids Res 2000*, 28:37–40, 2000.

[98] T. Milo and D. Suciu. Index structures for path expressions. *ICDT*, pages 277–295, 1999.

[99] M. Minsky and S. Papert. *Perceptrons.* MIT Press, Cambridge MA, 1969.

[100] T.M. Mitchell. *Machine Learning.* McGraw-Hill, 1997.

[101] G. Navarro. Searching in metric spaces by spatial approximation. *The VLDB Journal*, 11:28–46, 2002.

[102] A.Y. Ng, M. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. *Advances in Neural Information Processing Systems 14: Proceedings of the 2001*, 2001.

[103] R.T. Ng and J. Han. Clarans: a method for clustering objects for spatial data mining. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1003–1016, 2002.

[104] H. Noltemeier, K. Verbarg, and C. Zirkelbach. Monotonous bisector* trees - a tool for efficient partitioning of complex schemes of geometric objects. *Data Structure and Efficient Algorithms volume 594 of Lecture Notes in Computer Science*, pages 186–203, 1992.

[105] J.M. Ogden, E.H. Adelson, J.R. Bergen, and P.J. Burt. Pyramid-based computer graphics. *RCA Engineer*, 30:4–15, 1985.

[106] K. Popat and R. Picard. Novel cluster-based probability model for texture synthesis, classification, and compression. *Visual Communications and Image Processing*, pages 756–768, 1993.

[107] J. Portilla and E.P. Simoncelli. A parametric texture model based on joint statistics of complex wavelet coefficients. *International Journal of Computer Vision*, 40(1):49–71, 2000.

[108] F.P. Preparata and M.I. Shamos. *Computational Geometry, An Introduction.* pringer-Verlag, New York, 1985.

[109] C.M. Procopiuc. Geometric techniques for clustering theory and practice. *Ph.D. Dissertation - Duke University*, 2001.

[110] A. Pulvirenti. Algorithms and data structures for optimization and advanced search problems on metric spaces of high dimension and their applications. PhD Thesis. University of Catania 2002.

[111] S. Rajappap. Interactive biasing in graph-based data mining. *Master Thesis in Computer Science and Engineering*, 2003.

[112] D. Reforgiato and D. Shasha. Graphclust. http://www.cs.nyu.edu/cs/faculty/shasha/papers/GraphClust.html.

[113] A. Sanfeliu and K.S. Fu. A distance measure between attributed relational graphs for pattern recognition. *IEEE Trans. on SMC*, 13:353–362, 1983.

[114] J. Shanmugasundaram, H. Gang., K. Tufte, C. Zhang, D. De Witt, and J.F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. *VLDB Journal*, 1999.

[115] L.G. Shapiro and R.M. Haralick. Structural description and inexact matching. *IEEE Trans. on PAMI*, 3:505–519, 1981.

[116] M. Shapiro. The choice of reference points in best-match file searching. *Communication of the ACM*, 20:339–343, 1997.

[117] D. Shasha, J. T. L. Wang, and R. Giugno. Algorithmics and applications of tree and graph searching. *Symposium on Principles of Database Systems*, 2002.

[118] D. Shasha and T.L. Wang. New techniques for best-match retrieval. *ACM Transaction on Information Systems*, 8(2):140–158, 1990.

[119] G. Sheikholeslami, S. Chatterjee, and A. Zhang. Wavecluster: A wavelet based clustering approach for spatial data in very large databases. *VLDB Journal*, 8(3-4):289–304, 2000.

[120] L. Sheng, Z.M. Ozsoyoglu, and G. Ozsoyoglu. A graph query language and its query processing. *ICDE*, pages 572–581, 1999.

[121] J. Silberg. http://www.cinesite.com/core/press/articles/1998/10_00_98-team.html, 1998.

[122] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[123] D. Suciu. An overview of semistructured data. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 29, 1998.

[124] S. Sumanasekara and M.V. Ramakrishna. Chilma: An efficient high dimensional indexing structure for image database. *Proceedings of the First IEEE Pacific-Rim Conference on Multimedia*, pages 76–79, 2000.

[125] W.H. Tsai and K.S. Fu. Error-correcting isomorphisms of attributed relational graphs for pattern analysis. *IEEE Trans. on SMC*, 9:757–768, 1979.

[126] W.H. Tsai and K.S. Fu. Subgraph error-correcting isomorphisms for syntactic pattern recognition. *IEEE Trans. on SMC*, 13:48–62, 1983.

[127] P. Uetz, L. Giot, G. Cagney, T.A. Mansfield, R.S. Judson, and J.R. Knight. A comprehensive analysis of protein-protein interactions in saccharomyces cerevisiae. *Nature 2000*, 403:623–627, 2000.

[128] J.K. Uhlmann. Satisfying general proximity/similarity queries with metric. *Information Processing Letters*, 40:175–179, 1991.

[129] J. Ullmann. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23:31–42, 1976.

[130] J. Wang, B. Shapiro, and D. Shasha. Pattern discovery in biomolecular data. *New York Oxford, oxford university press edition*, 1999.

[131] L.Y. Wei. Texture synthesis by fixed neighborhood searching. *Ph.D. Dissertation - Stanford University*, 2002.

[132] L.Y. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. *Proceedings of ACM-SIGGRAPH*, pages 479–488, 2000.

[133] L.Y. Wei and M. Levoy. Texture synthesis over arbitrary manifold surfaces. *Proceedings of ACM-SIGGRAPH 2001*, pages 355–360, 2001.

[134] D. Weininger. Smiles, introduction and encoding rules. *Journal Chemical Information in Computer Science*, 28(31), 1988.

[135] T. Welsh, M. Ashikmin, and K. Mueller. Transferring color to greyscale images. *Proceedings of ACM SIGGRAPH 2002*, 2002.

[136] E.A. Winzeler, D.D. Shoemaker, A. Astromoff, H. Liang, K. Anderson, and B. Andre. Functional characterization of the s. cerevisiae genome by gene deletion and parallel analysis. *Science 1999*, 285:901–906, 1999.

[137] Y.N. Wu, S.C. Zhu, and X.W. Liu. Equivalence of julesz ensemble and frame models. *International Journal of Computer Vision*, 38(30):245–261, 2000.

[138] Y. Xu, B. Guo, and H.Y. Shum. Chaos mosaic: Fast and memory efficient texture synthesis. Technical Report MSR-TR-2000-32, Microsoft Research, 2000.

[139] Y. Xu, S.C. Zhu, B. Guo, and H.Y. Shum. Asymptotically admissible texture synthesis. *Proceedigns of International Workshop on Statistical and Computational Theories of Vision*, 2001.

[140] R. Baeza Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. *The 5th Combinatorial Pattern Matching, volume 807 of Lecture Notes in Computer Science*, pages 198–212, 1994.

[141] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. *Proceedings of the 3rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 311–321, 1993.

[142] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 103–114, 1996.

[143] S.C. Zhu, X. Liu, and Y. Wu. Exploring texture ensembles by efficient markov chain monte carlo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(6), 2000.

[144] S.C. Zhu, Y.N. Wu, D. Mumford, and Filters. Random fields, and maximum entropy: Towards a unified theory for texture modeling. *International Journal of Computer Vision*, 12(2):1–20, 1998.