Fábio Fabris

A novel cooperative algorithm for clustering large databases using sampling

Vitória - ES, Brasil

April, 2012

Fábio Fabris

A novel cooperative algorithm for clustering large databases using sampling

Dissertation presented for obtaining the M.Sc. degree on Informatics at the Federal University of Espírito Santo (UFES)

Advisor:
Flávio Miguel Varejão, Dsc.

DEPARTAMENTO DE INFORMÁTICA
CENTRO TECNOLÓGICO
UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO

Vitória - ES, Brasil

April, 2012

Dissertation of Master Research Project under the title of *"A novel cooperative algorithm for clustering large databases using sampling"*, defended by Fábio Fabris and approved in April, 2012, at Vitória, state of Espírito Santo, by the examination of the following professors:

Flávio Miguel Varejão, Dsc. Advisor
Universidade Federal do Espírito Santo - UFES

Hélio Barbosa Dsc. Invited Outside Member
Laboratório Nacional de Computação Científica

Arlindo Gomes de Alvarenga, Dsc. Invited Member
Universidade Federal do Espírito Santo - UFES

Alexandre Loureiro Rodrigues Invited Member
Universidade Federal do Espírito Santo - UFES

Abstract

Clustering is a recurrent task in data mining. The application of traditional heuristics techniques in large sets of data is not easy. They tend to have at least quadratic complexity with respect to the number of points, yielding prohibitive run times or low quality solutions. The most common approach to tackle this problem is to use weaker, more randomized algorithms with lower complexities to solve the clustering problem. This work proposes a novel approach for performing this task, allowing traditional, stronger algorithms to work on a sample of the data, chosen in such a way that the overall clustering is considered good.

Resumo

Agrupamento de dados é uma tarefa recorrente em mineração de dados. Com o passar do tempo, vem se tornando mais importante o agrupamento de bases cada vez maiores. Contudo, aplicar heurísticas de agrupamento tradicionais em grandes bases não é uma tarefa fácil. Essas técnicas geralmente possuem complexidades pelo menos quadráticas no número de pontos da base, tornando o seu uso inviável pelo alto tempo de resposta ou pela baixa qualidade da solução final. A solução mais comumente utilizada para resolver o problema de agrupamento em bases de dados grandes é usar algoritmos especiais, mais fracos no ponto de vista da qualidade. Este trabalho propõe uma abordagem diferente para resolver esse problema: o uso de algoritmos tradicionais, mais fortes, em um sub-conjunto dos dados originais. Esse sub-conjunto dos dados originais é obtido com uso de um algoritmo co-evolutivo que seleciona um sub-conjunto de pontos difícil de agrupar.

Acknowledgements

This work would not be possible without the support and guidance, direct or indirect, of some individuals that in a way or another helped in the realization of this work.

First and foremost, Dsc. Flávio Miguel Varejão, whose commitment and willingness to boost the research quality of the University made this work possible.

Dsc. Alexandre Loureiro Rodrigues, whose statistical knowledge and helpful insights greatly improved the quality of this work.

Undergraduate student Estevão Costa for the help in executing the experiments.

ESCELSA company, in special, Rodrigo Marin Ferro for the availability and disclosing data used in tests in this work.

Last but not least, to my family, including Caroline Rizzi, whose patient and meticulous reviews of the texts greatly helped the conclusion of this work.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Data Clustering is the task of assigning data points to groups (called clusters) so that similar objects belong to the same group given a similarity metric. This task is an *unsupervised technique*, because it does not depend on the actual labels of the data points to build the underlying organization. Grouping similar objects is a recurrent problem when dealing with analysis of large sets of data. In fact, grouping continuous data points plays a fundamental role in *Knowledge-Discovery in Databases* (KDD). The clustering task is essential in many relevant areas of human knowledge, e.g.: data visualization (SHERLOCK, 2000), bio-informatics (SHERLOCK, 2000; BENSMAIL et al., 2005), load balancing (MARZOUK; GHONIEM, 2005) and image processing (LIEW; YAN, 2003).

The general clustering task may be either *hard* (KAUFMAN; ROUSSEEUW, 2005), *overlapping* (BANERJEE et al., 2005) or *soft* (GUSTAFSON; KESSEL, 1978). Soft clustering (or fuzzy clustering) allows for points to belong to many different clusters at the same time with a certain degree of pertinence. Overlapping clustering allows for a point to belong to two or more clusters at the same time. In hard clustering, all points belong to one and only one cluster at a given time. This work deals only with the hard clustering version of the general clustering task. This variation is the most common incarnation of the clustering problem in real-world applications.

Algorithms that deal with hard clustering may be divided in two big groups: *partitioning* and *hierarchical* (KAUFMAN; ROUSSEEUW, 2005). Partitioning clustering algorithms iteratively construct solutions by creating disjoint and complete regions on the search space and assign each point to a region. Hierarchical clustering algorithms iteratively join (or separate) the two most similar clusters of a given iteration into a single cluster (or two). Hierarchical algorithms tend be very sensitive on the data set size and are only viable for relatively small

clustering tasks. This type of clustering algorithm has the property that each cluster is always formed by two or more clusters, building a taxonomy of data points. This notion of hierarchy may be useful in some applications that wish to have different levels of data separation. For instance, an application could use this clustering technique for grouping similar documents together. The first two big groupings could be documents written in different languages, the divisions inside the group's language could be major subjects written in that language, and so on. Because of the scalability problems associated with traditional *hierarchical* algorithms, they will not be considered in the work, however, an adaptation of a hierarchical algorithm for large data sets will be examined.

1.1 Algorithms for hard clustering

This work will focus on tackling the hard clustering problem in metric spaces using the *Sum of Squared Errors* as the minimizing function.

There are many algorithms proposed in the literature to tackle the *hard clustering problem*. They are divided in *exact*, *approximate* and *meta-heuristic* algorithms. Exact algorithms usually implement some smart search strategy (e.g. branch and bound techniques (FRANTI; VIRMAJOKI; KAUKORANTA, 2002)). Approximate algorithms (OSTROVSKY; RABANI, 2000) usually guarantee an $(1-\varepsilon)$ approximation where ε is coupled to the complexity of the underlying algorithm. Meta-heuristics do not hold any quality guarantee and usually aim at solving the problem in viable times.

An exact clustering algorithm finds the optimal solution for a clustering problem. This solution is the partitioning that minimizes the *Sum of Squared Errors* (SSE) for a given number of clusters k and a set of points D. However, the hard clustering problem in metric spaces is *NP-Hard*, i.e., there is no known algorithm capable of finding an optimal solution in polynomial time on the base size and probably there will never be (ALOISE et al., 2009). This kind of solution is only applicable for very small data sets.

Similarly, approximate clustering algorithms guarantee that the cost of an approximate solution multiplied by an *Approximation Factor* ε is smaller than the optimum solution. The value of ε regulates both the quality of the solution and the running time of the algorithm. Large values for ε yield poor solutions but fast running times, while values closer to 1 result in optimal solutions and very large running times. Thus, these algorithms yield complexities that won't compute good solutions for relatively small data sets in practical times (OSTROVSKY; RABANI, 2000). Approximate Clustering algorithms have exponential complexity on ε, the closer

ε is to 1, the more costly the underlying approximation is.

Given these facts, the most common way to solve the generic clustering problem on medium-sized data sets is by using heuristic algorithms. This kind of algorithm guarantees neither optimal nor approximate solutions, but usually has polynomial complexity. There are many different heuristics, each one with its specific application niche.

1.2 Scope of this work

This work focuses on the application of clustering algorithms in large data sets (more than 10.000) and low dimension (less than 6). In these conditions, even heuristic algorithms with polynomial complexity, like the k-means algorithm, may struggle to finish execution in practical times and/or reasonable memory loads. This work will also perform experiments on smaller data sets to test how the algorithms will scale as the problems get more difficult to solve.

Given this fact, there are algorithms specially conceived to work in large data sets. They usually avoid scanning the database many times and rely on sophisticated data structures to query the data. Also, there are algorithms specialized in building a summary of the data, a set of points that represents well the whole data set.

This work examines a set of classical algorithms designed for large data sets. In addition, experiments are performed to compare them to a novel, co-evolutionary algorithm for solving the clustering problem in large data sets in a robust manner.

Currently, the use of co-evolutionary algorithms for solving optimization problems is ubiquitous (MICHALEWICZ, 1994; MICHALEWICZ; JANIKOW, 1996; BERGH; ENGELBRECHT, 2004; AUGUSTO; BARBOSA; EBECKEN, 2008; BARBOSA, 1996, 1999). The co-evolutionary approach is based on the natural behaviour of species in nature and employs the use of two or more populations competing or collaborating towards a common goal. In this work, a dynamic population (A) of natural numbers that selects which points to consider in a given dataset and a population (B), of traditional clustering algorithms, that shall evaluate the quality of the points selected by population A. The population A should be capable of evolving to a set of relevant points and the population B should provide fast and distinct solution for them. The more heterogeneous the algorithm in population B is, the more unlikely it is for the algorithm to "get stuck" in a local minima. It is expected that the overall result will be a fast and robust clustering algorithm for large data sets.

The comparison of time and clustering error of the methods will be carried in 8 classical

data sets present in the literature and one unique real-world dataset. A statistical analysis test will be performed to check if there are significant differences between the evaluated algorithms.

1.3 Organization

The remainder of this dissertation is organized as follows: Chapter 2 consists on a revision of related work regarding clustering of large data sets, Chapter 3 contains the formal definition of the clustering problem and a description of the traditional clustering methods for large data sets, Chapter 4 describes the novel approach, Chapter 5 exposes the set up of the experiments, special considerations and results and finally, in Chapter 6 the conclusions are drawn.

Chapter 2

Related Work

Clustering is basically the grouping of non-labeled data in a way that minimizes a given quality measure or, in other words, the *unsupervised automated learning* of data distribution in different clusters (JAIN; MURTY; FLYNN, 1999). Unsupervised learning is the extraction of useful information from non-labeled data. It is the opposite of *supervised learning*, that builds a discriminative model based on pre-classified data. The model is then used for classifying unlabeled instances into the previously defined groups.

The objective of creating these clusters of unlabeled instances is to extract useful information from the data. This information is often hidden when the data points are scattered with no clear separation. By analysing the formed groups, one can gain insight over an otherwise apparently random point distribution. For instance:

- In marketing, for grouping similar clients and analysing their consumption profile (PUNJ; STEWART, 1983).
- In many image processing applications, like *color image quantization* (finding the most important colors of an image for building a good pallet) of images for *lossy compression* (XIANG; JOY, 1994).
- For image segmentation, the automatic division of images into similar groups (XIA et al., 2007).

The problem of finding clusters in large data sets with sparse points is widespread in the literature, for example:

- *Text/web mining*, for clustering similar web sites in categories (BEIL; ESTER; XU, 2002).

- *DNA clustering of large DNA sequences*, for finding similar chains of DNA to organize genetic information (RUSSELL et al., 2010; EDGAR, 2010).
- *Astronomical data processing*, for finding clusters of celestial bodies (TANG et al., 2008).

The previous examples often require clustering of hundreds of thousands of data points.

Each application niche develops its own algorithms for dealing with their particular problem. For instance, DNA data is mostly discret and multidimensional, thus, it is natural that specific algorithms must be developed for dealing with this kind of data. Astronomical clustering algorithms often have to deal with points with an enormous number of numeric attributes, thus, once again, special care must follow while designing algorithms for dealing with this data. The goal of this work is to develop a generic algorithm for large spatial datasets, not bound to specificities of any particular field of interest. Thus, this work will not go into much detail on algorithms that are too specific to a given area.

This work deals only with data points in *regular metric spaces*, not considering *non-metric spaces* (vectors with one or more nominal attributes changing from one category to another, without any clear enumeration, like colors, names or shapes). Although not being the major field of application, vast literature is available for clustering data in non-metric spaces. For instance, the *SCLUST (Symbolic Clustering)* algorithm deals with clustering of exclusive symbolic data on large datasets (LECHEVALLIER; VERDE; CARVALHO, 2006). There are also algorithms for dealing with large datasets with mixed attributes. For instance, (HE; XU; DENG, 2005) proposes a scalable algorithm for clustering large datasets containing symbolic and numeric attributes.

Also, this work will focus on algorithms for large data sets. This kind of algorithm must be efficient, i.e., avoid the need of scanning the whole data set multiple times. Complexities greater than quadratic time on the size of the input must be avoided at all costs. Exponential complexities are almost always prohibitive. Also, it is desirable that the algorithms possess some kind of mechanism for adapting themselves to the available hardware resources and time constraints. Classical tree-building algorithms for spatial data are often used to assist the summarization (adapting the available resources to the problem) and fast region query strategies (for reducing the complexity of common spatial operations when the data sets are large, assisting the clustering process by increasing the number of points that may be processed in practical times).

The next Section focuses on reviewing works that deal with the indexing and summarization of points in large data sets.

2.1 Fast Data Indexing and Summarization

There are three types of special data structures in the basis of many clustering algorithms. These structures are designed for allowing the manipulation of large data sets by extracting important information in an efficient way. The first important type of data structure is the *k-neighborhood* spatial index algorithms. These algorithms perform fast *k-Nearest Neighborhood Queries*, or *k*-NN queries, that are responsible for a fast retrieval of the *k* neighbors of a given point.

Algorithms for *k*-NN queries are classified in 5 major categories: simple *k*-NN queries (BERCHTOLD et al., 1998; CHEUNG; FU, 1998), approximate *k*-NN queries (ARYA et al., 1998; CIACCIA; PATELLA, 2000), reverse NN queries (KORN; MUTHUKRISHNAN, 2000), constrained *k*-NN queries (FERHATOSMANOGLU et al., 2001) and join *k*-NN queries (HJALTASON; SAMET, 1998). Simple *k*-NN queries deal with actually finding the *k* nearest neighbors of a given point. Approximate *k*-NN queries find the best neighbors with some associated probability. Reverse *k*-NN queries find the set of points that have the given point as their *k*-nearest neighbor. Constrained *k*-NN queries return the closest *k* neighbors of a given point inside a polygonal area. Finally, *k*-NN join algorithms return the *k* closest points of a *set* of query points.

This work focuses on *simple k-NN queries*. The most commonly used data structure for this task is the *KD-tree*. A *KD*-tree works by first building a tree containing all points of the data set in a way that searching of the *k*-nearest neighbors may be performed quickly by pruning paths that will not lead to a set of closest points (LIU; LIM; NG, 2002). This data structure was greatly studied by (MOORE, 1991) and was designed for retrieving the *k* closest neighbors of a given point in $log(n)$ time.

The second important data structure is the *CF-tree* (ZHANG; RAMAKRISHNAN; LIVNY, 1996). The *CF-tree* builds a summarized representation of the data set, scanning the data only once. This is important if the amount of data to process is beyond the capabilities of the hardware being used. In just one pass through the dataset, the *CF-tree* is capable of building a summarized, yet representative, version of the data. If more passes are performed, it is capable of improving even more the quality of the summarization. This procedure may be seen as a pre-clustering of the data, creating clusters in a way that a version of the whole dataset may fit in the memory at the same time allowing the use of traditional algorithm designed for smaller data sets.

The third important data structure developed for dealing with large data sets is the R-tree

(GUTTMAN, 1984). This data structure aims to retrieve efficiently all the points in a rectangular region of the metric space. This kind of spatial query is called *window indexing*. These tree algorithms are very useful for dealing with large datasets where scanning all points for answering a query is prohibitive.

Another valuable approach to reduce the complexity of the data is to employ feature selection, reducing the dimensionality of the data. There are cases where some attributes have strong correlation with others, implying that removing one of them from the dataset affects very little the final clustering decision. There are many methods for reducing dimensionality in the literature: Sequential Forward Selection, Sequential Backwards Selection (PUDIL; NOVOVICOVA; KITTLER, 1994), Principal Component Analysis (WOLD; ESBENSEN; GELADI, 1987) and A-Priori algorithms (AGRAWAL; SRIKANT, 1994) are among the most used. This approach, however, will not be considered in this work; this work considers that all features are equally relevant.

The invention of these data structures allowed the development of many clustering algorithms for large data sets. The next Section exposes the most relevant algorithms present in the literature.

2.2 Classical algorithms for solving the k-means problem in large data sets

The NP-Hard k-means problem consists in grouping points of a data set in k clusters. The grouping must minimize the overall *SSE* error between points of clusters and their respective centroids. Exact algorithms for solving this problem are exponential with complexity of $O(n^{kd})$ (d being the dimension of the data set, n the number of points and k the number of clusters), and thus impractical even for small data sets. In a tentative for reducing this large complexity, a number of polynomial time approximation schemes have been developed (VEGA et al., 2003). Many conclusions may be extracted from this work, however, when the approximation factor is close to 1, the degree of polynomial complexity is too large in k, yielding unpractical algorithms even for small dimensions and data sets sizes.

The most famous algorithm for dealing with the k-means problem is called the *Loyd Algorithm* (MACQUEEN, 1967) (or simply the *k-means algorithm*, due to its ubiquity). The k-means algorithm is widely used in the literature with very good results, its use is so spread that a survey of data mining techniques stated: *"is by far the most popular clustering algorithm used in scientific and industrial applications"* (BERKHIN, 2006).

The k-means algorithm, however, has particularities that restrict its use for large data sets. The first problem is the running time of the algorithm. Although fast in small data sets, the algorithm does not scale well when the number of points increase. With large data sets the use of this algorithm is unfeasible due to its n^3 complexity in most cases and super-polynomial in the worst case (ALSABTI; RANKA; SINGH, 1997; ARTHUR; VASSILVITSKII, 2006). Also, the algorithm is not designed for saving memory. In its traditional form, all points must remain in memory at the same time and multiples scans in the data are necessary. The k-means algorithm is sensitive to the initial, randomized, selection of centroids.

The dependence of the final result on the starting seeds is a well-known problem of the k-means algorithm. This issue is even more problematic when dealing with large data sets, since a bad initial set of clusters greatly slows down the convergence time of the algorithm (ARTHUR; VASSILVITSKII, 2006). Some variations of the original k-means algorithm were devised for dealing with the initialization problem, in particular the k-means++ algorithm (ARTHUR; VASSILVITSKII, 2007) initializes the centroids by sampling them inversely proportional to the distance between points and the closest centroid chosen so far (the first centroid is chosen at random). This improves convergence times of the algorithm, however the very initialization procedure is quite costly and impractical for large databases. There are many other initialization procedures that aim to select good points for k-means (NA; LOZANO; NAGA, 1999). All of them solve the problem of initial sensitivity, but introduce a scalability problem, since initialization procedures tend to be costly. Even with all these restrictions, it is still possible to apply variations of the k-means algorithm to large datasets with the use of the previously introduced data structures, although the final results are not always satisfactory.

The first attempts to elucidate the problem of clustering large data sets was solving the k-medoids problem. The k-medoids problem sets the centroids of the clusters to actual points of the data set, not arbitrary ones, like the k-means algorithm. This restriction simplifies the algorithms by restricting the possible distance queries to a restricted set of points.

One of the first k-medoids clustering algorithms proposed for large data sets was the *CLARANS* (Clustering Large Applications Based on Randomized Search) algorithm (NG; HAN, 2002). This algorithm was inspired by the *PAM* (Partitioning Around Medoids) and *CLARA* (Clustering Large Applications) algorithms (NG; HAN, 2002). It performs a very simple random search in a graph, randomly selecting neighbors of a given solution and checking if the new solution is better than the old one, with the possibility of accepting few worst solutions before falling back to the previous best solution found. This simple idea turned out to be both fast and effective, and it is one of the major algorithms for solving the problem of clustering large dataset until

today, being widely used as a validation algorithm in benchmarks of the literature (LUO et al., 2008).

CLARANS works well in large datasets because it builds the solution iteratively, i.e., a solution is available early, and continually improved. Also, the computation of the error has an effective optimization that avoids scanning the whole dataset in most cases. However, the *CLARANS* algorithm suffers from a known curse of many heuristic algorithms, tending to fall easily into a local minima and may return not-so-good results.

Shortly after the publication of the seminal paper describing *CLARANS*, (SANDER et al., 1998) proposed the *DBSCAN* algorithm for clustering large spatial datasets. The DBSCAN algorithm uses the *R-tree* structure to implement region queries in a fast manner. However, it requires that all points are stored in memory at the same time, which may be prohibitive in some cases. The idea of DBSCAN is that isolated, high density regions, must belong to the same cluster. This implies that it is not possible to choose the number of clusters directly. This may be useful in applications where the number of clusters is unknown but it is not advantageous if the number k of cluster is known *a priori*.

(ANKERST et al., 1999) proposed a generalization for the *CLARANS* algorithm called *OPTICS* (Ordering Points To Identify the Clustering Structure). It allows an automatic adaptation of the densities in different regions of the data set, i.e., the density threshold for grouping different regions of the data set may be different. This generalization resulted in a slightly more costly algorithm but much more robust. The *OPTICS-OF* algorithm is an extension of the original algorithm design for finding *outliers* (distant points with no clear cluster pertinence that greatly increase the overall error). (ACHTERT; BOHM; KROGER, 2006) proposed the *DeliClu* algorithm, an improvement over the original *OPTICS* algorithm for reducing the number of parameters from two to only one. The problem of not being possible to set the number of clusters, however, still remains.

Another approach for dealing with large data sets is adapting classical algorithms for large data sets. (KANUNGO et al., 2002) proposed a combination of a local search algorithm with the k-means algorithms for solving the clustering problem in large data sets. Also (LIU; LIM; NG, 2002; YAO; LI; KUMAR, 2010) proposed adaptations for the k-means algorithm for dealing with large data sets. These approaches reduce the complexity of the problem by selecting the most relevant attributes for the clustering procedure.

More recently, (LUO et al., 2008) devised a way to use the classical *k-means* algorithm for clustering large datasets. The idea is to first reduce the dimensionality of the dataset by selecting the most relevant features by using an *a priori*-based algorithm that finds sets of correlated

variables. After that, the *k*-means algorithm runs in a sampled subset of the data to find a good set of centroids. Finally these centroids are used to cluster the whole dataset. The idea is that a good selection of centroids drawn from the sample reduces the number of iterations needed for the convergence of *k*-means.

2.3 Co-evolutionary algorithms and cluster ensembles

This work will use a co-evolutive algorithm for solving the clustering problem. This kind of algorithm is inspired in the field of game theory, where two agents compete until they find the *Nash Equilibrium* of the game. (FICICI, 2004) discusses the roots of co-evolutionary algorithms, their application and variations.

The use of co-evolutive algorithms is common when one needs to optimize two distinct sets of parameters with a coupled fitness function. The sets of parameters may be competing, i.e., while one set tries to minimize the objective function, the other tries to maximize it (competitive co-evolution); or cooperating, i.e., both populations try to optimize the same objective function. Much work has been published with this approach, in particular in function optimization with restrictions. The most common approach is to encode the parameters of the function as individuals of one population, that tries to optimize the objective function, and the other population as the multipliers of the restrictions. The objective function is encoded as the Lagrangian of the objective function and its restrictions. Many works use this approach to solve complex design problems with restrictions (POON; MAHER, 1997; MAHER; POON; BOULANGER, 1996).

(BARBOSA, 1996) uses this idea coupled with *Genetic Algorithms* for solving many restricted optimization benchmark functions with great success. Also, (AUGUSTO; BARBOSA; EBECKEN, 2008) developed a competitive co-evolution algorithm that iteratively selects the most difficult instances to classify and the most efficient algorithms, each taken from specific populations.

Since co-evolution is designed for solving the generic optimization problem with dual objective functions, this work applies an idea based on the work of (AUGUSTO; BARBOSA; EBECKEN, 2008), creating a population for choosing “hard” points for the clustering process and another population of classical algorithms for solving the clustering process. The goal is to find the set of points that are most relevant for the final clustering process and the best possible solution for this set of points.

There are some works in the literature that couple co-evolution and clustering, for instance, (CHAKRABARTI; KUMAR; TOMKINS, 2006) uses a co-evolutionary clustering algorithm

for minimizing both the clustering error and the result quality of the algorithm. In (HE; WANG, 2007) the author uses co-evolution for selecting a good set of attributes for the clustering algorithm using co-evolution. In (DHILLON, 2001) the author uses cooperative clustering for grouping documents and the words that form them in a cooperative and simultaneous manner. These approaches differ greatly from the one that will be exposed in this work.

The next Chapter defines the clustering problem in detail and gives the description of four classical algorithms for dealing with large data sets.

Chapter 3

Traditional Clustering Algorithms

For verifying the performance of the proposed clustering algorithm, four classical approaches for dealing with large data sets were tested. This Chapter defines the underlying clustering problem and the classical algorithms used for solving it.

From this point forward, *clustering* is defined as follows:

Definition 1. Clustering is the restricted discret optimization problem of dividing n spatial points $\bar{x_i} \in \Re^d, i \in [1..n]$ of a set X, in k complete and disjoint sets (clusters) $C_i, i \in [1..k]$. This division must minimize the *Sum of the Squared Errors (SSE)* of all points with respect to the *cluster centroid*. The centroid of a cluster i, $\bar{c_i}$ is the mean of all points of cluster i.

More formally, if $X = \{\bar{x}_1, \bar{x}_2, \bar{x}_3, \ldots, \bar{x}_n\}$ is a set of points, $X \subset \Re^d$, then the clustering result must adhere to the following restrictions:

$$\bigcup_{i=1}^{i<=k} C_i \equiv X \tag{3.1}$$

$$\bigcap_{i=1}^{i<=k} C_i \equiv \emptyset \tag{3.2}$$

And minimize the following expression:

$$SSE(c) = \sum_{i=1}^{i<=k} \sum_{j=1}^{j<=S_i} (||C_{i,j} - \bar{c_i}||_2) \tag{3.3}$$

Where $C_{i,j}$ is the j-th point of the i-th cluster and S_i is the size of cluster j. In addition, $||\bar{t}||_2$ represents the euclidean norm of the vector $\bar{t} \in \Re^d$.

This work uses the evaluation metric defined in Equation 3.3 to assess the quality of the

considered candidate solution. There are many different metrics for measuring cluster quality. The majority of them depends on a subjective human feeling of whether one clustering result is more "pleasant" to the eye than another. The paper (AMIGÓ et al., 2009) presents some approaches for building a clustering evaluation metric that is independent of the number of clusters and makes sense for the human eye. The problem with these metrics is that some different clustering techniques are better than others in minimizing certain evaluation criteria. Also, when dimensionality increases, it becomes hard to verify the quality of those techniques. So, to avoid bias towards a certain clustering algorithm, this work uses only the *SSE evaluation metric*. This simple metric is widely spread in the literature and universally accepted as a valid approach for evaluating clustering algorithms.

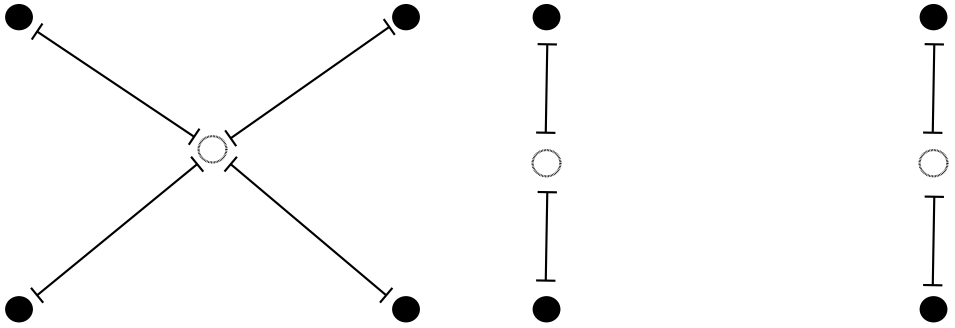
The number of clusters to be created (k) is also a very sensitive experimental parameter, especially if one is using the *SSE* clustering evaluation metric. The more clusters the algorithms build, the smaller can be the *SSE* of the final clustering result. This fact is easily understood by analyzing the Figure 3.1. If one splits any cluster in two, the overall clustering *SSE* will decrease. Imagine the extreme case that each point defines one cluster ($n = k$), in this scenario the centroids of all clusters will be themselves, and the overall *SSE*, zero. Thus, the bigger k is, the smaller is the clustering *SSE*. Given this fact, it makes no sense to compare algorithm runs with different values for k during experimental evaluation. For a given data set, this work will use a fixed value for k, if it is provided by the literature. If it is not, a range of arbitrary values will be tested and the comparison carried out independently for each k.

Although the *SSE* can potentially decrease when k increases, this is not always the case when using meta-heuristics. Sometimes the algorithm cannot find a better solution when k increases, even though it certainly exists. When k increases, the clustering problem becomes more difficult. For this reason, sometimes the *SSE* increases with k because the algorithm cannot deal well with the augmented complexity of the problem for a given computational budget.

In the next Section, classical algorithms for solving the traditional clustering problem for large data sets are presented.

3.1 Traditional Methods for Clustering Large Data sets

This work focuses on the problem of clustering large data sets. Here "large" means sets X that are not small enough to fit in main memory at the same time, requiring some kind of summarization technique or other mechanisms for fast calculation of the necessary values during algorithm execution.



(a) Before split (b) After split

Figure 3.1: Example to demonstrate that the *SSE* of a clustering response decreases as k increases. It is always possible to decrease the *SSE* of cluster by introducing a new centroid. Thus, the bigger the k, the smaller the *SSE* may be.

Many methods for clustering large data sets were proposed throughout the years. To test the *Coevolutive Clustering Algorithm for Large Data Sets (COCLU)* proposed in this work, classical and novel techniques are applied in 8 classical benchmark data sets and 1 unique data set. Those data sets vary in size from average to large, the behavior of the algorithms in function of the data set sizes are analyzed in respect to both response-time and result-quality. It is expected that the proposed algorithm achieves best results in large data sets.

Next, a brief explanation of the applied clustering algorithms and the underlying data structures used for implementing them are presented. The following Sections present the pseudo-code, highlight the most import features and point out the limitations and caveats.

3.1.1 CURE

The *CURE* (GUHA; RASTOGI; SHIM, 1998) algorithm was developed especially for dealing with large data sets with noise. The authors claim that the algorithm is capable of properly clustering data even if the cluster format is non-trivial, i.e., has non-spherical shapes and/or a spread point distribution. This algorithm has a strong mechanism for dealing with outliers and is easily modifiable for dealing with large data sets.

The *CURE* algorithm is a hierarchical agglomerative clustering algorithm that achieves both scalability and robustness by selecting a fixed number of representative points to perform its calculations. Hierarchical agglomerative clustering algorithms begin by considering each data point as a separate cluster and then merge the closest pair into a single cluster, repeating this procedure until the desired number of clusters is reached. So, if one needs k clusters in a data set of size n, these algorithms merge clusters $(n-k)$ times. It is clear that if n is big and k is small, many merging operations need to be performed. Thus, merging operations must be

fast for this algorithm to be useful for clustering large data sets.

Hierarchical agglomerative clustering algorithms require the calculation of distances between clusters many times during the clustering process. There are three common ways for calculating this distance: minimum pairwise distance between points, maximum pairwise distance between points and average pairwise distance between points.

The *CURE* algorithm uses the first option: minimum euclidean distance between pair of points of clusters T and K , denoted by $||T,K||_2^{min}$ and defined in the following Equation:

$$||T,K||_2^{min} = min(||T_i - K_j||_2), \forall i \in [1 \dots |T|], j \in [1 \dots |K|] \tag{3.4}$$

Where $|S|$ denotes the cardinality of set S.

The calculation of this distance is computationally expensive when the number of points of each cluster is large, leading to prohibitive processing times in large data sets. The cost of calculating 3.4, $O(|T||K|)$ may be reduced to $O(r^2)$ by using a subset of r representative points for each cluster.

The *CURE* algorithm proposes an approximate way for calculating the distance defined in Equation 3.4, decreasing the complexity of the overall algorithm. The idea is to use a sample of representative points of the cluster for calculating their distance. *CURE* only evaluates a fixed-sized set of representative points for the merging decision. This reduces the complexity of the overall algorithm from $O(n^3)$ to $O(r^2 log(n))$ (GUHA; RASTOGI; SHIM, 1998), where r is the amount of representative points.

For the algorithm to work properly, the representative points must capture the overall shape of the clusters and must be descriptive enough for a good approximation to the inter-cluster distance. To construct a good set of representative points, the algorithm uses a simple method for selecting representative points that will be described in the Algorithm 1 and explained afterwards.

When dealing with large data sets with even greater size, the *CURE* algorithm also uses the idea of randomly sampling partitions of the data, clustering them and merging the resulting clustering solutions. This approach will be explained in Algorithm 2.

Pseudo-code of the *CURE* algorithm

The pseudo-code of the algorithm is presented in Algorithm 1.

Algorithm 1 The *CURE* Algorithm

procedure CURE(The data set X, The number of clusters k)
Create a min-heap H
for all $\bar{x}$ in X **do**
Set $\bar{x}.closest$ to the closest neighbor of $\bar{x}$
Set $\bar{x}.representatives$ to $\bar{x}$
Insert $\bar{x}$ in H using the distance of $\bar{x}$'s closest neighbor as the heap criterion
end for
Create a kd-tree T, containing all n points of the data set X
while $|C| \neq k$ **do**
Extract v from H, the cluster on the top of the min-heap
Extract $v.closest$ from H, the closest cluster of v
Merge clusters v and $v.closest$, using algorithm 2, creating a new cluster w
Insert w in the min-heap
Set $w.closest$ to a point in infinity
Remove the representatives of v and $v.closest$ from the kd-tree
Insert the representatives of the new cluster w in the kd-tree T
for all cluster t in H **do**
if $||w, w.closest||_2^{min}$ is greater than $||w,t||_2^{min}$ **then**
There is a cluster closest to w than the current, make $w.closest = t$
end if
if $t.closest$ is either v or $v.closest$ (the current closest cluster of t changed) **then**
if $||t,w||_2^{min}$ is greater than $||t,t.closest||_2^{min}$ **then**
The new closest cluster must be found using the representative set
Query the kd-tree for the closest points of t:
$t.closest = closest(T,t,||t,w||_2^{min})$
else
The new closest cluster in relation to t must be w, i.e., $t.closest = w$
end if
Relocate t in the heap
else if $||t,t.closest||_2^{min}$ is greater than the distance $||t,w||_2^{min}$ **then**
Update the current closest cluster of t to w, i.e., $t.closest = w$
Relocate t in the heap
end if
end for
Insert the new cluster w in the heap H.
end while
return the clustering result.
end procedure

Line 2 of the algorithm describes the initialization of a *min-heap* structure for storing the current minimum pairwise distances of all clusters. Min-Heaps are useful for retrieving (and removing) the smallest value of a set of points in $O(log(n))$ time, in this case, the most similar clusters.

The loop presented in line 3 describes the initialization process of the algorithm. It creates n clusters with one element each and then finds the nearest neighbors of all clusters in $O(n^2)$ time.

After that, it inserts the resulting clusters in the *min-heap* in $O(nlog(n))$ time using the distance of the closest cluster as the heap criterion. After the initialization process, the first heap-deletion operation will remove the cluster v, whose distance to its neighbor *v.closest* is the smallest of the data set.

Line 12 refers to the merge operation, the core of the CURE algorithm, which is responsible for taking the two closest clusters, merging them and selecting a proper set of representative points. The merging procedure is presented in Algorithm 2.

After the clusters are merged, the algorithm updates the heap structures in the *for-loop* beginning on line 17, so that the next heap removal returns the proper pair of closest clusters. For implementing this step, it is necessary to find the points inside a base's region. The *kd*-tree is essential for this, since this data structure is designed for efficiently retrieving the points around a region of the space in a given radius.

The update procedure works by iterating over each element t of the heap H and performing the following operations.

1. Updating the closest cluster of the newly created cluster w (the if block beginning on line 18).

2. Verifying if the current closest cluster of t is either v or *v.closest* (the if block beginning on line 21). If this is the case, the closest cluster of t must be updated, because v and *v.closest* were merged into w. For this, the algorithm uses the *kd*-tree to find the points around t in a radius equal to $||t,w||_2^{min}$. At least the closest representative point of w will be returned. But the closest point may not be the closest representative point from t, since the change in representatives may result in another cluster being closest to t. In this case, $t.closest$ is updated. If the comparison fails, this means that the newly formed cluster is closer than the old closest cluster (the *else* clause (line 26)), so, it must necessarily be the new closest cluster from t, since no other cluster was created.

3. Relocating t in the heap in $O(log(n))$, i.e., performing the necessary "promotes" or "demotes" for maintaining the heap structure (line 29).

4. Updating $t.closest$. If the closest cluster to t is neither v nor $v.closest$, but the newly formed cluster w, it is closer than $t.closest$. Update $t.closest$ to w and update the heap (the if block beginning in line 30).

Algorithm 2 The CURE Algorithm, Merging Procedure

```
    procedure MERGE(Cluster v, Cluster u, The number of representatives c)
        Initializes the set of representatives, Rep = ∅
        Unite clusters v and u, w = v ∪ u (only the representatives)
        Calculate w.centroid, the centroid of the union of v and u
 5:     Initializes the set C with the most distant point of w in relation to the centroid
        for do i = 2...min(c, |w|)
            Set d_min to zero
            for each point p of w do
                if The minimum distance d, between p and C is greater than d_min then
10:                 Set d_min to d
                    Set p_max to p
                end if
            end for
            Add p_max to the set of representatives Rep
15:     end for
        Set the representatives of w to Rep, w.representatives = Rep
    end procedure
```

The algorithm 2 first unites the set of representative points of the clusters that are being merged (line 3) and calculates the centroid of the merged set w (line 4). Each iteration of the for loop (line 6) adds one more representative point into the set of representatives until the size of the set is c or there is no point left in the set w.

The procedure initializes the set of new representatives with the most distant point of w in regards to the calculated centroid (line 5). After that, for every other point p of set w, the distance between p and the current set of representatives, Rep, is calculated (line 9) using a distance metric defined in Equation 3.5. The farthest point from the current set of representatives Rep is added to it.

$$D(p, Rep) = min(||p, r||_2, \forall r \in Rep) \tag{3.5}$$

All the distance calculations between clusters are performed using only their representatives. The actual distance metric between two used clusters may vary. The authors suggest the use of the minimum of all pairwise distances between the points of the two clusters, as defined in Equation 3.5.

Finally, to speed up the CURE algorithm even more for dealing with large data sets, p random, independent partitions of equal size ($f = n/p$) are created from the original data set. The clustering process is performed in each partition until there are k clusters in each one of them. At this point, the algorithm finishes partitioning by running the *CURE* algorithm again on the pk clusters generated by the partial clustering phase. Algorithm 3 describes this algorithm.

Algorithm 3 The Adapted CURE Algorithm for larger data sets

procedure BIGCURE(Data set D, The number of clusters k, Distribution Factor f)
  Initializes $S = \emptyset$
  while $|D| > 0$ **do**
    Create set R, Randomly removing f elements from D (or $|D|$ if $|D| < f$)
    Run the CURE algorithm on R, retrieving the clustering result C. The clustering result is a set of clusters with their representative points
    Add the clustering result into set S
  end while
  Run the CURE algorithm on S, retrieving the final clustering result C_{final}
end procedure

The previous algorithms reduce the complexity of the overall clustering process from $O(n^2 log(n))$ to $O(f^2 log(f)r/f)$.

3.1.2 CLARANS

The CLARANS algorithm (NG; HAN, 2002)(Clustering Large Applications based on randomized search) is a randomized searching algorithm that aims to find k medoids (centroids that are points of the data set) that minimize the overall *SSE* cost of the clustering task. This algorithm combines ideas of two other classical algorithms: the *PAM (Clustering Around Medoids)* (WONG; LUK; HENG, 1997) and the *CLARA (Clustering LARge Applications* (WONG; LUK; HENG, 1997) algorithms.

PAM randomly selects k medoids from the whole data set and tries all possible exchanges of one medoid in the current solution, keeping the one that minimizes most the *SSE*. This algorithm is greedy and deterministic, it never backtracks once a decision is made and it always follows the best available path, even if this path leads to bad solutions in the future. The algorithm halts when no improvement is found in the current iteration. Those are characteristics of algorithms that tend to fall into local minima. The authors claim that this algorithm is only suited for data sets that have at most 100 points. This number is smaller than the vast majority of data sets available in the literature and very far from the actual requirements of real-world applications.

For improving the algorithm, a modification was developed: to cluster large data sets, the

CLARA algorithm takes a random sample of limited size of the original set and uses the *PAM* algorithm to find a solution for the simplified problem. The algorithm repeats this procedure a given number of times, always starting from a different set of sampled points, calculating the *SSE* with regards to the whole data set and keeping the best solution found so far. It relies on the fact that if this repetitions are performed many times, eventually a good set of representative points will be randomly selected.

The *CLARANS* is an improvement over *CLARA* algorithm, and may be explained with a simple graph abstraction: suppose a graph G, in which the nodes represent a possible medoid selection. Edges are valid transformations from one solution to another that exchange a single medoid. Figure 3.2 illustrates this concept. Two nodes are directly connected by an arc if they differ only by one medoid.

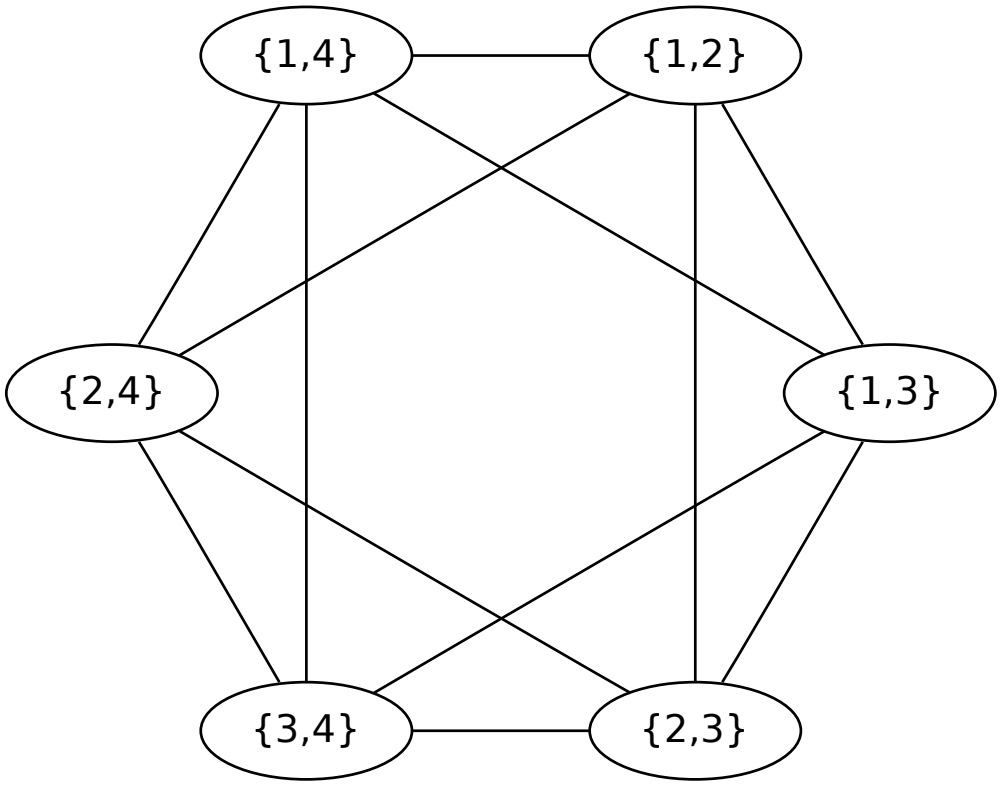


Figure 3.2: Example of a graph abstraction of the *CLARANS* algorithm. Vertices represent possible solutions (medoids) and edges, valid transitions. In this example, the data set size is $n = 4$ and the number of clusters $k = 2$.

CLARANS starts with a random node representing a solution and randomly visits its neigh-

bors, always exchanging the current solution with a random neighbor if it has a smaller SSE. Also, from the current best solution S the algorithm explores worst solutions until a given number of hops (*numLocal*) is reached. If no better solution is found after *numLocal* hops, the algorithm falls back to the best node found so far. This procedure is repeated until a given stopping criterion is reached, in this case, the number of iterations *maxNeighbors*. The ability to work in large data sets comes by the iterative nature of the algorithm, i.e., a solution is available quickly. Also, there is an optimization in the error calculation of neighbors solutions, i.e., there is an efficient way for calculation the error of a new solution given the error of the parent solution. Algorithm 4 describes the *CLARANS* algorithm.

Algorithm 4 The CLARANS Algorithm

```
procedure CLARANS(The data set X, The worst solution tolerance numLocal, Number
of iterations maxNeighbor)
    i = 0
    Set best to a random node of G (graph abstraction of Figure 3.2)
    while i < maxNeighbor do
        minCost = ∞
        Set the current search node cur to a random node of G
        j = 0
        while j < numLocal do
            if the SSE of cur is smaller than the SSE of best then
                Set the current best node to cur
                j = 0
            else
                j++
            end if
            Set the current search node cur to a random neighbor of best
        end while
        i++
    end while
    Return the best node, best
end procedure
```

This algorithm includes a smart calculation of the SSE difference between neighbors. This optimization is especially useful when the number of clusters is large. The intuitive idea of the optimized *SSE* update procedure is simple. Suppose that the algorithm creates a neighbor of the current solution by swapping the medoid O_m to O_p. For each other point O_j of the data set, we have four possible scenarios:

1. O_j belongs to the cluster defined by O_m and, after medoid swap, it does not change to O_p, but to a cluster defined by other medoid, O_2, increasing the clustering cost, i.e., $||O_j, O_p||_2 \geq ||O_j, O_2||_2$.

The following expression defines the clustering cost increase, C_1:

$$C_1 = ||O_j, O_2||_2 - ||O_j, O_m||_2 \tag{3.6}$$

Figure 3.3 illustrates the concept, after the swap, the closest medoid form O_j is O_2, not O_p.

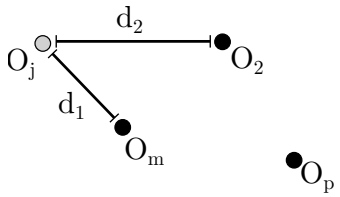


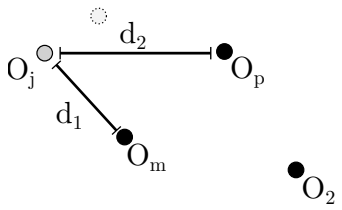
Figure 3.3: If the new medoid O_p is farthest than O_2, the cost of O_j will increase, $(d_1 < d_2)$

2. O_j belongs to the cluster defined by O_m and, after medoid swap, it changes to O_p, i.e., $||O_j, O_p||_2 \leq ||O_j, O_2||_2$. This exchange may reduce or increase the overall clustering cost.

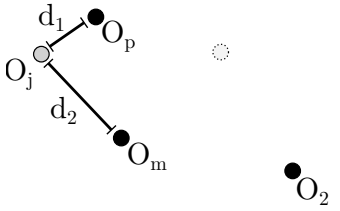
   The following expression defines the clustering cost difference, C_2:

$$C_2 = ||O_j, O_p||_2 - ||O_j, O_m||_2 \tag{3.7}$$

   Figure 3.4 illustrates the concept, after the swap, the clustering cost may or may not decrease.



(a) Cost of O_j increases $(d_1 < d_2)$



(b) Cost of O_j decreases $(d_1 < d_2)$

Figure 3.4: In both Figures, the new medoid O_p is closer to O_j than the medoid O_2. But in the first Figure, O_p is farther than O_m, yielding a greater SSE cost. In the second Figure, the opposite occurs, yielding a smaller SSE cost.

3. O_j belongs to cluster O_3 and, after medoid swap, it remains in that clusters, i.e., $||O_j, O_3||_2 \leq ||O_j, O_p||_2$. This exchange does not modify the overall clustering cost.

$$C_3 = 0 \tag{3.8}$$

   Figure 3.5 illustrates the concept, after the swap, the clustering remains the same.

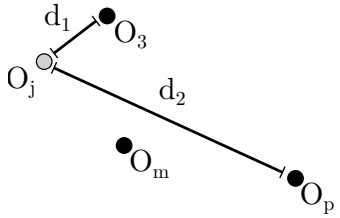


Figure 3.5: Medoid O_m is exchanged with medoid O_p, however, the new medoid is still far from O_j's medoid, O_3 ($d_1 < d_2$).

4. O_j belongs to other cluster O_3 and, after medoid swap, it exchanges clusters, i.e., $||O_j, O_3||_2 \geq ||O_j, O_p||_2$. This exchange always results in a decrease in the overall SSE, given by:

$$C_4 = ||O_j, O_3||_2 - ||O_j, O_p||_2 \tag{3.9}$$

Figure 3.6 illustrates the concept, after the swap, the clustering cost decreases.

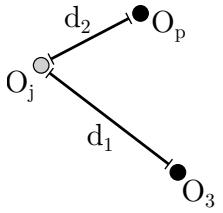


Figure 3.6: If point O_j is inside a cluster defined by a point other than O_m and exchanges clusters, then its cost certainly decreases ($d_2 < d_1$).

The clustering gain of a neighbor swap is given by the sum $\delta = \sum_{i<=4}^{i=1} C_i$. If the value is negative, then the neighbor has a better (smaller) *SSE* than the node that originated it. The SSE of the whole clustering result need to be calculated only once. When a new neighbor is created, one needs only to add the δ to the previous *SSE* value to get the next, using the four rules defined above.

The previous set of rules reduces the complexity of calculating the *SSE* of the new solution from $O(nk)$ (scan all points for each medoid to get the smallest distance) to $O(n)$ (scan all points and use the appropriate rule to get the value for δ.

3.1.3 BIRCH

BIRCH (Balanced Iterative Reducing and Clustering Using Hierarchies) is a clustering algorithm that takes into account resource availability while solving the clustering problem. Memory constraints may be used for tuning the algorithm, minimizing I/O cost. It requires

only one pass in the data set to cluster it. Additional scans to improve the quality of the clustering results are optional.

The advantages of this algorithm are many:

1. It is a local algorithm. Each decision of the cluster assignment for each point does not require scanning of the whole data set, just a limited neighborhood.
2. It has a mechanism for treating outliers and compressing high density regions of the space into a single point.
3. It adapts itself to the quantity of memory in the system, minimizing I/O cost.
4. It does not require the whole data set in advance, being able to generate partial solutions as new data arrives.

BIRCH relies on the use of a complex data structure called *Clustering Feature Tree (CF Tree)*. A *CF-Tree* summarizes large sets by building a *B-Tree*-like structure that contains in each node a *feature vector* consisting of important characteristics of all descendants of that node. The feature vector i is a triple $(N_i, \bar{X}_i, X_i^2)$ containing the quantity of all descendants N_i, the average point (centroid) of all descendants $\bar{X}_i$ and the sum of the squared points X_i^2. The amount of nodes in the *CF-Tree* regulates the memory usage and the time that posterior clustering algorithm will take to cluster the summarized data.

The construction of the CF-Tree follows the idea of the B-tree. Points are inserted into the closest nodes, and after the insertion the node may be split under certain conditions, increasing the number of nodes in the data structure. While building the CF-Tree, some conditions must be established for splitting a node. In (HARRINGTON; SALIBIáN-BARRERA, 2010) three conditions were proposed to improve the tree building processes firstly developed by (ZHANG; RAMAKRISHNAN; LIVNY, 1996). These conditions regulate the size and the quality of the CF-Tree and consequently the amount of memory used to summarize the whole data set (ZHANG; RAMAKRISHNAN; LIVNY, 1996).

The first condition is the *closeness* criterion, the maximum distance that a point must have in relation to the node centroid in order to be inserted in that node. If this distance is too small the tree will be sparser, summarizing less the data set.

The second condition is the *compactness*, the trace of the covariance matrix of the target node and the point to be inserted. The trace of the covariance matrix informs the variance (i.e. dispersion) of the cluster. This parameter complements the *closeness* criterion by reducing

sensibility regarding insertion order. If one uses only the *closeness* criterion, the following undesired case may happen. Suppose that we have n points $a_1, a_2, \ldots a_n$. If the points are in the same line and the distance between consecutive points is smaller than the distance between the new point and the current centroid, the new position of the centroid will shift towards the new point, allowing for distant points from the original point a_1 to belong to the same cluster.

The last condition is the *branching factor* that regulates the maximum amount of entries that each node may hold. The bigger the *branching factor* the smaller the tree height. This parameter should be smaller than the memory page size of the system that the algorithm is running, avoiding excessive paging.

The pseudo-code of the tree-building algorithm is presented in Algorithm 5.

Algorithm 5 The CF Tree Building Algorithm

procedure CF TREE(The data set D, The closeness parameter c, The compactness co, The branching factor b)
  for each x in D **do**
    Transverse the tree using the centroid of the nodes and find the closest leaf L in relation to x
    Insert x in the leaf L, updating the summary
5: **if** The insertion of x in the node L violates one of the three conditions **then**
      Split the node L creating two more nodes L_1 and L_2 using the farthest points of L as seeds and adjust the summary to reflect the descendants of L_1 and L_2 for the new nodes
      Insert the most distant node from L_1 and L_2 in the parent node
      if There is a overflow in the parent node **then**
        Recursively split the parent nodes until no split is necessary or until the root node is reached, in this case increase the height of the tree by one
10: **end if**
    end if
  end for
end procedure

The procedure for adjusting the summary (in line 4) is trivial: the feature vectors just need to be added. To split nodes (lines 6 and 9), the procedure is similar: the new parents only need to add the summary of its children.

Once the tree is built, any traditional clustering algorithm may be employed for clustering the summary of the data (the leafs of the CF-Tree). In this work we used the *k-means* algorithm to cluster the resumed data.

3.1.4 DBSCAN

DBSCAN (SANDER et al., 1998) is a classical clustering algorithm developed for large spatial data sets. This heuristic uses the idea of density for grouping points with arbitrary shapes in the same cluster. If a given area of the data set is relatively denser then its surroundings, this area is considered a cluster. Also, isolated points with density below a given threshold $MinPts$ are considered noise.

The algorithm uses the notion of *direct density reachability* and *density reachability*. Two points p_1 and p_2 are *direct density reachable* from one to another (with parameters ε and $MinPts$) if their distance is less than ε and there are more than $MinPts$ points in the neighborhood of point p_1. Therefore, two points p_1 and p_n are *density reachable* with parameters ε and $MinPts$ from one to another if there exists a sequence of *direct density reachable* $p_1, p_2, \ldots, p_n$ points that connects them. The set of points that are mutually *density reachable* are called *core points*. Figure 3.7 exemplifies the concepts discussed above.

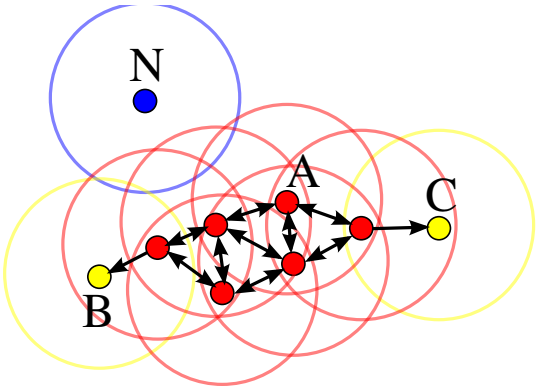


Figure 3.7: Example of a cluster formed by DBSCAN ($MinPts = 3$). The red region contains core points, i.e., points that are mutually *density reachable*. Points B and C are *density reachable* from A, however the opposite is not true. There is only one point in their neighborhood. Point N is regarded as noise since it is not *density reachable* and its neighborhood has less than $MinPts$ points.

Each point is visited at least once to have its neighborhood explored. The final clusters are formed with all sets of *density reachable* points.

The DBSCAN algorithm requires two parameters: the sampling radius ε that regulates the size of the neighborhood and the noise threshold $MinPts$. The pseudo-code of the algorithm is presented in Algorithm 6.

Algorithm 6 The DBSCAN Algorithm

```
procedure DBSCAN(The data set D, the sampling radius ε, the noise threshold MinPts)
    for each point x of the data set D do
        Visit x
        Retrieve N, the ε neighborhood of the point x
        if The size of N is less than MinPts then
            Mark x as noise
        else
            Find the set E of all density reachable points of x that were not visited yet and
create a new cluster with E
            EXPAND(x, N, ε, MinPts)
        end if
    end for
end procedure
```

Line 3.1.4 is implemented as follows:

Algorithm 7 The Expansion Procedure

```
procedure EXPAND(A base point x, The neighborhood of N, ε, MinPts)
    add x to a new cluster C
    for all point P' in N do
        if P' is not visited then
            mark P' as visited
            N' = regionQuery(P', ε)
            if |N'| >= MinPts then N = N joined with N'
            end if
        end if
        if P' is not yet member of any cluster then add P' to cluster C
        end if
    end for
end procedure
```

This clustering algorithm has one important particularity: there is no way to directly set how many clusters it will return. This parameter is implicitly adjusted by the density sampling radius ε. This fact may be advantageous if the average density of the clusters is known, however this is rarely the case. Also there is no clear indication on how to set this parameter with a "good" value, depending on the distribution of the data this parameter may vary greatly.

The experiments require strict control of the overall number of clusters k created at the end of the algorithms. Since there is no way to explicitly set k in this algorithm, a meta search over the values of ε was performed to compare effectiveness of the algorithms. A binary search algorithm defined in Algorithm 8 was used to search a good value for ε, since the function that defines k is uni-modal, if the desired number of clusters exists in the ε search space it will be found. The algorithm for finding k, given a search interval, is presented below.

Algorithm 8 Local search for finding the value for ε that generates k clusters

procedure LOCAL SEARCH(Beginning of the interval a, End of the interval c, Max iterations $maxIt$)
  Run DBSCAN with $\varepsilon = a$ and get the number of clusters generated, $cluA$
  Run DBSCAN with $\varepsilon = c$ and get the number of clusters generated, $cluC$
  if $cluA = k$ **then**
    Return a
  else if $cluC = k$ **then**
    Return c
  else if $cluA < k$ or $cluC > k$ **then**
    Return an error
  end if
  $i = 0$
  while $i < maxIt$ **do**
    $b = \frac{a+c}{2}$
    Run DBSCAN with $\varepsilon = b$ and get the number of clusters generated, $cluB$
    if $cluB = k$ **then**
      Return b
    end if
    if $cluB < k$ **then**
      $c = b$
    else
      $a = b$
    end if
  end while
  Return an error
end procedure

The algorithm presented in the Algorithm 8 implements a simple local search for finding a good value for ε. The algorithm assumes that the value of k in function of ε is a monotonically decreasing function, i.e. the greater the value of ε the smaller the value of k.

The procedure starts with a pre-defined interval $[a,c]$, supplied by the user, and retrieves the number of generated clusters for both $\varepsilon = a$ and $\varepsilon = c$. If the value found using $a(c)$ is smaller (greater) than the desired cluster amount k, the algorithm halts with error (line 9). The reason is that the first value of a (c) returns the biggest (smallest) number of clusters possible. If the biggest (smallest) value for the number of clusters is smaller (bigger) than the desired number, the algorithm won't be able to find the desired value, so the algorithm halts with error. If this occurs, the initial bounds must be reconfigured to a more adequate set of values. If one of these values happens to be on the desired set of values, the algorithm returns with success.

The algorithm proceeds by retrieving the number of generated clusters in the middle of the interval $b = \frac{a+c}{2}$ (line 14). If the correct number of clusters is found (line 15), the algorithm stops and returns b. Otherwise, it repeats the search procedure on intervals $[a,b]$ or $[b,c]$, depending on the value of *nCluB* (line 18).

This algorithm will find the value of k in a finite number of steps, if the interval is properly set and the algorithm is capable of finding such solution. In many instances of the performed experiments this algorithm could not found the proper value for ε, thus, invalidating some experiments for this algorithm.

3.1.5 R-Tree

In order to implement the DBSCAN algorithm properly, a way of indexing multidimensional spatial data efficiently using hyper-spheres in large data sets is necessary. The data structure *R-Tree* (GUTTMAN, 1984) is designed in a way that retrieves efficiently objects (points or rectangles) of a arbitrary rectangular region of the space. However, this algorithm requires a pre-processing stage whose runtime is significant.

An R-Tree is very similar to B-Trees, being a self-balancing structure. Its leaf nodes contain points of the data being stored. The structure is designed so that few nodes must be visited to return the nodes in a given rectangular region (Region Query). It supports random access queries and, although not strictly required in this work, has a dynamic nature. Points can be inserted and deleted with no extra cost once the structure is built.

According to (GUTTMAN, 1984), an R-Tree does not achieve good worst case performance, it can perform as slow as the naive approach, but has good real-world response times.

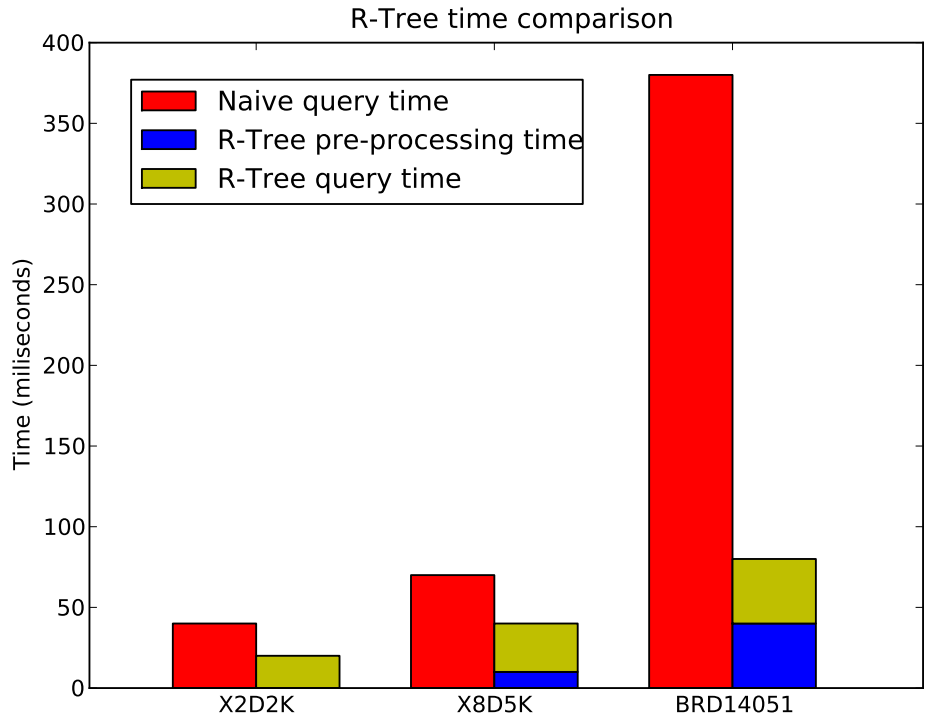


Figure 3.8: Naive X R-Tree runtime for the first three data sets

Also, the R-Tree does not behave well in high dimensional data sets.

In Figures 3.8, 3.9 and 3.10 it is shown the effectiveness of the R-Tree algorithm in a suite of tests using the benchmark data sets. Batch inserts and queries were performed using the R-Tree algorithm and the runtimes were compared to the naive approach of scanning all data points to verify, which points are in a given hypersphere.

This experiment consists in querying 1000 random regions for points in the benchmark data sets. The querying region is defined as follows: select two random points p_{r1} and p_{r2} of the space. The point p_{r1} will be the center of the hypersphere and the radius will be the distance between p_{r1} and p_{r2} multiplied by a random, uniform factor between 1.0 and 0.001.

From the bar plots it is clear that the R-Tree algorithm performs very well in low dimensional data sets. The only data set that the use of R-Tree is not advantageous is the *MiniBooNE_PID*. That is probably due to the high dimensionality of the instances (50 dimensions).

Finally, to analyze how the R-Tree algorithm processing time behaves when the data set scales, a second experiment was performed in a synthetic, low dimensional (2 dimensions) data set, following the same protocol of the previous experiment. The results are presented in Figure 3.11

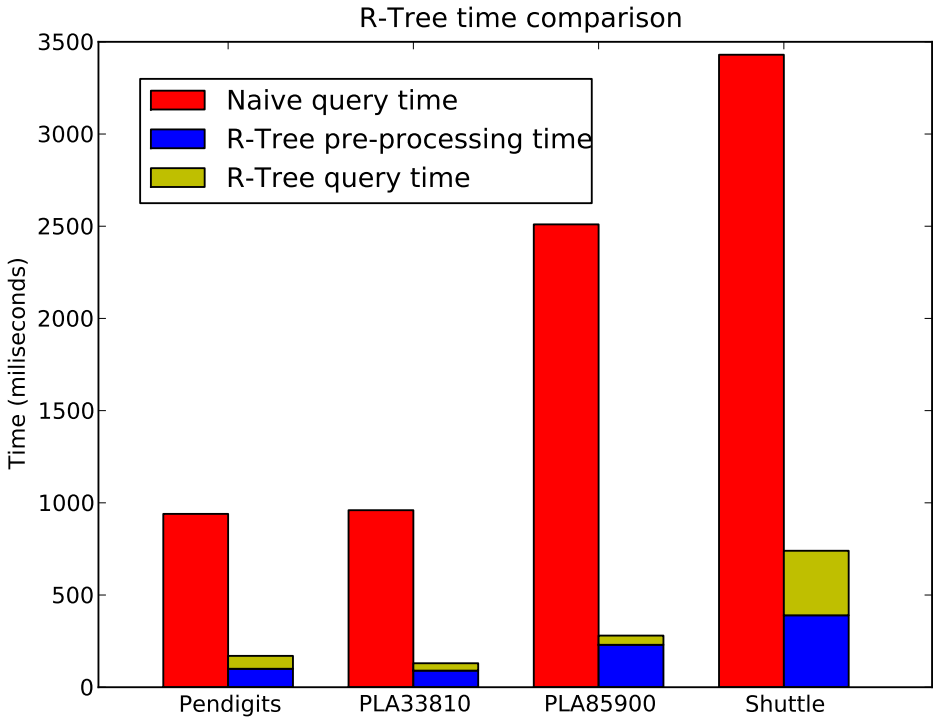


Figure 3.9: Naive X R-Tree runtime for the forth to the seventh data sets

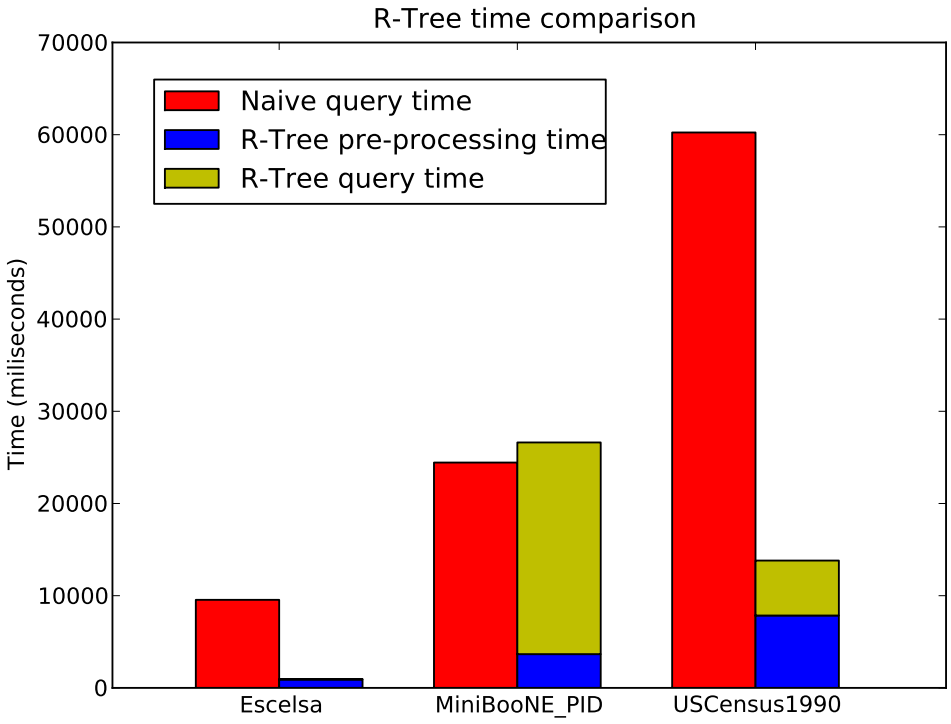


Figure 3.10: Naive X R-Tree runtime for the last three data sets

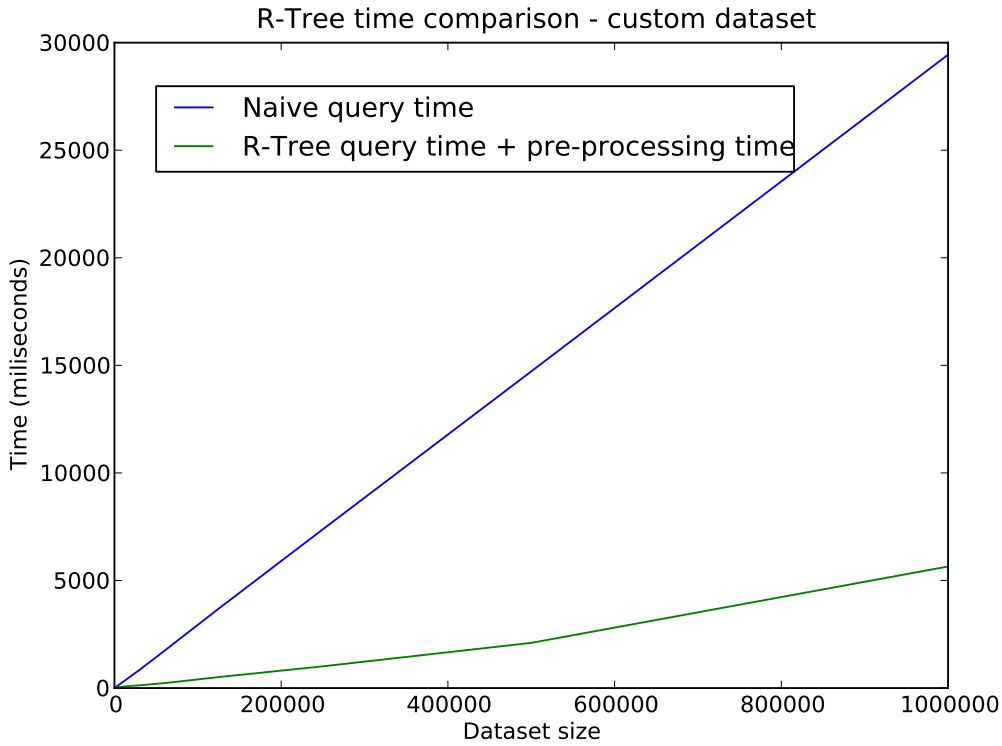


Figure 3.11: Time comparison considering the synthetic data set, varying the size

From the Figure 3.11 it is clear that both algorithms scale linearly with the database size, but the R-Tree algorithm has a much better scalability, as expected.

It is also apparent that the R-Tree algorithm is well suited for the benchmark data sets being considered, with exception of the high-dimensional *MiniBoone_PID* data set.

Chapter 4

Co-evolutionary Clustering

Evolutionary algorithms are a *soft computing* technique for finding good solutions for difficult optimization problems (in the $NP-Hard$ sense). Examples of classical evolutionary algorithms include *Genetic Algorithms*, *Particle Swarm Optimization* and *Differential Evolution*. All these algorithms perform a guided random search on the space of possible solutions by iteratively changing and combining individuals of a population. Although useful for solving problems with a single objective function, this approach is not directly applicable to constrained problems. However, it is always possible to adapt common evolutionary algorithms for solving single-objective, constrained problems by using penalty functions. Penalty Functions transform a constrained, single-objective problem into a single-objective problem by worsening the global solution to compensate for a bad solution on another secondary criteria. Expression 4.1 presents the general penalty function approach.

$$f'(\vec{x}) = f(\vec{x}) + p(\vec{x}) \tag{4.1}$$

The penalized function f' equals the original function f plus a penalty function p, defined by the user. The use of such function usually introduces new parameters to the problem, which are in many occasions, difficult to configure. For instance, if one wants to minimize a function f, restricted to a set inequality functions $\vec{g}$, the final fitness function could be f plus a constant if any of the restrictions is violated, or zero if none is violated. This approach tends to fail in complex optimization problems by falling into local minima because it is hard to choose a good value for the penalty constant without some *a priori* knowledge of the problem. The challenge of solving complex, restricted problems motivated the development of other methods for optimizing these kind of functions that do not require the configuration of penalty factors by the user.

In the recent years, some co-evolutionary algorithms were developed for dealing with competing dual-objective problems. Co-evolutionary algorithms differ from traditional evolutionary algorithms by maintaining two or more populations at the same time for optimizing different, but coupled, objective functions. For instance: the design of reliable ship vessels that maximize the ship performance (the more reliable the vessel the more inefficient it tends to be) or to solving complex, non-convex optimization problems with non-linear restrictions.

The mechanics of co-evolutionary algorithms is simple. Both populations try to maximize their profit in respect to the decision of the other population until neither one can improve their solution independently. For instance, suppose the existence of two populations, one trying to maximize the speed of a vehicle in a given terrain by changing some parameters of the project design and other population trying to minimize the speed of the vehicle by changing terrain parameters. The idea is that both populations evolve until neither one can improve their fitness given the best solution in the other population. For instance, given a type of terrain, the speed of the vehicle cannot improve no matter the design parameters, i.e., this is the best possible design for that terrain type. Similarly the terrain population can not decrease the speed of the vehicle by changing terrains parameters, i.e., this is the worst possible terrain for that project.

The configuration that does not allow for any individual in both population to improve their fitness without changing individuals of the other population is called a *Nash Equilibrium*. In this state, no population can improve their quality individually. In the example, the result is a robust vehicle design (works well in all terrains) and the hardest terrain for that design (a terrain that is difficult for all designs). This approach results in good solutions for the worst case possible.

This Chapter presents the contribution of this work: the development of a *co-evolutionary* algorithm for clustering large data sets called *COCLU*. Co-evolution is employed in this work for choosing the most difficult points to cluster in one population and the best algorithms to cluster those points, in another population. The subset of points that are the hardest to cluster contributes the most to the *SSE* of a given clustering solution. Similarly, a good set of algorithms for hard points consists in algorithms capable of clustering well those points, reducing the overall SSE. Experiments show that selection of difficult points to cluster considering a variety of clustering algorithms in the antagonistic population yields a good overall clustering result.

Section 4.1 defines in detail the concept of co-evolutionary algorithms, Section 4.2 presents the pseudo-code of the *COCLU* algorithm with some important consideratons and Section 4.3 presents the classical algorithms used in the population of clustering algorithms.

4.1 Co-evolutionary Algorithms

Co-evolutionary algorithms try to mimic the idea of *predatory* or *symbiotic* behavior that regulates populations in nature. Symbiotic approaches enforce cooperation of populations for achieving a common goal, while predatory behavior introduces competition: if the prey population wishes to survive, it has to adapt itself to its predators. The opposite is also true, i.e., the predator must react to changes in prey configuration if it wishes to maintain its existence. In nature, this arms race constitutes an important evolutionary force that leads to very complex evolution patterns.

Symbiotic Co-evolution

Symbiotic Co-evolution is useful for optimizing two populations towards a common goal. For instance, this approach is applied on the *GENOCOP III* method (GEnetic algorithm for Numeric Optimization of COnstrained Problems) (MICHALEWICZ; NAZHIYATH, 1995). This approach maintains two sets of populations: S_p, called "*Search Points*" and S_r, called "*Reference Points*".

A special Genetic algorithm is used for evolving both populations. The first population has a more exploratory nature, accepting occasional unfeasible solutions, while the second population maintains a pool of strict feasible solutions. The second population evolves more slowly, in a ratio of one iteration per r iterations of population S_p, where r is a parameter of the algorithm.

Because the first population may contain unfeasible individuals, the second population is used to calculate its fitness. Since all individuals of the second population are feasible, their fitness is simply their objective function value. The objective function value of a given individual taken from the first population is calculated as follows:

$$fp(\vec{x}_i) = \begin{cases} f(\vec{x}_i) & \text{If individual } x_i \text{ is feasible,} \\ f(\vec{w}), \vec{w} = a\vec{z} + (1-a)\vec{x}_i & \text{If individual } \vec{x}_i \text{ is infeasible, } \vec{z} \text{ is randomly taken from } S_r. \end{cases} \tag{4.2}$$

The variable a is a random real number in the interval $[0,1]$. This number defines how much of the feasible individual $\vec{z}$ must be used for building the point that will be evaluated. If $a = 1$, the evaluated point will be $\vec{z}$ itself, a feasible solution from the second population, if $a \neq 1$ the individual being evaluated is a hybrid between a feasible individual of the second population and the unfeasible individual of the first population.

Since the second population is always feasible, $f(\vec{w})$ will also be feasible given a suffi-

ciently large value for a. The optimum value of a is the smallest possible taken from an interval $[0,1]$. This value must generate an individual $\vec{w}$ that satisfies all constraints. Since the optimum value of a varies a lot with the objective function and the considered individuals, it is not possible to precisely determine it. Thus, a is randomly selected in the interval between 0 and 1 until a valid value is found. This searching may take many iterations, so it is advised to set a budget condition: if no feasible solution is found after r_{search} iterations the value $a = 1$ is used, yielding a feasible solution.

If the resulting transformed vector $\vec{w}$ has a smaller objective function, it replaces the vector $\vec{z}$ taken from the Reference Points. Also, if this happens, $\vec{w}$ may replace x_i with probability p, p being a parameter of the algorithm.

The expected behaviour is that as iterations passes, both populations will converge to a feasible solution. The best feasible solution is then taken from the populations as the response of the algorithm.

This algorithm is an example of how populations may collaborate for achieving a common goal. The *search population* uses the *reference population* for constructing feasible solutions, while the *reference population* uses individuals generated by the *search population* for composing better feasible solutions. This solution depends on a specialized operator defined in Equation 4.2 for mixing solutions. The operator is not always effective and not directly applicable for solving the problem of clustering large data sets.

A more recent work (LIU et al., 2007) applies another approach: two populations of points, one minimizing the fitness, regardless of the constraints and the other minimizing the constraints violations, regardless of the fitness function. The algorithm works by using the first population to evolve solutions considering only the fitness, ignoring the constraints, after some iterations, the solution tends to be very good in minimizing the objective function, but unfeasible. The second population evolves by considering only the constraints violations, i.e., finding solutions that are as far away as possible from the restrictions, regardless of the fitness value. The "more feasible" an individual, the better it is. After a given number of iterations, migration occurs from one population to another and the process continues until convergence.

The idea is that populations will mix feasible solutions, but poor in the fitness sense, with good fitness solutions, but poor in the feasibility sense. Over time both populations will converge to the same set of points.

There are many other approaches for solving optimization problems with the use of cooperative co-evolutionary techniques. Unfortunately, the majority of them require some a priori

knowledge of the problem that is difficult to assess in the general case. Predatory approaches tend to be more directly applicable for the general optimization problem, as shown in the next Section.

Predatory Co-evolution

This work will rely on the use of a predatory (competitive) approach for solving the clustering problem called "*Co-evolutionary Min-Max*". This approach is simple in its pure form: suppose that two populations, *prey population* and *predator population* compete in the same environment. The adaptability for a given individual is measured by how well it can perform against the *best individual* of the antagonist population. To avoid extinction, both populations must continuously adapt themselves in relation to each another. This arms race of competing populations yields a co-adaptation between prey and predator, i.e., if the predator population evolves in a certain way, the prey population must also evolve if it wants to keep alive. A population will always adapt itself to respond for changes in the opposite population. Under the right circumstances, this adaptive behaviour will go on until a certain equilibrium is reached. At this point, prey and predator converge to their optimal configuration in relation to each other. I.e., it is not possible to improve the solution quality of one individual in population A in respect to the best individual of population B, and vice-versa.

In the last few years, a lot of effort has been put on investigating the use of competitive co-evolutionary methodologies for solving complex problems. The general idea of having two populations evolving towards different goals and interacting somehow is greatly explored in the literature.

There are many ways to encode both populations and how the interaction will take place. In (PAREDIS, 1994) the objective is to minimize a given formulation restricted to a set of constraints. One population encodes the parameters of the fitness functions and the other the constraints. When a solution of the first population is not feasible, its fitness is the objective function plus a penalty constant. When all constraints are met, the fitness is just the value of the objective function. The fitness of the constraints are calculated in function of how many individuals of the opposite population failed to satisfy it.

(BARBOSA, 1996) uses co-evolution with two populations for solving the augmented Lagrangian version of restricted optimization problems. One population encodes the parameters of the fitness function and the other encodes the Lagrangian values of the augmented formulation. A *GA* algorithm is used for solving the augmented Lagrangian formulation of the problem by evolving both the function parameters and the Lagrangian multiplier's in a co-evolutive manner.

Our previous work (FABRIS; KROHLING, 2011) uses the *Differential Evolution* (DE) algorithm, with two populations, coupled using co-evolution techniques for solving different hard optimization problems with restriction. The approach was successful in finding the optima in all tested benchmark functions taken from the literature.

The present work combines classical clustering algorithms via co-evolution, as far as we known, for the first time for solving the problem of clustering large data sets.

4.1.1 Using Co-evolution for clustering large data sets

This approach relies on the fact that not all points are equally important for the clustering process. It is possible to select a subset of the original data set without changing the final clustering result. Figure 4.3 presents this idea: if the clustering algorithm is run in a subset of good, selected points, it is expected that the final clustering result will be the same. However, not every subset is a good one. Also, random selection of points tends to leave important points out of the clustering procedure, resulting in poor results.

From this point forward, "selected points" will denote a subset of points of the original data set D that is being considered for the clustering algorithms. Similarly, "non-selected points" will denote the set of points that is not being considered in the clustering algorithms.

(a) Original data set - All points will be used as input for the clustering algorithm. The "x" marks the calculated centroids

(b) A good point selection - The clustering result in the sampled (selected points) set is the same as the original data set

(c) A bad point selection - The clustering result in the sampled set (selected points) is worse than the original data set

Figure 4.1: Example to demonstrate that some subsampling choices result in a far worse clustering result. Here, the doted grey circles represent points being discarded (non-selected points) and solid black circles points being considered (selected points). Figure (c) displays a clustering result that yields a poor SSE result in the original data set.

It is clear that the selection of hard points to cluster (those farthest from the centroids) generates poor SSE results when the underlying clustering solution is bad. Therefore, to find good overall solutions, it is required to find hard points to cluster, i.e., a set of points that are

the hardest to cluster for every possible centroid distribution. A good solution for those points should yield also a good overall clustering result.

This work develops a way for iteratively building a hard set of selected points to cluster given a current centroid solution. To select these hard points, the first step is to formalize the notion of "easy" selected points in relation to a clustering solution. These easy points need to be exchanged with harder points to improve the representativity of the selected point. The *easiness criterion* considers how much a selected point influences a fixed clustering solution, i.e., how the point contributes to the overall clustering result given a centroid solution. The smaller the contribution (the closer they are from the closest centroid) the more probable must be the swapping of that point to a random, non-selected point.

After some points swapping, the algorithm performs the adaptation of the centroids to the newly selected points by updating the centroid's position. In this step the selected points are fixed and the clustering algorithms try to find a good solution for the new set of selected points.

The observed dynamic here is that centroids will migrate to the region that has the hardest points, making the otherwise hard selected points easier. The centroids tend to chase regions of selected points, while selected points flee from the centroids.

Once the new clustering solution is found, the algorithm loops to the first step and repeats the point selection procedure. This two-step co-evolutionary procedure is repeated until a given number of iterations is passed or convergence is reached.

The population of selected points is called *Population A*. As previously stated, it encodes which points of the original data set will be used for clustering the data, i.e., a small set of representative points that contains sufficient information for minimizing the SSE of the whole population. The goal of the individuals in population A is to maximize the cost of the clustering solutions generated by individuals of population B. The Population A also maintains the best clustering solution for that specific clustering solution, i.e., the clustering solution that minimizes the *SSE* of the selected points.

Individuals in population A suffer two modification operations: *crossover* and *mutation*. *Crossover* works by randomly exchanging selected points of two individuals sampled from the population A and generating one new individual. Mutation works by randomly picking a point $\vec{a}$ of population A, and randomly choosing a selected point p_1 of $\vec{a}$. Next, a random non-selected point (is in the point pool but not in $\vec{a}$) p_2 is also selected. After that, p_1 and p_2 are exchanged. If p_2 has a greater distance to the closest centroid than p_1, the new randomized individual is kept. If the distance between p_2 to its closest centroid is smaller than p_1, the operation discards

this mutation candidate and tries again, for a maximum of n_{tries} iterations.

Population B encodes a set of traditional clustering algorithms that aims to minimize the clustering cost (SSE) given all individuals of population A. While population A evolves towards the set of difficult points to cluster in the SSE sense, population B evolves towards selecting the most efficient algorithms for clustering the points of population A. A small number of algorithms is maintained in the algorithm pool and in each evaluation cycle of population B, one randomly selected inactive algorithm and one randomly selected active algorithm are tested against one randomly selected individual of population A, if the inactive algorithm has a smaller *SSE* result, it becomes active, entering population B while the old active algorithm leaves population B.

For choosing both the best points and clustering algorithms, the fitness function for a given individual must be defined. The fitness of a given individual x of population A, $f_p(\vec{x})$, is calculated via the min-max method: the fitness of $\vec{x}$ is the best clustering result (the minimum SSE) considering all algorithms in population B. The fitness of a individual y in the population B is the worst result (the largest SSE) resulting in running algorithm y in all individuals of population A.

For population A, the bigger the value of the fitness function, the better the individual is. Thus, after generating some new individuals via crossover and mutation, population is trimmed to its original size keeping the SA_{pop} individuals with biggest fitness. Population A evolves towards a sample of difficult points to cluster. If the sample size is big enough to represent the whole population, the result of the clustering in the sample will converge to the result in the whole data set.

Figure 4.2 shows a representation of the procedure where the individual x_k is being evaluated against all individuals of population B in order to calculate its fitness. In other words, the clustering cost of the selected points in x_k is evaluated against all clustering algorithms existing in population B in the *SSE* sense. The fitness of individual x_k is the smallest *SSE* considering all solutions in population B. The bigger the value of the fitness of x_k, the better the individual. In other words, the most difficult set of points to group in the *SSE* sense is the one with greatest fitness.

Figure 4.3 illustrates how selecting points that worsen the current solution found by algorithms in population B improve the overall clustering result.

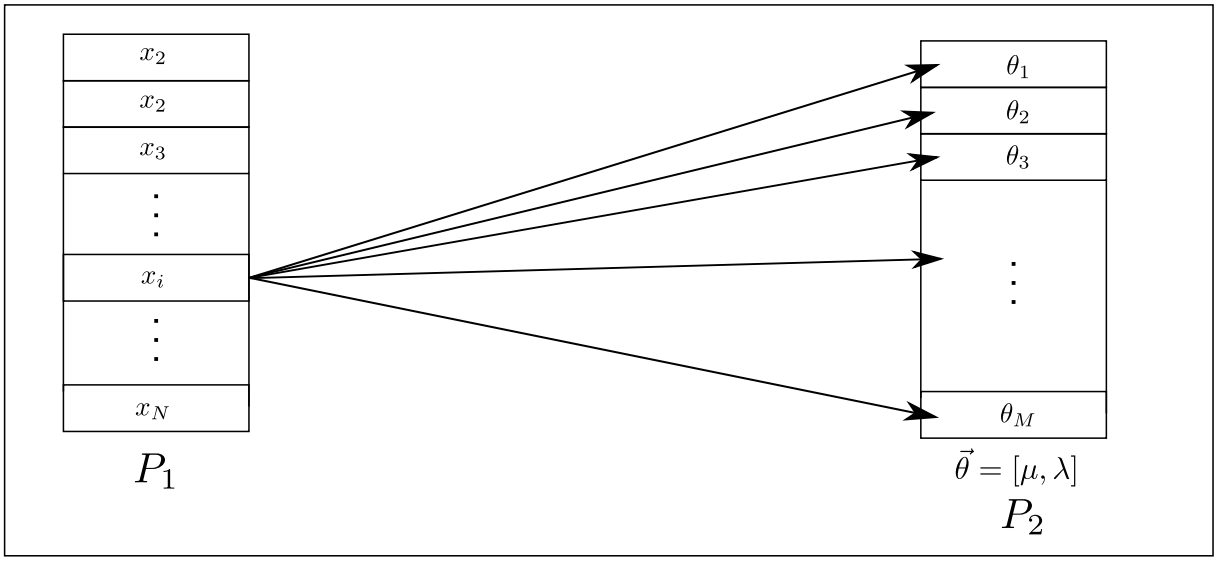
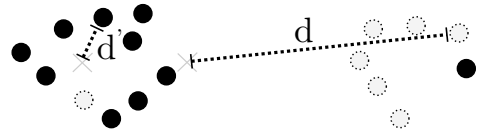


Figure 4.2: Min-max representation

(a) Initial set of points - Bad point choices leading to poor clustering results

(b) Random swapping of points - Exchanging the near point (with distance d' to a distant point in relation to the closest centroid, with distance d).



(c) After the swap, the clustering algorithm is run again and the centroids walk towards a better configuration. The Resulting swapping improved the clustering result slightly - Another swapping selection is displayed, the point with distance d' will be replaced by the point with distance d in the next iteration.

(d) The result of the previous point swapping yields a much better overall clustering. The generated clusters in the whole population would generate the optimum solution in this example.

Figure 4.3: The Figures illustrate how the swapping procedure of points iteratively constructs a better solution on the original data set by choosing data points that worsen the solution found in the selected set.

Next Section describes the details of the *COCLU* algorithm, presenting general considerations, pseudo code and the classical algorithms for small data sets used in the *algorithm pool*.

4.2 The COCLU Algorithm

In this Section the *COCLU* algorithm is presented in detail with important considerations regarding performance and practical issues.

The *COCLU* algorithm uses the basic framework of the general co-evolutionary algorithm for solving constrained optimization problems. Two populations A and B, with a coupled objective function $f(\vec{x},y)$ ($\vec{x}$ coming from population A and y coming from population B), compete towards opposite goals, an individual $\vec{x}$, of population A, tries to maximize the functional $f(\vec{x},B)$, defined as follows:

$$f(\vec{x},B) = min(f(\vec{x},y) \forall y \in B) \tag{4.3}$$

While an individual y, of population B, tries to minimize the functional $f(A,y)$, defined as follows:

$$f(A,y) = max(f(\vec{x},y) \forall x \in A) \tag{4.4}$$

Both populations will go through the process of finding new solutions until they converge, i.e., no improvement is possible in either one of them. This point is called the *saddle point* of the min-max problem. The generic classical co-evolutionary algorithm for finding the saddle point of the min-max problem is described in Algorithm 9.

Algorithm 9 The generic co-evolutionary algorithm for solving a min-max problem

```
    procedure CoEvo(Number of Cycles (MaxCycles) , Number of Iterations on Population
    A (MaxGenA), Number of Iterations on Population B (MaxGenB))
        Initialize Population A
        Initialize Population B
        for each k = 1 to MaxCycles do
5:          for each j = 1 to MaxGenA do
                Evaluate new Population A
                Generate new Population A
            end for
            for each j = 1 to MaxGenB do
10:             Evaluate new Population B
                Generate new Population B
            end for
        end for
        Return the best individual of population A given some quality metric.
15: end procedure
```

The first steps of this algorithm involve the initialization of individuals in populations A and B (line 2 and 3). The algorithm begins in the *for* loop in line 4, that controls how many times the evaluation of populations A and B will be performed. The algorithm will repeat the adjustment of populations A and B $MaxCycle$ times.

After that, population A evolves for $MaxGenA$ iterations (the loop in line 5). The evolution of population A includes evaluation of newly generated individuals (line 6) and generation of new individuals (line 7).

The *for* loop in line 9 is analogous to the previously explained with the exception that it operates on population B, instead of A. The concepts however, are identical. The encoding of the individuals in the population and the details of procedures of initialization, generation and evolution of populations A and B will be described in the following Subsections.

4.2.1 Encoding of the Individuals

The first step for applying some meta-heuristic to the problem of clustering large data sets is to encode the problem properly so that the algorithms may perform the required operations on individuals of the population.

Each individual in population A corresponds to a s-sized subset of distinct point indexes of the original data set. The encoding of these individuals is a vector of size s containing in each position a unique data point index of the original data set. This representation is convenient for both performance and operation definition reasons. The parameter s regulates the summa-

rization ratio of the co-evolutionary algorithm. If s is too big, there is no performance gain in executing the co-evolutionary algorithm.

The definition of population A is formalized below.

$$\vec{x_i} \in A, i > 0, i \leq A_{pop} \tag{4.5}$$

$$\vec{x_i} \in \mathbb{N}^s, s \leq n \tag{4.6}$$

$$\vec{x_{i,j}} \neq \vec{x_{i,k}} \forall i, j, k, (j \neq k) \tag{4.7}$$

Individuals in population B, on the other hand, are simply a fixed-sized set of clustering algorithms extracted from a pre-defined algorithm pool.

4.2.2 Initialization of the Populations

Initialization is an initial sampling of the search space, before the first evolution iteration begins. In this work it only happens once for each population. It is important that the randomized initialization procedure is broad enough so that the search space is not restricted to a small subspace of all possible solutions.

Population A

Initialization of population A is a simple random uniform sub-sample of size s of the data set D, performed multiple times. The only caveat is to remove repetitions from the generated sub-sample. The following pseudo-code presents the initialization procedure:

Algorithm 10 Population Initialization

```
    procedure POPULATION INITIALIZATION(Size of the population (SA_pop), Number of sample points (s), The original data set (D))
        A = []
        for each i in [1..SA_pop] do
            x_tmp = []
 5:         for each j in [1..s] do
                a = rand(1, |D|)
                while a ∈ x_tmp do
                    a = rand(1, |D|)
                end while
10:             x_tmp,j = a
            end for
            A_i = x_tmp
        end for
        return A
15: end procedure
```

This algorithm merely builds the initial population A by creating for each individual of the population a random sample of unique points. This algorithm is not suitable for generating random initial solution when $|D|$ and s are large. The while loop (line 7) tends to take a considerable amount of time for randomly generating all distinct values for A_i in these cases.

Population B

The population B consists of the clustering algorithms. The initialization of the population is simple: one needs only to sample s_b algorithms from the algorithm pool to initialize the B population and randomly select an individual of population A for solving the reduced clustering problem using a traditional algorithm.

4.2.3 Fitness Evaluation

The algorithm for evaluating the fitness of the individuals in both populations is similar; one has to select an individual in one population and iterate over the other population looking for the minimum (for an individual of population A) or the maximum (for an individual of population B).

Population A

In the Algorithm 11, the algorithm for calculating the fitness of a individual $\vec{x}$ is presented.

Algorithm 11 Evaluation of the fitness of an individual of population A

procedure FITNESS EVALUATION OF POPULATION A($\vec{x}$: Individual of A, B: Population B)
  $Fit_{best} = \infty$
  for each b in B **do**
    $Fit_{tmp} = f(\vec{x}, b)$, i.e., evaluate $\vec{x}$ using algorithm b.
    if $Fit_{tmp} < Fit_{best}$ **then**
      $Fit_{best} = Fit_{tmp}$
    end if
  end for
  Return Fit_{best}
end procedure

The process for calculating the fitness of an individual in population B is similar and is described in the next listing.

Population B

Similarly, the algorithm to calculate the fitness of an individual b in population B is described in Algorithm 12.

Algorithm 12 Evaluation of the fitness of an individual in population B

procedure FITNESS EVALUATION IN POPULATION B(b: Individual of B, A: Population A)
  $Fit_{best} = 0$
  for all a in A **do**
    $Fit_{tmp} = f(a, b)$, i.e., evaluate algorithm b using individual a.
    if $Fit_{tmp} > Fit_{best}$ **then**
      $Fit_{best} = Fit_{tmp}$
    end if
  end for
  Return Fit_{best}
end procedure

The two previously defined algorithms calculate the fitness of one individual of population A or B by iterating over all individuals of the opposite population. It is possible to sacrifice accuracy to gain performance by sub-sampling the opposite population instead of iterating over every point of it. This approach, however, was not tested in this work.

With the evaluation for every point, the selection procedure is simple. For population B one

needs only to remove the worst individuals of the population with regard to the fitness function, i.e., the clustering algorithms with the highest *SSE*. Similarly, for population *A* one needs only to remove the worst individuals of the population with regard to the fitness function, i.e., the selected points with the lowest *SSE*.

4.2.4 Generation of Population

Once the fitness calculation of the individuals of the population *A* and *B* is defined, one can use use special operations to create new individuals, considering the fitness as a selection parameter and evolving both populations over time. The generation of the new individuals of population *A* follows the logic of the algorithm presented in Algorithm 13.

Algorithm 13 Generation of new individuals of population *A*

procedure GENERATING NEW INDIVIDUALS OF POPULATION *A*(*A*: Population *A*)
    Execute mutation on population *A*
    Execute crossover on population *A*
end procedure

The generation of individuals in population *B* is simpler than of population *A*, and follows the algorithm presented in Algorithm 14.

Algorithm 14 Generation of new individuals of population *B*

procedure GENERATING NEW INDIVIDUALS OF POPULATION *B*(*B*: Population *B*, SB_{pop}: The number of individuals of population *B*)
    Selects new clustering algorithm from the algorithm pool for executing the co-evolution.
end procedure

Next, we define the operations used for both populations.

Mutation of individuals in population *A*

Mutation of individuals in population *A* is similar to the mutation operation of the *Genetic Algorithm*, a number of N_{mut} individuals of the population *A* (of selected points) is picked at random and a selected point of the individual is randomly picked as a pivot. After that, another point is picked from the non-selected set of points. If the replacing procedure yields a smaller *SSE* than the solution that generated it, the mutation procedure is repeated. The algorithm tries to find a mutated individual that minimizes the *SSE* until a given budget condition is reached.

It is important to notice that there may not be duplicated values of a given point after the mutation operation. The new individual is appended to the end of the current population. The algorithm in Algorithm 15 explains the mutation procedure.

Algorithm 15 Mutation of an individual x of population A

procedure MUTATION PROCEDURE($\vec{A}$: Population A)
  for each i in $[1..N_{mut}]$ **do**
    for each j in $[1..n_{tries}]$ **do**
      Randomly pick a selected point of A, a_{pivot}, to be exchanged
      Randomly pick a non-selected point of A, a_{non}, to exchange a_{pivot}
      Take S, as the cluster solution (set of centroids) that defines the fitness of individual a
      Find S_{pivot} and S_{non}, the closest centroids from a_{pivot} and a_{non}, respectively
      if The distance between S_{pivot} and a_{pivot} is smaller than the distance S_{non} and a_{non} **then**
        Take the new individual formed by substituting a_{pivot} with a_{non}, replacing the old individual by the new individual.
      end if
    end for
  end for
end procedure

The mutation algorithm tries to find hard points to cluster by randomly exchanging selected points with non-selected points.

Crossover of individuals in population A

Crossover is implemented by randomly choosing two individuals using roulette wheel selection and building one new individual by randomly picking the attributes of both individuals, always verifying if the resulting individual is valid, i.e., has no selected point appearing twice.

The algorithm in Algorithm 16 explains this procedure.

Algorithm 16 Crossover individuals of population A

procedure CROSSOVER PROCEDURE($\vec{A}$: Population A, individual size s)
  for each i in $[1..n_{ncross}]$ **do**
    Randomly pick two different individuals, $\vec{a_1}$ and $\vec{a_2}$, of population A, proportional to their fitness, i.e., the greater the SSE value, the more probable the selection (roulette wheel selection)
    $a_{cross} = \{\}$
    while $|a_{cross}| < s$ **do**
      $r = rand(0,1)$
      if $r > 0.5$ **then**
        Randomly pick a selected point p from individual $\vec{a_1}$
      else
        Randomly pick a selected point p from individual $\vec{a_2}$
      end if
      if $p \notin a_{cross}$ **then**
        $a_{cross} = a_{cross} \cup \{p\}$
      end if
    end while
    Return a_{cross}
  end for
end procedure

Selection of individuals in population A

After mutation and crossover, the selection of individuals is performed. This selection is merely the elimination of the most unfit individuals so that the population returns to its original size.

To determine the fitness of the selected points, the algorithm 11 must be executed for the newly generated individuals. The individuals with smallest SSE must be replaced.

Replacement of algorithms in population B

In every iteration of the co-evolutionary algorithm on population B, the N_{unfit} most unfit clustering algorithms are replaced by N_{unfit} clustering algorithms of the algorithm pool, selected at random. This approach has two important characteristics: it automatically chooses the more efficient algorithms to the data and avoids local minima by always trying new solutions for the existent points. Once a new algorithm is selected, it randomly chooses one individual of population A, proportional to its fitness for finding a solution for the reduced clustering problem. Once a solution for the reduced clustering problem is found, it replaces the old clustering algorithm if its SSE is smaller than one of the current clustering algorithm present in the population.

4.3 Classical clustering algorithms used for population *B*

As previously stated, population *B* consists in a pool of classical clustering algorithm used to solve a sub problem of the original, large, clustering task. In the following Section, the algorithms used for generating the pool are presented. In theory, any clustering algorithm could be used as the co-clustering engine, this set was selected for being widely spread in the literature, having attested performance in small data sets and fast response times.

4.3.1 K-Means Clustering Algorithm

The *k*-means clustering algorithm or just "*k*-means", was introduced in the seminal work of (MACQUEEN, 1967; STEINHAUS, 1956) and has persisted until today as one of the most used clustering algorithms.

The canonical *k*-means algorithm first selects points of the data set randomly to create *k* centroids and iteratively moves the initial set of centroids to a better cluster selection. Algorithm 17 presents the algorithm.

Algorithm 17 The *k*-means algorithm

procedure *k*-MEANS(Data set *D*, The number of Clusters *k*, The maximum number of iterations it_{max})
    Randomly select *k* points for acting as the initial centroids
    Assign each point to its closest centroid
    $i = 0$
    while a point of the data set changed clusters and $i < it_{max}$ **do**
        Recalculate the centroids as the mean point of all points of the cluster
        Recalculate the closest centroid of every point
    end while
end procedure

Theoretically the *k*-means algorithm is known to converge in a finite number of iterations, but in some cases it may take many iterations and in practice may not even converge at all, due to rounding errors. Thus, it is important to fix a maximum number of iterations it_{max} for the algorithm. In this work we set the maximum number of iterations for the *k*-means method to 100 ($it_{max} = 100$). In the tests performed, the algorithm took less than it_{max} iterations to converge in every occasion.

4.3.2 Spectral Algorithm

The Spectral Clustering Algorithm (NG; JORDAN; WEISS, 2001) is a widely spread clustering algorithm for small-sized data sets. It works by taking a similarity distance matrix $L_{n\times n}$ of all n points and calculating the eigenvectors and eigenvalues of M_d, stacking them, creating a new matrix and performing the clustering in this new data set. The algorithm builds matrix A using the following steps.

1. $A_{i,j} = exp(-||s_i - s_j||^2/2\sigma^2)$ if $i \neq j$, and $A_{ii} = 0$.
2. Define D as the diagonal matrix whose (i,i) element is the sum of A's i-th row and construct the matrix $L = D^{-1/2}AD^{-1/2}$.
3. Find the k largest eigenvalues of L and the respective eigenvectors $x_1, x_2, ...x_k$, forming a matrix $X_{n\times k}$ by stacking the k eigenvalues in columns.
4. Form a matrix Y of X by normalizing each row of X to have unit length.
5. Run a classical clustering algorithm in Y (e.g. k-means).
6. Assign each point of the resulting clustering to an original point of data set D.

According to (NG; JORDAN; WEISS, 2001), this algorithms works quite well in small data sets with great dissimilarity between points. Also, there exists a quality guarantee for the resulting clusters.

4.3.3 CURE and CLARANS Algorithm

These algorithms were used for both constructing the clustering algorithm pool B and as a stand-alone benchmark algorithm for large data sets. Refer to the previous Chapter for more detail.

Chapter 5

Clustering Results

In this Chapter, the results of the experimental runs for all data sets are presented and discussed. These data sets vary in size and complexity, from small data sets for validating the results to large data sets for testing the actual performance of the algorithms in difficult scenarios. Both solution quality and running time were measured, the solution quality was compared using statistical tests.

Before presenting the results, the benchmark data sets are presented, with the original source and some considerations. Also, the parameters used by the algorithms are presented.

5.1 Benchmark data sets

To test the clustering algorithms, nine benchmark data sets were used, among those, eight were extracted from the literature and one exclusive data set was built from register information of consumers of the local energy distributer company.

Table 5.1 presents the details of the chosen benchmark data sets used in this work.

When available, the number of classes of the data set is used. It is worth noticing that the classes of the data may not represent a clear spatial separation of the data. I.e., there may be samples in the same class but in different regions of the attribute space. In this case, the number of classes would not reflect the correct number of clusters. Upon examination, the used data sets with available classes does have a spatial distribution that would be properly organized in clusters. The *X2D2K* and *X8D5K* data sets were created for the very reason of verifying the performance of methods for clustering spatial data sets. Upon inspection, the pen digits data set showed to have a spatial division of the existent classes.

Base name	Size	Dimensions	Clusters (k)	Original source
X2D2K	1000	2	2	(STREHL; GHOSH, 2003)
X8D5K	1000	8	8	(STREHL; GHOSH, 2003)
BRD14051	14051	2	N/A	(REINELT, 1991)
Pen Digits	7494	16	10	(FRANK; ASUNCION, 2010)
PLA33810	33810	2	N/A	(REINELT, 1991)
Shuttle	43500	9	N/A	(FRANK; ASUNCION, 2010)
PLA85900	85900	2	N/A	(REINELT, 1991)
ESCELSA	324515	2	N/A	N/A
MiniBooNE_PID	65032	50	N/A	(FRANK; ASUNCION, 2010)

Table 5.1: When the literature does not provide a value for the number of clusters, the range of values between 2 and 29 is used for testing. Only the trainning sets of the data sets were used in the algorithm's evaluation

Each algorithm was run 10 times for each test to measure the median and variance of their SSE value (when there is some stochastic behavior in the algorithm) and run-times. Some tests could not be run due to the impossibility of setting up important algorithm parameters or because of the excessive running times.

To verify if there is a statistical difference between the COCLU method and the other algorithms when the value of k varies, the *non-paired Wilcoxon signed-ranks test* was used for each value of k. This test is indicated for comparing the performance of two algorithm in the same data set (DEMSAR, 2006) with independent runs. This is a non-parametric test, no assumption is made on the underlying distribution of the data, serving as an alternative to the paired t-test. After the execution of the *Wilcoxon test*, the p-values for all values of k were corrected using the *Holm* method, for allowing the comparison between runs of the same experiment.

When the value of k is fixed, the *Kruskall-Wallis* test is used for detecting if the algorithms are different. If any difference is detected, the *Wilcoxon* test is used to find which algorithm(s) differs from the *COCLU* algorithm, finally, the p-values are corrected using the *Holm* method, for allowing direct comparison between the COCLU algorithm and the other methods. If this comparison were not applied it would be not possible to ascertain that the COCLU algorithm is better than other methods.

5.2 Setup of algorithms

The next Tables present the default set of parameter values used for all data sets, unless stated otherwise in their particular Sections. These values are the recommended set up found in the literature.

5.2.1 CLARANS

The parameter *maxNeighbors* regulates how many times the CLARANS algorithm will try to generate random solutions. It is the most sensitive parameter with respect to processing time and solution quality. The seminal paper of CLARANS suggests using the following expression for configuring this parameter:

$$maxNeighbor = k * (|D| - k) * 0.0125 \tag{5.1}$$

This expression regulates the parameter so that the more difficult the problem (the bigger the values of k and $|D|$) the more tries the algorithm has for solving the problem.

The parameter *numLocals* controls how many worst solutions the algorithm will tolerate finding before restarting the searching procedure. The literature suggests using $numLocals = 2$.

5.2.2 BIRCH

The BIRCH algorithm's most important parameter is also the most sensitive and hard to estimate, since it has direct relation to the nature of the data set. Several experiments were performed to find the set of parameters described in Table /5.2. To find these set of values, first the smallest pairwise distance between points is found. The algorithm is run using this distance multiplied by a factor that varied linearly until the best clustering result was found.

Data set	Closeness and compactness parameter
X2D2K	Smallest Distance
X8D5K	Smallest Distance
BRD14051	9000
Pen Digits	Smallest Distance
PLA33810	$4 * 10^7$
Shuttle	86
PLA85900	700000
ESCELSA	$1 * 10^7$
MiniBooNE	$9 * 10^{28}$

Table 5.2: Parameters of the BIRCH algorithm for all data sets

5.2.3 CURE

The CURE algorithm has only one relevant parameter, the size of the representative set. According to the literature, a good value for this parameter is 10. According to preliminary

tests, this value resulted in a good compromise between time and performance in all data sets.

5.2.4 DBSCAN

The *similarity factor* controls indirectly how many clusters the algorithm will form. This value had to be indirectly set up with the local search algorithm 8.

The value for $MinPts$ was set up based on suggestions taken from the literature and experimentation on the available data sets. A good value for this parameter was found to be 2.

5.2.5 COCLU

All test runs used the following set of parameters. These parameters were estimated in preliminary runs of the algorithm on the previously presented data sets. This set was considered good for all available algorithm runs.

Parameter	Value	Description
$MaxCycles$	50	Number of outer iterations of the min-max algorithm (Algorithm 9)
$MaxGensA$	10	Number of iterations over the population A while population B is frozen. (Algorithm 9)
$MaxGenB$	1	Number of iterations over the population B while population A is frozen. (Algorithm 9)
SA_{pop}	4	Size of the population A of selected points (Algorithm 10)
$ncross$	1	Number of crossovers on each iteration of the inner loop (Algorithm 16)
$nmut$	1	Number of mutations on each iteration of the inner loop (Algorithm 15)
s	$0.05 * \|D\|$	The number of points that need to be considered by a given individual of the population (Algorithm 10)
SB_{pop}	1	Size of the population B of clustering algorithms Algorithm 14)

Table 5.3: Parameters of the COCLU algorithm for all data sets

5.2.6 Testing Environment

All tests were run on a *AMD Athlon(tm) 64 X2 Dual Core Processor 4000+* with *1 Gb* of available physical memory. The algorithms were implemented using *ANSI* $C++$ in a UNIX environment.

5.3 Results

In the following Subsections, the results of algorithms for all data sets are presented with a statistical test for verifying the results.

5.3.1 Results for X2D2K Base

Next, the results for the data set X2D2K for all five algorithms with $k = 2$ are presented. The first table presents the SSE of the tested algorithms while the second table presents their running times.

Table 5.4: Sum of squared errors (SSE) of all algorithms for base X2D2K, $k = 2$

Method	Mean SSE	Best SSE	Worst SSE	Median SSE	SSE Std. Dev.
CLARANS	**122.78**	122.60	**123.19**	**122.72**	0.19
DBSCAN	210.69	210.69	210.69	210.69	N/A
BIRCH	124.02	124.02	124.02	124.02	**0.00**
CURE	226.03	226.03	226.03	226.03	N/A
COCLU	122.90	**122.59**	123.58	122.78	0.34

The Table 5.4 shows the results of all algorithms in a relatively small data set for validating their behaviour. In this benchmark all algorithms, with the exception of the CURE and DBSCAN algorithms, managed to successfully find a good set of centroids for the clustering problem. The CLARANS algorithms achieved the best results in this data set with the proposed COCLU algorithm achieving similar SSE results.

In order to perform the statistical analysis, the *Kruskall* test is executed to detect differences in the algorithms. This test reported a p-value of $1.837 * 10^{-07}$, clearly pointing to a significant difference between algorithms.

Next, we need to verify if the CLARANS algorithm is in fact better than the COCLU algorithm in this data set, the adjusted *Wilcoxon* signed-rank test was performed on all ten runs

of the algorithm and the *Holm* adjustment was executed. The *null hypothesis* H^0 is that the means of the algorithm's SSE are the same and the *alternative hypothesis* H^1 is that they are different. The test resulted in a p-value of 0.4492 (confidence interval of 95%), therefore, not rejecting the *null hypothesis*. A p-value of 0.05 would be enough to reject the null hypothesis.

When repeating the statistical test with the BIRCH algorithm, the found p-value was 0.0002 (confidence interval of 95%), being possible, therefore to reject the *null hypothesis* and state that the COCLU algorithm is better than the BIRCH algorithm for this data set. The same is also true for the other two algorithms.

Table 5.5: Running times of all algorithms on base X2D2K

Method	Mean Time (ms)	Best Time (ms)	Worst Time (ms)	Median Time (ms)	Time Std. Dev. (ms)
CLARANS	**28.0**	**20.0**	**40.0**	**20.0**	9.80
DBSCAN	144.0	140.0	150.0	140.0	4.90
BIRCH	60.0	60.0	60.0	60.0	**0.00**
CURE	633.3	620.0	650.0	630.0	8.16
COCLU	537.8	390.0	720.0	490.0	117.36

Table 5.5 shows that the CLARANS algorithm also has the best runtime result. It manages to be the best algorithm in both result quality and running time. It is also worth noticing how the running time of the *COCLU* algorithm is worse than the running time of the CLARANS algorithm in this data set. It is clear that in this experiment, the CLARANS algorithm outperforms every other algorithm tested.

5.3.2 Results for X8D5K Base

Next, the results for the data set X8D5K for all five algorithms with $k = 8$ are presented. The table 5.6 presents the SSE of the tested algorithms while the table 5.7 presents their running times.

Table 5.6: Sum of squared errors (SSE) of all algorithms for base X8D5K, $k = 8$

Method	Mean SSE	Best SSE	Worst SSE	Median SSE	SSE Std. Dev.
CLARANS	**266.78**	**265.24**	270.79	**266.59**	1.44
DBSCAN	346.02	346.02	346.02	346.02	N/A
BIRCH	291.86	266.30	370.68	270.10	42.00
CURE	276.37	276.37	276.37	276.37	N/A
COCLU	267.46	266.49	**268.49**	267.61	**0.58**

The table 5.6 shows the results of all algorithms in another relatively small data set for the purpose of validating their behaviour. It was observed that, once again, all algorithms with the exception of the DBSCAN algorithm managed to successfully find a good set of centroids for the clustering problem. The CLARANS algorithms achieved, again, the best results in this data set with the proposed COCLU algorithm in the second place, achieving similar SSE results.

In order to perform the statistical analysis, the *Kruskall* test is executed to detect differences in the algorithms. This test reported a p-value of $1.197 * 10^{-05}$, clearly pointing to a significant difference between algorithms.

Next, we need to verify if the CLARANS algorithm is in fact better than the COCLU algorithm in this data set, the adjusted *Wilcoxon* signed-rank test was performed on all ten runs of the algorithm and the *Holm* adjustment was executed. The *null hypothesis* H^0 is that the means of the algorithm's SSE are the same and the *alternative hypothesis* H^1 is that they are different. The test resulted in a p-value of 0.019 (confidence interval of 95%), therefore, rejecting the *null hypothesis*, i.e., the CLARANS algorithm is better than the COCLU algorithm in this data set.

When repeating the statistical test with the CURE algorithm, the found p-value was 0.00012 (confidence interval of 95%), being possible, therefore, to reject the *null hypothesis* and state that the COCLU algorithm is better than the CURE algorithm for this data set. The same is also true for the other two algorithms.

Table 5.7: Running times of all algorithms on base X8D5K

Method	Mean Time (ms)	Best Time (ms)	Worst Time (ms)	Median Time (ms)	Time Std. Dev. (ms)
CLARANS	153.00	140.0	200.0	150.0	16.16
DBSCAN	306.00	300.0	320.0	305.0	6.63
BIRCH	**80.00**	**80.0**	**80.0**	**80.0**	**0.00**
CURE	1157.78	1130.0	1230.0	1150.0	28.59
COCLU	2207.78	1760.0	2830.0	2160.0	380.49

Is this experiment, it is clear from Table 5.7 that the BIRCH algorithm got the best running time, but only the third SSE result. In this experiment, the COCLU algorithm got the worst running time. Since both X2D2K and X8D5K data sets contain the same amount of points, only differing in the dimension and the number of clusters k, it appears that the COCLU algorithm is more sensitive to the dimensionality scaling of the data than the other algorithms.

5.3.3 Results for BRD14051 Base

This data set does not contain the number of clusters that must be formed, thus, the set of values in the interval between 2 and 29 was tested to assess both result quality and running time. The results are presented in the following Figures.

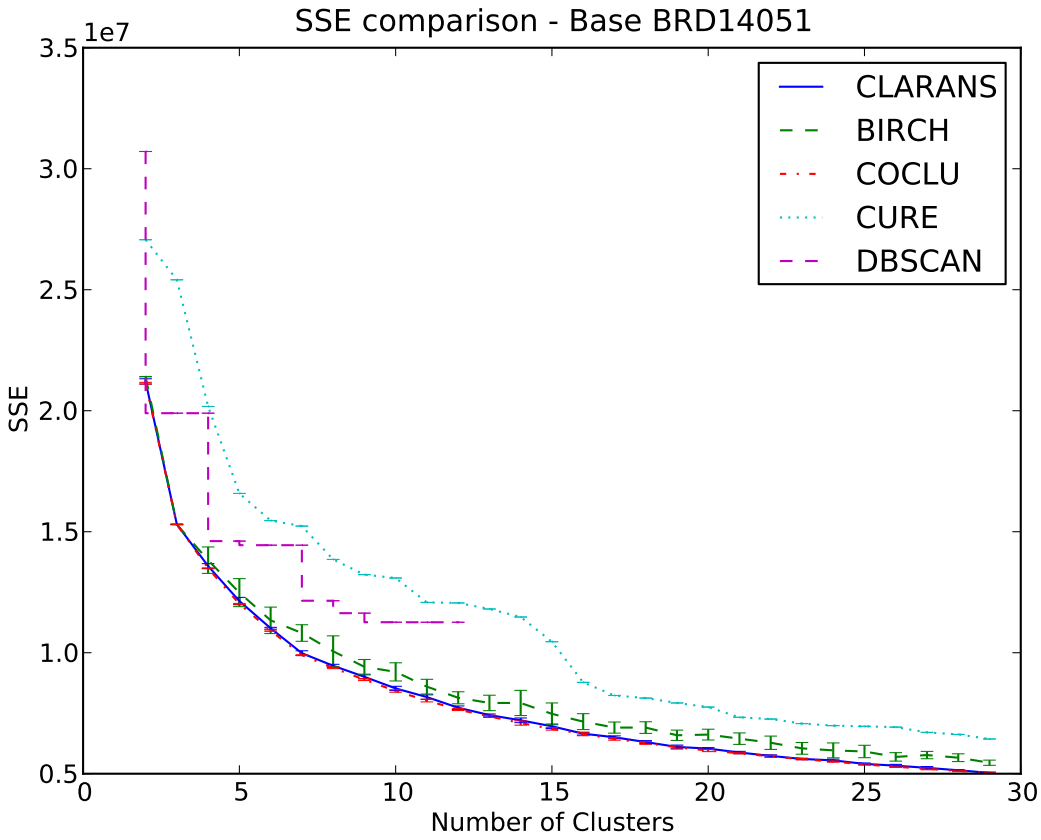


Figure 5.1: The SSE of all tested algorithms

From Figure 5.1 it is clear that both COCLU and CLARANS algorithms achieved the best SSE values for this data set, with very similar results. The BIRCH algorithm was the third best algorithm, achieving very close results. The DBSCAN algorithm could not run in great part of this data set: the algorithm failed to find the proper value of k when it was larger than 11. The reason for this is that the local search algorithm could not find good values for the *similarity factor* parameter that resulted on the desired number of clusters.

To examine the statistical difference between CLARANS and COCLU algorithms, the adjusted *Wilcoxon* test was performed independently for every k. The test could not find a significant statistical difference between the two algorithms, excluding $k = 6$, which had a $p - value$ of 0.04, pointing to a better performance of the CLARANS algorithm (confidence interval of 95%).

However, when testing the COCLU algorithm against the third best algorithm, the BIRCH algorithm, significant statistical difference was found for every k, excluding $k = 3$, (p-value $<$ 0.05 and confidence interval of 95%). Also according to the *Wilcoxon test*, it is safe to say that the COCLU algorithm is better than the other two clustering algorithms in this data set.

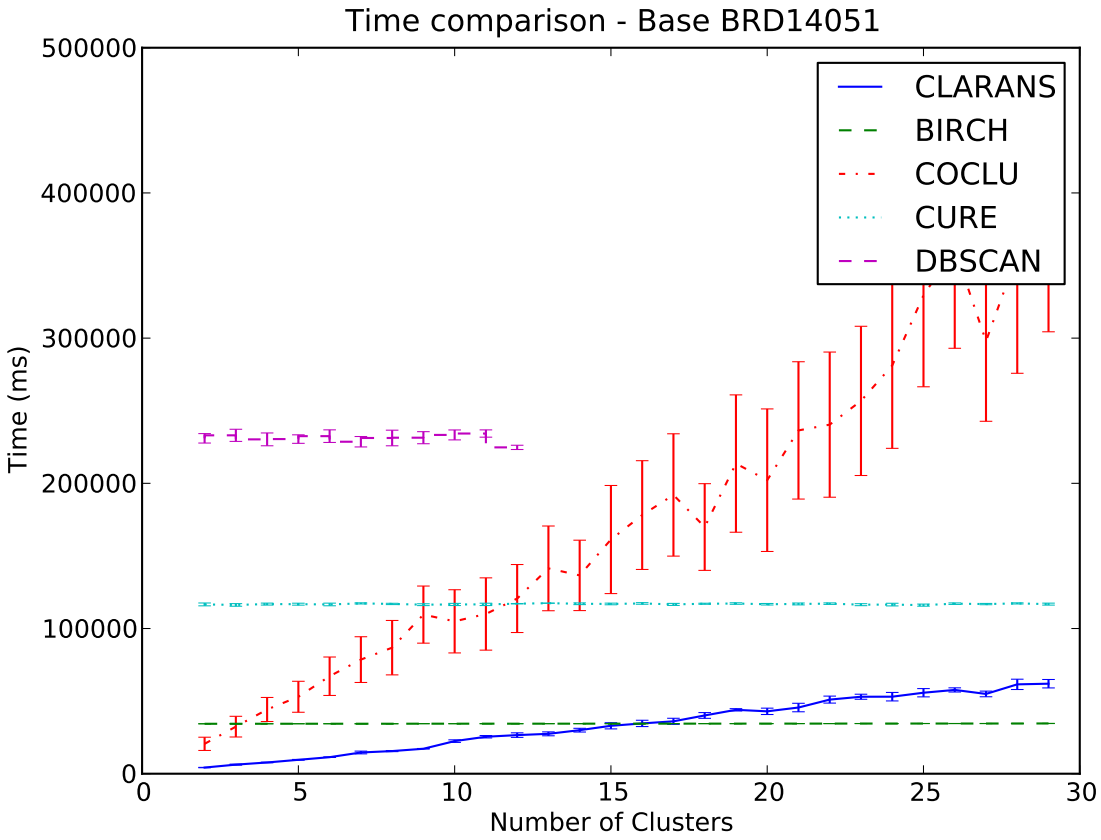


Figure 5.2: Running times of all algorithms on base BRD14051

Figure 5.2 shows two different asymptotic behaviours. The first behaviour comes from algorithms that are very sensitive to the parameter k (COCLU, and CLARANS). The three others are not so much influenced by the increase of the number of clusters. Also, it is clear that even though the COCLU algorithm presents a linear increase in run-time, it is a very steep one compared to the other algorithms. Also, the time variance of the COCLU algorithm is much greater when compared to the other algorithms.

5.3.4 Results for Pen Digits Base

Next, the results for the data set Pen Digits for all five algorithms with $k = 10$ are presented. The table 5.8 presents the SSE of the tested algorithms while table 5.9 presents their running times.

Table 5.8: Sum of squared errors (SSE) of all algorithms for base Pen Digits, $k = 10$

Method	Mean SSE	Best SSE	Worst SSE	Median SSE	SSE Std. Dev.
CLARANS	487542.2	480133.0	497726.0	488637.0	5106.2
DBSCAN	859522.0	859522.0	859522.0	859522.0	N/A
BIRCH	**482763.0**	**475349.0**	491656.0	**480653.0**	5557.0
CURE	821756.1	821756.1	821756.1	821756.1	N/A
COCLU	484142.3	479838.0	**487940.0**	484386.0	**2465.8**

The Table 5.8 shows the SSE results of the tested algorithms for the Pen Digits data set. The best algorithm for this data set was the BIRCH algorithm, with the COCLU algorithm and the CLARANS algorithm at the second and third places. In this test, however, the worst result of the BIRCH algorithm was very poor, yielding a high variance. The COCLU algorithm, for instance, had a much better worst case.

In order to perform the statistical analysis, the *Kruskall* test is executed to detect differences in the algorithms. This test reported a p-value of $3.357 * 10^{-05}$, clearly pointing to a significant difference between algorithms.

According to the adjusted *Wilcoxon* test it is possible to state that the BIRCH algorithm is statistically better than the COCLU algorithm with a p-value of 0.00002 (confidence interval of 95%) for the H^0 hypothesis that both algorithms are the same. It is also possible to state that the COCLU algorithm is statistically better than CLARANS with the same p-value and confidence interval. It also follows that the COCLU algorithm is better than other two algorithms, CURE and DBSCAN.

Table 5.9: Running times of all algorithms on base Pen Digits

Method	Mean Time (ms)	Best Time (ms)	Worst Time (ms)	Median Time (ms)	Time Std. Dev. (ms)
CLARANS	**16716.0**	**15770.0**	**17130.0**	**16925.0**	437.86
DBSCAN	62981.0	60570.0	65100.0	63115.0	1526.42
BIRCH	530058.9	529870.0	530220.0	530100.0	**115.18**
CURE	95275.6	94820.0	96030.0	95020.0	423.82
COCLU	63247.8	48670.0	70560.0	65260.0	6835.97

Table 5.9 shows the running times of the tested algorithms from this table. It is clear that the CLARANS algorithm outperformed every other tested algorithm on that regard. Also, the

best algorithm in the SSE sense (BIRCH) had the second best time. This is evidence that this algorithm works well in this data set.

5.3.5 Results for PLA33810 Base

This data set also does not contain the number of clusters that must be formed, thus, the set of values in the interval between 2 and 29 was tested to assess both result quality and running time. The results are presented in the following Figures.

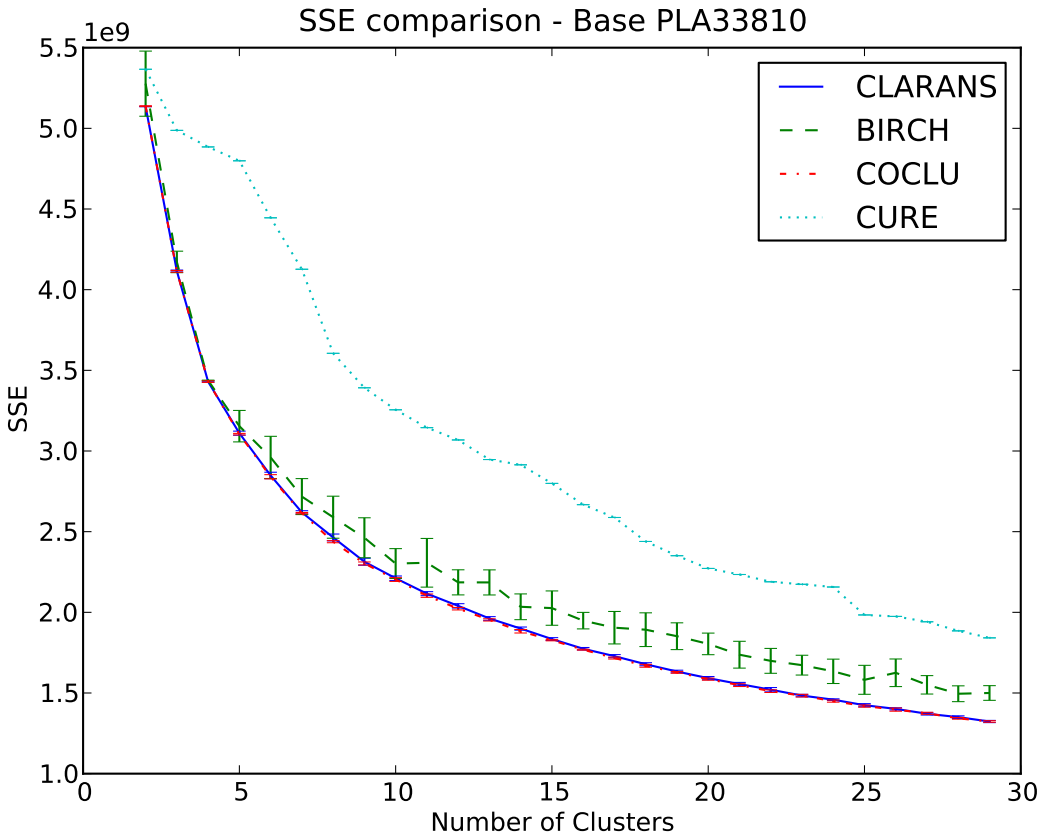


Figure 5.3: The SSE of all tested algorithms, with the exception of the DBSCAN algorithm on data set PLA33810

This experiment had similar results to those obtained on the test of BRD14051 data set. From Figure 5.3 it is clear that both COCLU and CLARANS algorithms achieved the best SSE values, with very similar results. The BIRCH algorithm was the third best algorithm, achieving close results from the two best algorithms. Once again, the CURE algorithm had the fourth best results, with an erratic behaviour, probably due to poor summarization of the data. The DBSCAN algorithm could not run at all on this data set. The algorithm failed to find the proper value for the *similarity factor*, generating bad values of k in every occasion.

According to the adjusted *Wilcoxon* test, it is not possible to ascertain that the COCLU algorithm is better than the CLARANS algorithm for any values of k that were tested (p-value < 0.05 and confidence level of 95%).

However, It was possible to ascertain (p-value < 0.05 and confidence level of 95%) that both the CLARANS and COCLU algorithms are better than the other two algorithms for every value of k.

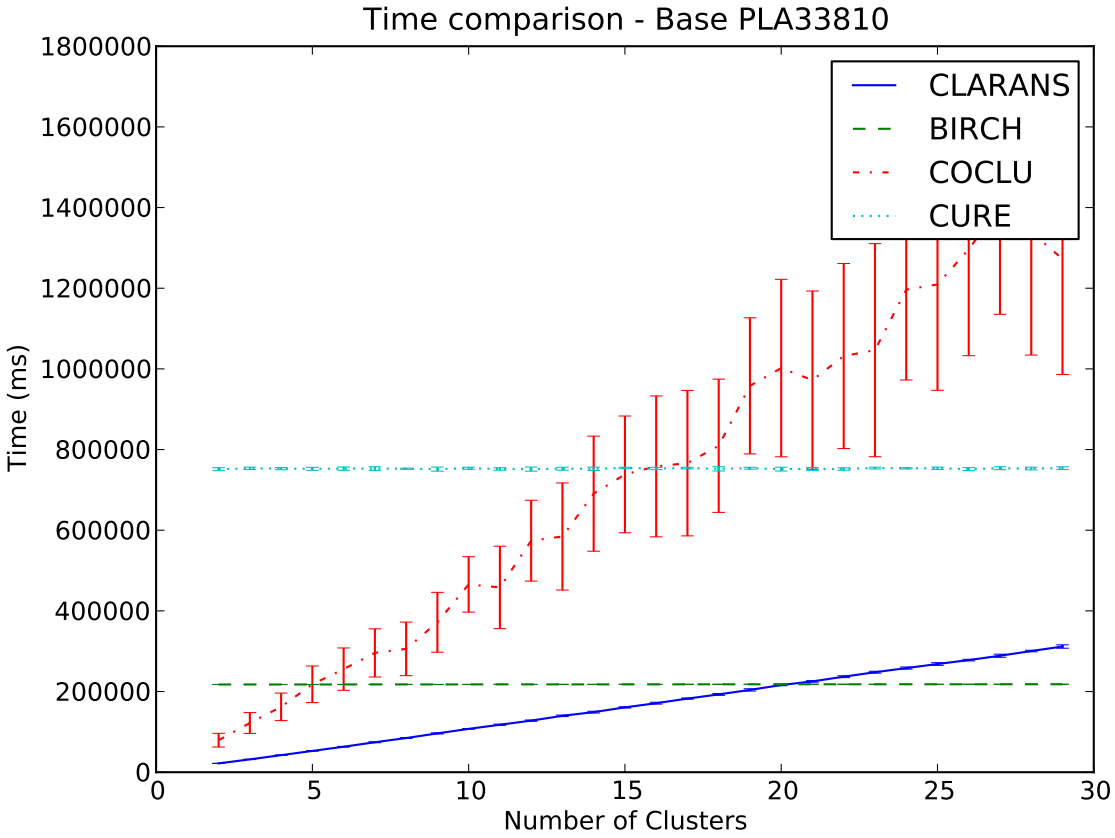


Figure 5.4: Running times of all algorithms on data set PLA33810

The previously observed phenomenon is repeated here. Once again, the BIRCH and CURE algorithms had a more scalable processing time when the number of clusters increases. It is also important to notice that the COCLU algorithm does not deal well with the increase of the value of k. When $k = 15$, or greater, its running time is the worst of all algorithms.

5.3.6 Results for Shuttle Base

This data set also does not contain the number of clusters that must be formed, thus, the set of values in the interval between 2 and 29 was tested to assess both result quality and running

time. The results are presented in the following Figures.

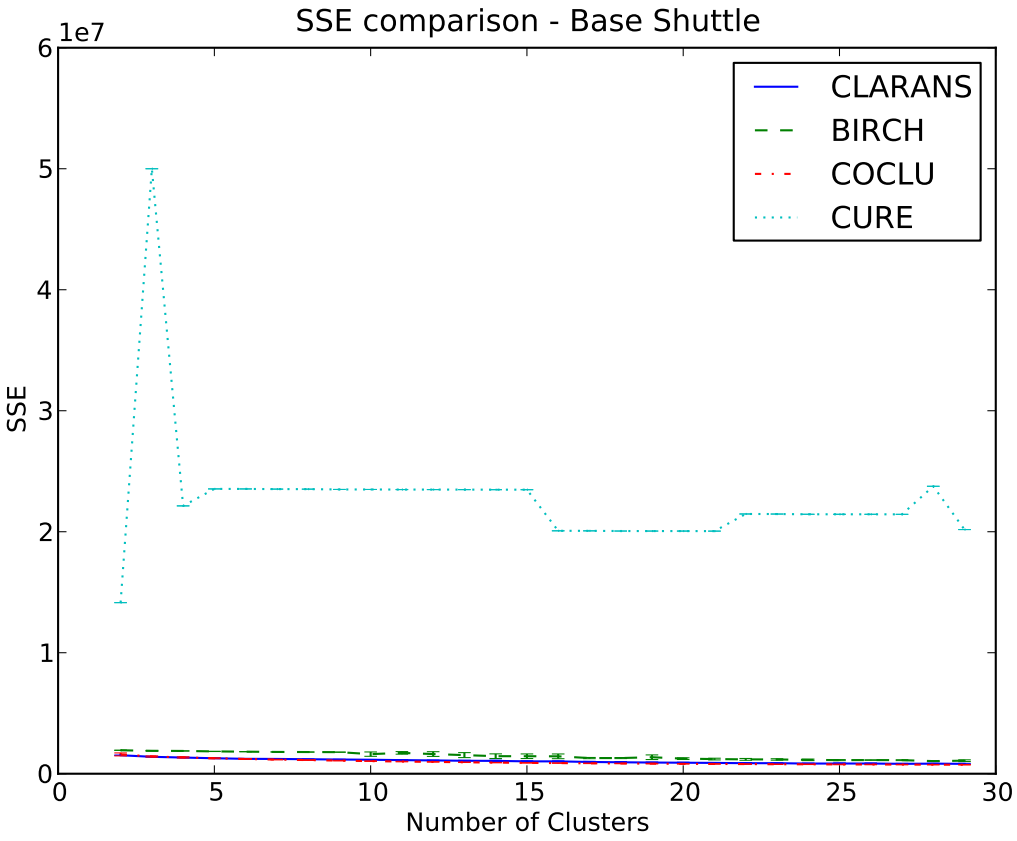


Figure 5.5: The SSE of all tested algorithms, with the exception of the DBSCAN algorithm, on data set Shuttle

From Figure 5.5 it is clear that the CURE algorithm could not deal well with this data set, the result was so poor that it is hard to analyse the other algorithms. For that reason, Figure 5.6 presents the results without the CURE algorithm.

According to the adjusted *Wilcoxon* test, there is a significant difference between algorithms the most values of k tested. In the interval between 2 and 5, the CLARANS algorithm had a better SSE performance than all the others. While in the interval between 6 and 29 the COCLU algorithm had a better performance (p-value < 0.05, confidence interval of 95%). The only exception was when $k = 22$, because it was not possible to detect any difference between algorithms.

The statistical tests also show that the other algorithms results are worse than the COCLU and CLARANS results.

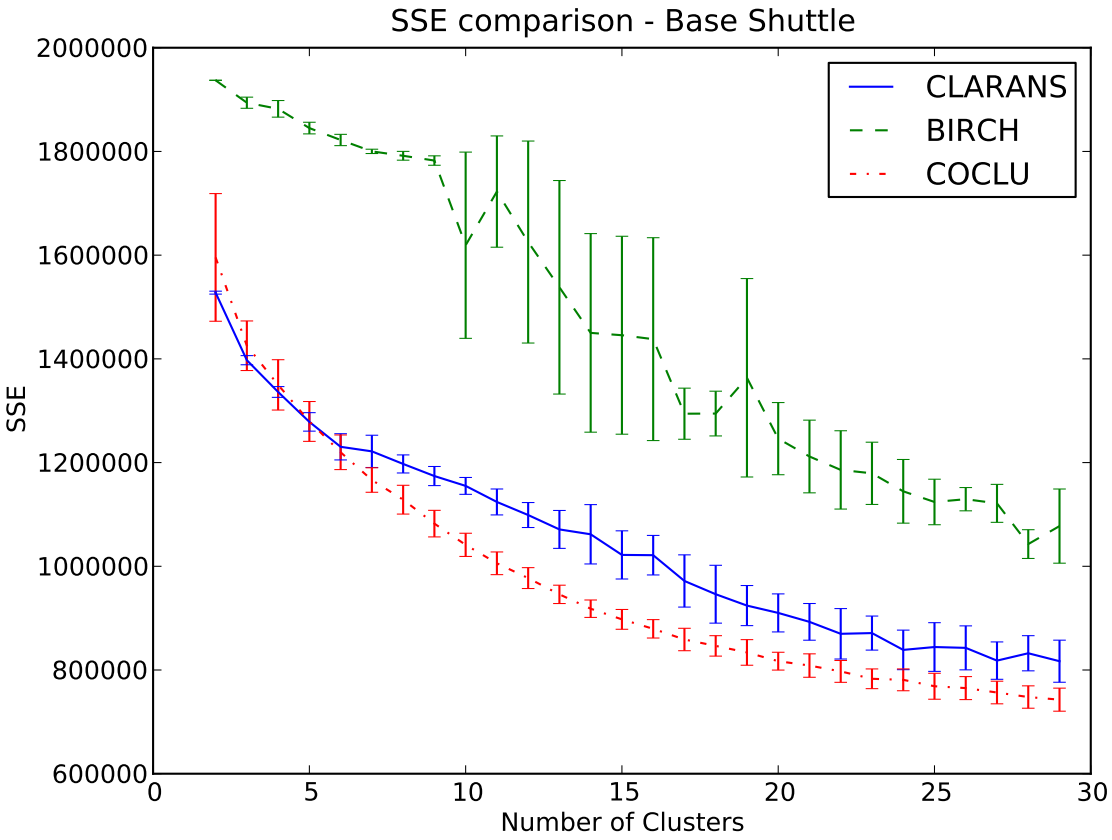


Figure 5.6: The SSE of all tested algorithms, with the exception of the DBSCAN and CURE algorithms (for a better visualization) on data set Shuttle

Figure 5.6 presents the SSE results again without the DBSCAN algorithm. From these results it is clear that the COCLU algorithm had a better clustering result the most part of the experiment, only being slightly worse in the beginning of the experiments.

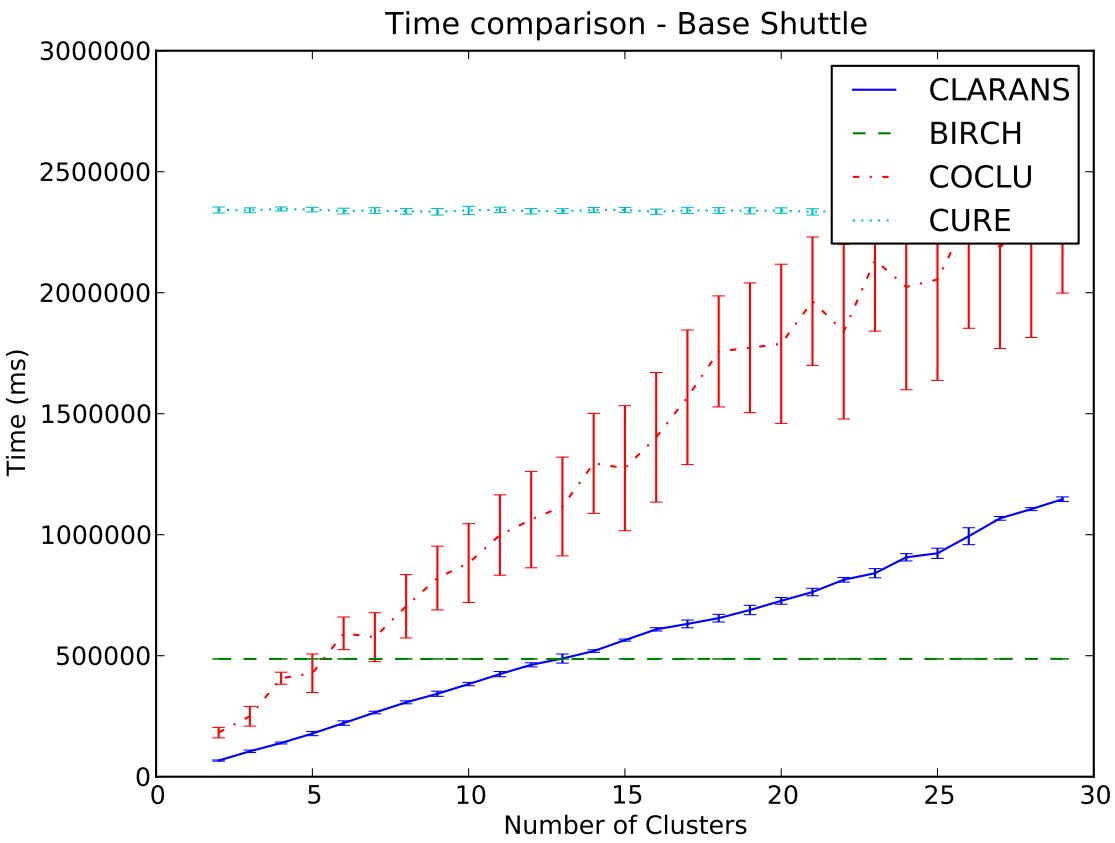


Figure 5.7: Running times of all algorithms, with the exception of the DBSCAN algorithm, on data set Shuttle

As far as time is concerned, this experiment presented a similar behaviour from the previous ones. Once again, the BIRCH and CURE algorithms had a more scalable processing time when the number of clusters increases. But this time, the COCLU algorithm had a better running time than the CURE algorithm for most part of the experiment. In the previous experiment, the COCLU algorithm's running time surpassed the CURE running time when k = 15 , in this experiment it took much longer, only with $k = 29$.

5.3.7 Results for PLA85900 Base

This data set also does not contain the number of clusters that must be formed, thus, the set of values in the interval between 2 and 29 was tested to assess both result quality and running time. The results are presented in the Figures 5.8 and 5.9. Once again, the DBSCAN algorithm could not be run in any of the test suits for this data set. It was not possible to find a proper value for the similarity factor that generated the desired amount of clusters.

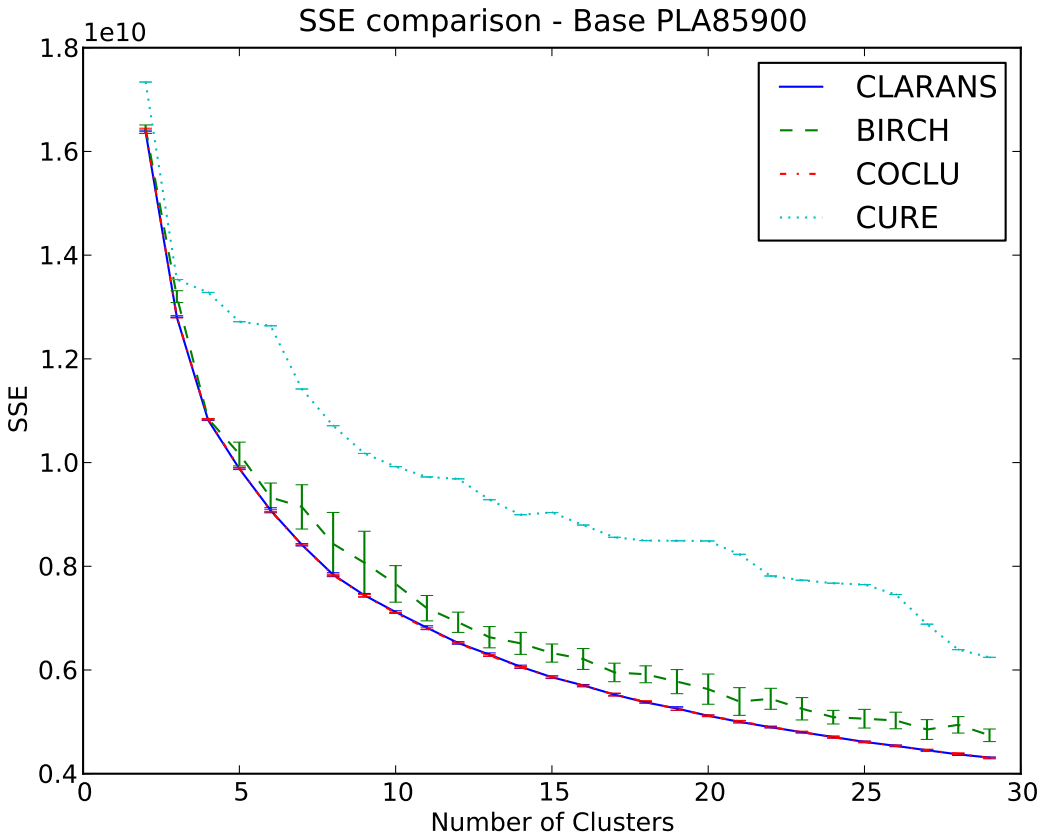


Figure 5.8: The SSE of all tested algorithms, with the exception of the DBSCAN algorithm on data set PLA85900

This result was similar to the ones found for the PLA33810 data set. This is natural due to the similar nature of the data sets (spatial points in a 2D space).

The CLARANS and COCLU algorithm achieved similar clustering results with slight advantage to the CLARANS method. Once again the BIRCH and CURE algorithms were the third and fourth best algorithms. The adjusted *Wilcoxon* test could not find any difference between the CLARANS and COCLU algorithm for any value of k (p-value < 0.05 and confidence interval of 95%). The test was able to detect a significant difference between the COCLU algorithm and the other two algorithms for every value of k.

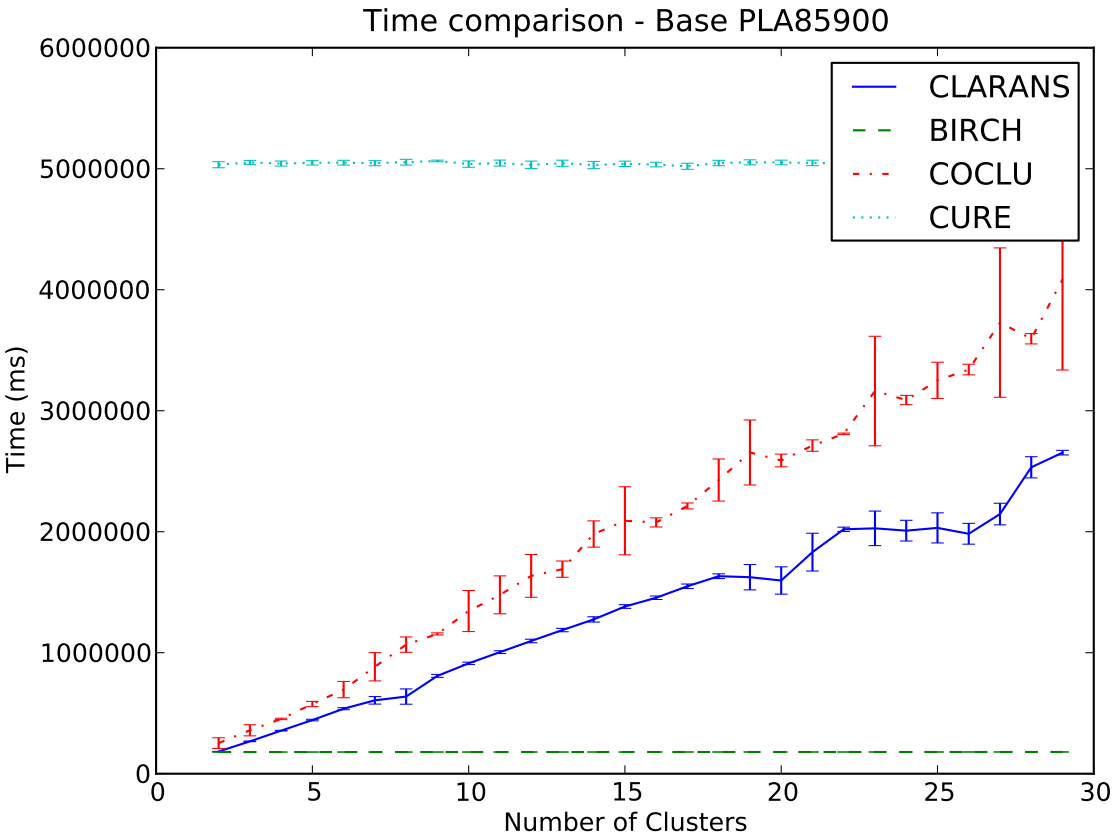


Figure 5.9: Running times of all algorithms on data set PLA85900

Also, the behaviour of running times of the algorithms was similar to those observed on the results of data set PLA33810, but this time the COCLU algorithm had a better performance than the CURE algorithm for every value of k. Its asymptotic behaviour, however, is the worst of all tested algorithms, so the expected running time of the COCLU algorithm will get higher as k increases.

5.3.8 Results for ESCELSA Base

The ESCELSA base is an unique test case constructed with register data of the electricity consumers of the local distributer energy company. This base consists of circa 100000 instances. They represent compact groups of clients with information of their approximate Geo-referential coordinates.

This data set also does not contain the number of clusters that must be formed, thus, the set of values in the interval between 2 and 29 was tested to assess both result quality and running time. The results are presented in the following Figures. Once again, the DBSCAN algorithm

could not be run in any of the test suits for this data set. It was not possible to find a proper value for the similarity factor that generated the desired amount of clusters.

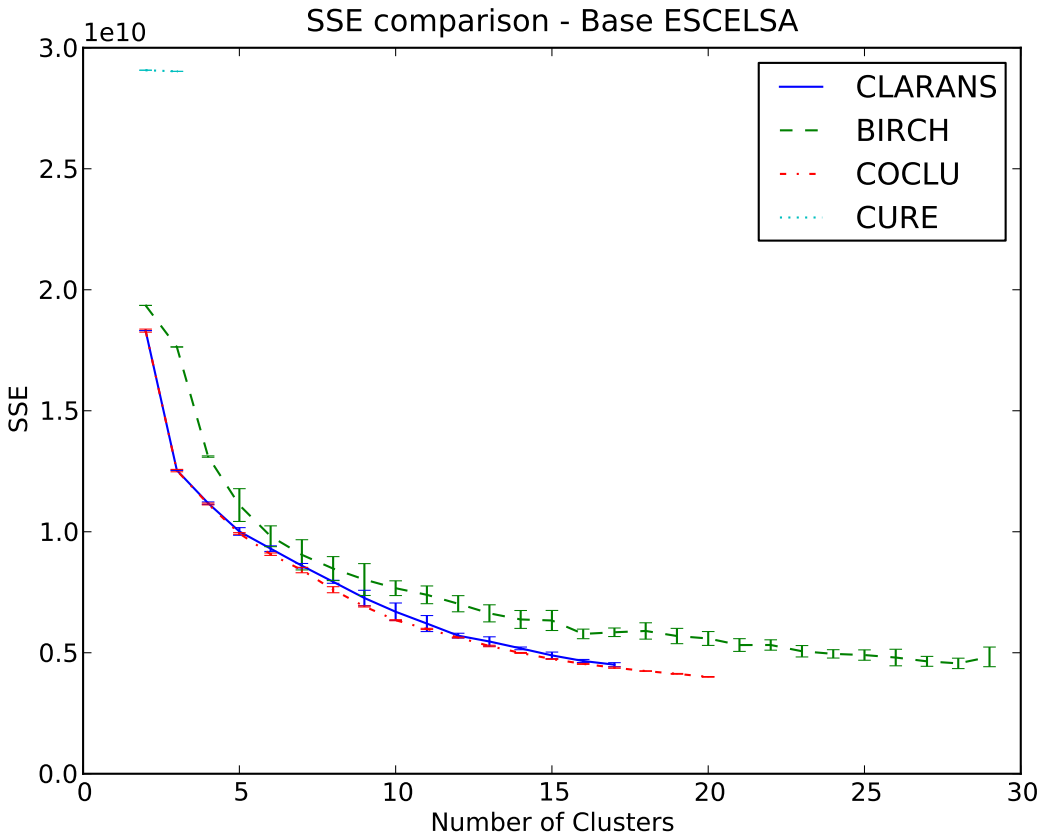


Figure 5.10: The SSE of all tested algorithms, with the exception of the DBSCAN algorithm, on data set ESCELSA

The Figure 5.10 displays the experiments performed on the ESCELSA database. There are only two data points for the CURE algorithm because it took an excessive amount of time to conclude executions. The other algorithms were also preempted due to the same reason. From the available data it is clear that they did not get good results for this data set.

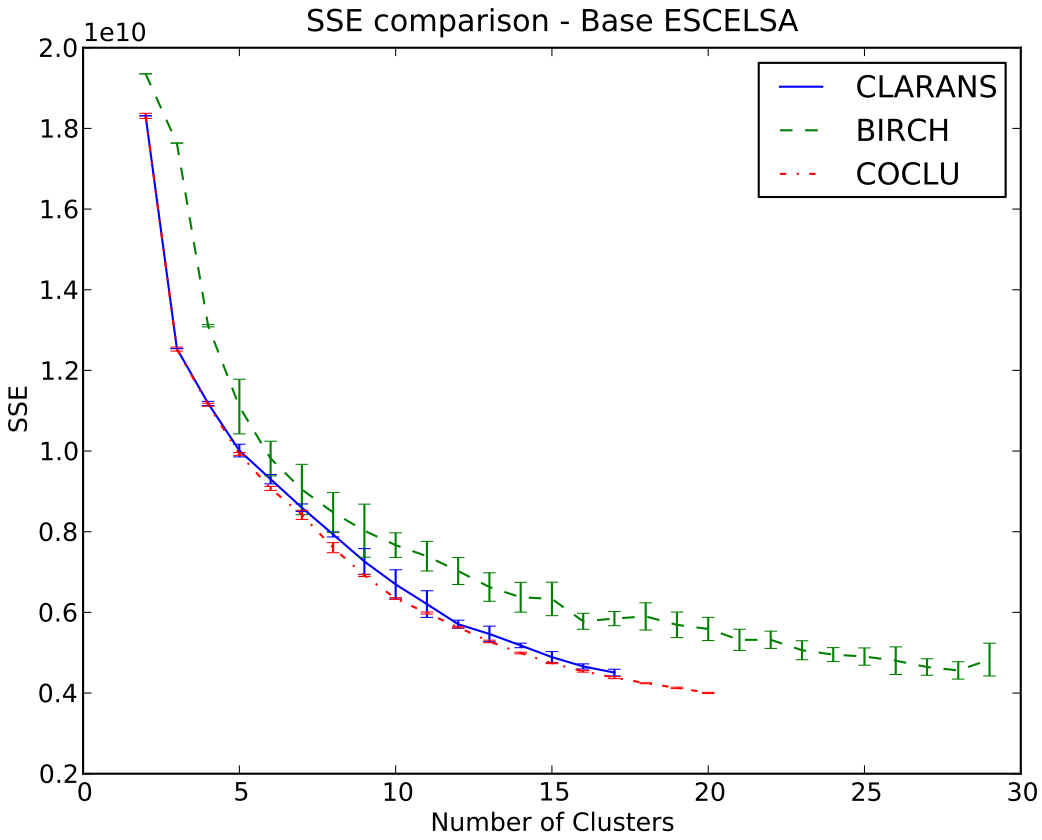


Figure 5.11: The SSE of all tested algorithms, with the exception of the DBSCAN and CURE algorithms (for a better visualization) on data set Shuttle

Figure 5.11 displays the data without the interference of the CURE algorithm. From this graphic it is clear that, once again, the algorithms CLARANS and COCLU had the best results. Also, the BIRCH algorithm had worse, but well-behaved results.

The adjusted *Wilcoxon* test found no difference between the CLARANS and COCLU algorithms when k is 5 or smaller. There is a significant difference between algorithms when k is bigger than 5, in this interval the COCLU algorithm is statistically better than the CLARANS algorithm (p-value < 0.05 and confidence interval of 95%).

Also, both algorithms are better than the BIRCH algorithm for every value of k (p-value $<$ 0.05 and confidence interval of 95%).

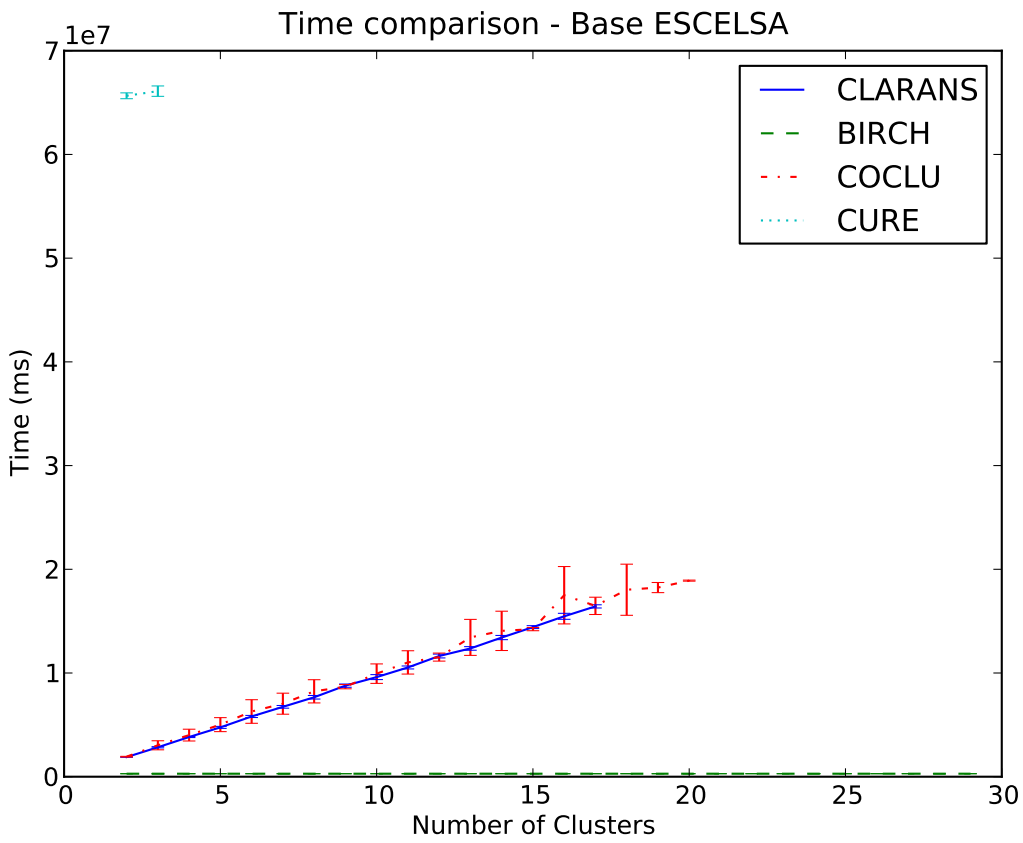


Figure 5.12: Running times of all algorithms, with the exception of the DBSCAN algorithm, on data set ESCELSA

Figure 5.12 displays the running times for the tested algorithms. From this Figure it is clear that the CURE algorithm had a much worse running time. In Figure 5.13, the data is displayed again with the exception of the CURE running time, for a better visualization.

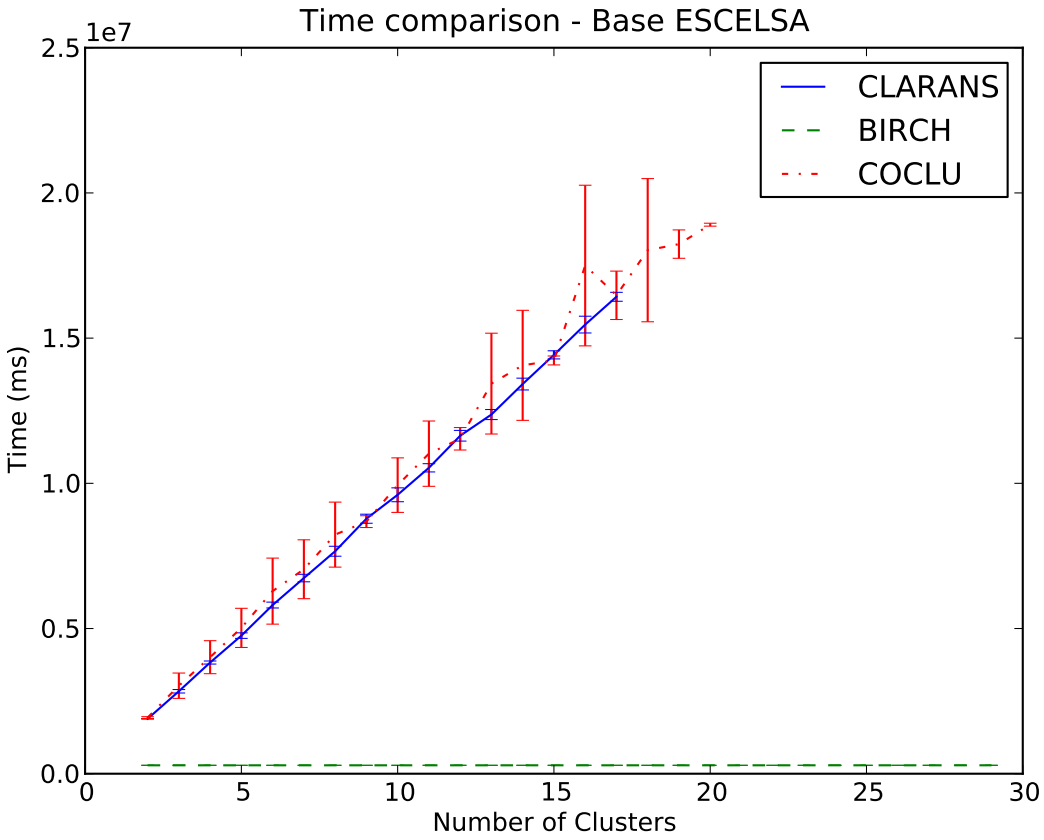


Figure 5.13: Running times of all algorithms, with the exception of the DBSCAN algorithm and CURE algorithms (for a better visualization), on data set ESCELSA

It is possible to observe in Figure 5.13 that the runtimes of the CLARANS and COCLU algorithm are very similar, different from the other data sets. It is also possible to notice the almost negligible time the BIRCH algorithm took to run, compared to the running times of the other algorithms, its performance regarding solution quality, however, was poor.

5.3.9 Results for MiniBooNE Base

In this data set the only algorithms that were capable of successfully returning valid results in reasonable running times were the CLARANS and COCLU algorithms.

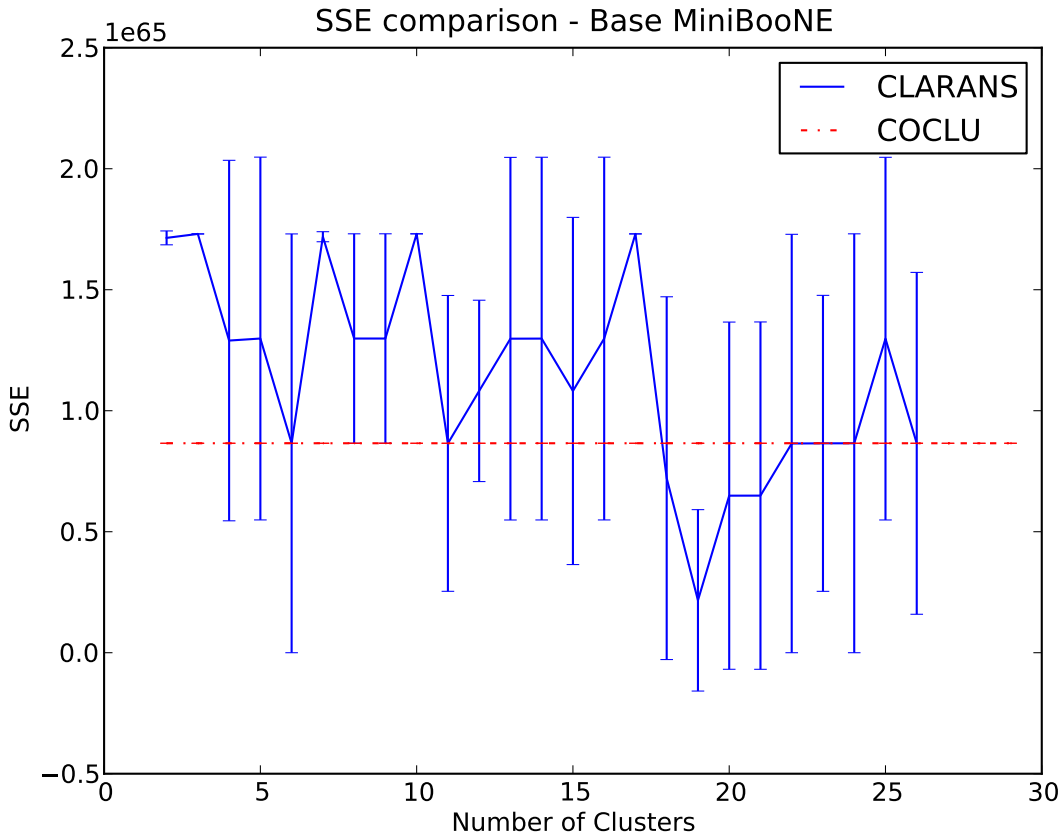


Figure 5.14: The SSE CLARANS and COCLU, on data set MiniBooNE

The results observed on Figure 5.14 are quite different from those observed in the previous tests. The results for the CLARANS algorithm were quite erratic and a straight line was observed for the COCLU algorithm. This is probably due to the difficulty of the problem. The CLARANS algorithm did not behaved as expected in this particular data base.

The adjusted *Wilcoxon* test could find differences in algorithms when $k = 2$, $k = 3$, $k = 7$, $k = 10$, $k = 17$ and $k = 19$ (p-value < 0.05 and confidence interval of 95%).

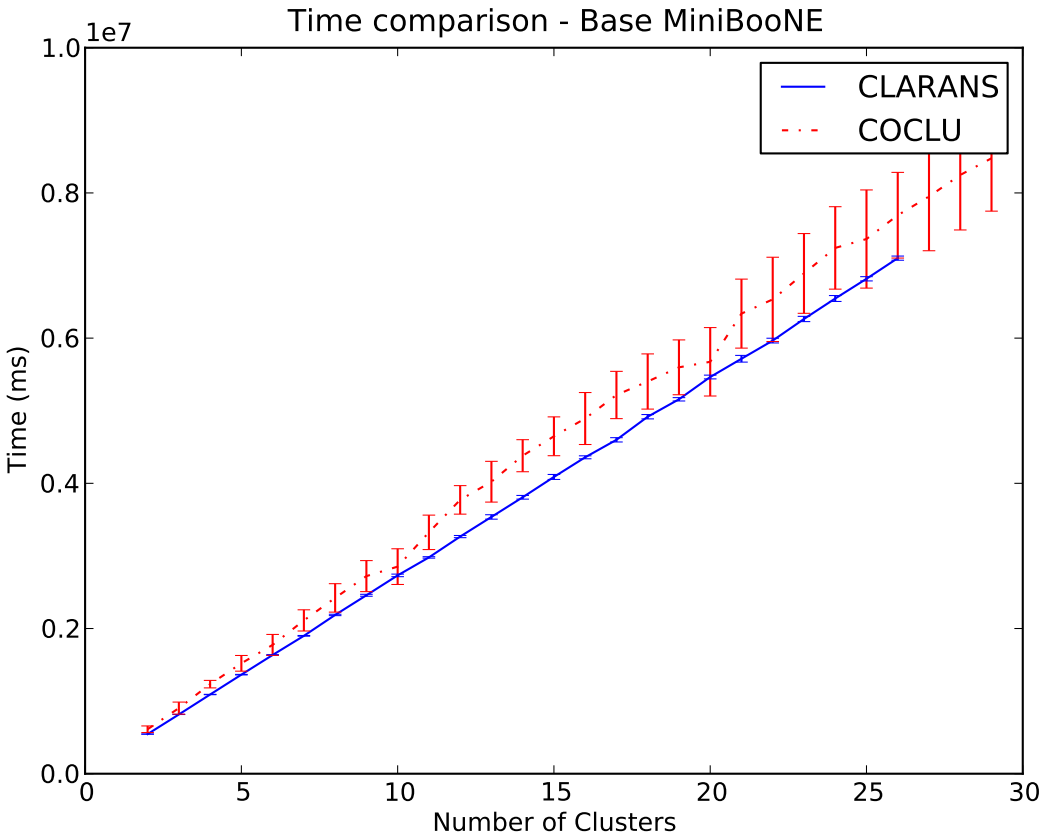


Figure 5.15: Running times of CLARANS and COCLU algorithms, on data set MiniBooNE

The results for the running time observed in Figure 5.15, however, are coherent with the observed run-times of the previous data sets.

Chapter 6

Conclusion and Future Work

This work proposes a new approach for dealing with the problem of clustering large data sets of continuous points in metric spaces. The objective of the clustering task is to find groups on data that minimize the *Sum of Squared Errors* between instances of a group and their centroids.

Four classical algorithms (BIRCH, CLARANS, CURE, DBSCAN) were presented with their background, pseudo-code, analysis of the necessary data structures, advantages and disadvantages.

A novel clustering algorithm for large data sets, called COCLU, was developed. The idea is to apply classical algorithms that perform well in small data sets on a subset of the original data. The problem of choosing a good, representative subset of points is solved by applying a co-evolutive min-max approach for the problem of selecting representative points.

The co-evolutive competitive min-max algorithm is commonly used for solving problems with two or more coupled and opposed objective functions. This approach suits well the problem of finding hard set of points to cluster, given some pre-configured algorithms. The adaptation was not hard to implement.

Two populations were created, one of clustering algorithms and another of selected points. The algorithm works by evolving both populations at the same time in a competitive manner. The expected behaviour is that both populations converge to a good set of representative points and clustering algorithms.

6.1 Experiments

From the experiments it is clear that no algorithm had a overall better result for every data set considering both response time and result quality. However, for the most part of the data sets tested, the CLARANS and COCLU algorithms had very competitive running times and were always among the best algorithms in regard to solution quality.

It is important to notice that the relative running times of the COCLU algorithm compared to the CLARANS algorithm decreased as the size of the data sets increased. I.e., the bigger the data set, the less advantageous is to use the CLARANS algorithm. It is noticeable that the time registered for both CLARANS and COCLU algorithm (Table 5.4) were very different from one another. The COCLU algorithm had longer running times in the first data sets than in the last ones. In the larger data sets both running times and solution quality had very similar results (Figure 5.12 and Figure 5.10).

The conclusion is that the COCLU algorithm is equivalent to the CLARANS algorithm in most of data sets and it has a better response as the data set increases. The CLARANS algorithm seems to not handle well data with an increased number of dimensions (Table 5.5 and Figure 5.14 present this phenomena). When the data set and dimensions are small, the algorithm performs very well, however, an erratic behaviour is observed when they increase.

From the tests the premise that no single clustering algorithm is better for every base is verified. Thus, an algorithm capable of unifying the decisions of many different heuristics is relevant. The COCLU algorithm successfully fills the gap between clustering techniques and data sets by using co-evolution for selecting both points and algorithms to consider in the overall clustering process. This automatic selecting yields a robust algorithm for clustering many different types of data sets, given that the adequate clustering algorithm is present in the population pool.

6.2 Future Work

The use of the min-max approach for solving complex problems often leads to algorithms with poor scalability capabilities. The constant need of evaluating an individual of one population against all individuals of the other results in high processing times. Two solutions are available for mitigating this problem.

The first solution is to use parallel algorithms for performing the fitness calculation. Fixing one individual of population A, all the fitness calculations on the opposite population may be

done at the same time, reducing the runtime of the algorithm greatly. The second approach is to change the algorithm of fitness calculation in the following way: given an individual $\vec{a}$ of population A, instead of iterating over all individuals of population B, randomly choose a fixed number of individuals for calculating the fitness of individual $\vec{a}$.

With this enhancement, more tests can be executed with even bigger data sets with different characteristics to verify if the observed behaviour is maintained.

More investigation may be done on selecting clustering algorithms for the core of the COCLU algorithm, changing the current set of algorithms (k-means, CLARANS and Spectral clustering) may improve the final result of the algorithm.

It is also possible to develop a way for iteratively changing crucial parameters of the COCLU algorithm based on the partial results of the co-evolution process. The compression factor, for instance, is very important for the overall result. Fixing it for every data set possible will not yield the best results.

It is also possible to use two other approaches for reducing the complexity of the COCLU algorithm: analyze the features of the data set to eliminate redundant characteristics, and therefore reducing the complexity of the overall procedure, and use approximate k-NN queries for finding the closest elements of a given point in space, also reducing the overall complexity.

Bibliography

ACHTERT, E.; BOHM, C.; KROGER, P. Deliclu: Boosting robustness, completeness, usability, and efficiency of hierarchical clustering by a closest pair ranking. In: *Proc. 10th Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD'06)*. Singapore: DBS, 2006. p. 119–128.

AGRAWAL, R.; SRIKANT, R. Fast algorithms for mining association rules in large databases. In: *Proceedings of the 20th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994. (VLDB '94), p. 487–499. ISBN 1-55860-153-8. Available from Internet: <http://dl.acm.org/citation.cfm?id=645920.672836>.

ALOISE, D. et al. NP-hardness of Euclidean sum-of-squares clustering. *Machine Learning*, Springer, Netherlands, v. 75, p. 245–248, 2009.

ALSABTI, K.; RANKA, S.; SINGH, V. An efficient k-means clustering algorithm. *Electrical Engineering and Computer Science*, n. 43, 1997. Available from Internet: <http://surface.syr.edu/eecs/43>.

AMIGÓ, E. et al. A comparison of extrinsic clustering evaluation metrics based on formal constraints. *Inf. Retr.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 12, n. 4, p. 461–486, ago. 2009. ISSN 1386-4564. Available from Internet: <http://dx.doi.org/10.1007/s10791-008-9066-8>.

ANKERST, M. et al. OPTICS: Ordering points to identify the clustering structure. In: *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 1999. p. 49–60.

ARTHUR, D.; VASSILVITSKII, S. How slow is the k-means method? In: *Proceedings of the twenty-second annual symposium on Computational geometry*. New York, NY, USA: ACM, 2006. (SCG '06), p. 144–153. ISBN 1-59593-340-9. Available from Internet: <http://doi.acm.org/10.1145/1137856.1137880>.

ARTHUR, D.; VASSILVITSKII, S. k-means++: the advantages of careful seeding. In: *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2007. (SODA '07), p. 1027–1035. ISBN 978-0-898716-24-5. Available from Internet: <http://dl.acm.org/citation.cfm?id=1283383.1283494>.

ARYA, S. et al. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, ACM, New York, NY, USA, v. 45, n. 6, p. 891–923, nov. 1998. ISSN 0004-5411. Available from Internet: <http://doi.acm.org/10.1145/293347.293348>.

AUGUSTO, D. A.; BARBOSA, H. J.; EBECKEN, N. F. Coevolution of data samples and classifiers integrated with grammatically-based genetic programming for data classification. In: *Proceedings of the 10th annual conference on Genetic and evolutionary computation*. New York, NY, USA: ACM, 2008. (GECCO '08), p. 1171–1178. ISBN 978-1-60558-130-9. Available from Internet: <http://doi.acm.org/10.1145/1389095.1389328>.

BANERJEE, A. et al. Model-based overlapping clustering. In: *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*. New York, NY, USA: ACM, 2005. (KDD '05), p. 532–537. ISBN 1-59593-135-X. Available from Internet: <http://doi.acm.org/10.1145/1081870.1081932>.

BARBOSA, H. A coevolutionary genetic algorithm for constrained optimization. *CEC 99. Proceedings of the 1999 Congress on Evolutionary Computation*, v. 3, p. 1611 Vol. 3, 1999.

BARBOSA, H. J. C. A genetic algorithm for min-max problems. *Proccedings of the First International Conference on Evolutionary Computation and its Applications*, Moscow, Russia, p. 99–109, 1996.

BEIL, F.; ESTER, M.; XU, X. Frequent term-based text clustering. In: *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. New York, NY, USA: ACM, 2002. (KDD '02), p. 436–442. Available from Internet: <http://doi.acm.org/10.1145/775047.775110>.

BENSMAIL, H. et al. A novel approach for clustering proteomics data using bayesian fast fourier transform. *Bioinformatics*, v. 21, n. 10, p. 2210–2224, 2005. Available from Internet: <http://bioinformatics.oxfordjournals.org/content/21/10/2210.abstract>.

BERCHTOLD, S. et al. Fast nearest neighbor search in high-dimensional space. In: *Proceedings of the 14th International Conference on Data Engineering*. Orlando, Florida: IEEE, 1998. p. 209–218. ISSN 1063-6382.

BERGH, F. van den; ENGELBRECHT, A. A cooperative approach to particle swarm optimization. *IEEE Transactions on Evolutionary Computation*, v. 8, n. 3, p. 225–239, june 2004.

BERKHIN, P. A Survey of Clustering Data Mining Techniques. In: KOGAN, J.; NICHOLAS, C.; TEBOULLE, M. (Ed.). Berlin/Heidelberg: Springer Berlin Heidelberg, 2006. p. 25–71. ISBN 3-540-28348-X. Available from Internet: <http://dx.doi.org/10.1007/3-540-28349-8_2>.

CHAKRABARTI, D.; KUMAR, R.; TOMKINS, A. Evolutionary clustering. In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. New York, NY, USA: ACM, 2006. (KDD '06), p. 554–560. ISBN 1-59593-339-5. Available from Internet: <http://doi.acm.org/10.1145/1150402.1150467>.

CHEUNG, K. L.; FU, A. W.-C. Enhanced nearest neighbour search on the R-tree. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 27, n. 3, p. 16–21, set. 1998. ISSN 0163-5808. Available from Internet: <http://doi.acm.org/10.1145/290593.290596>.

CIACCIA, P.; PATELLA, M. PAC nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In: *Proceedings of the 16th International Conference on Data Engineering.* San Diego, CA: IEEE, 2000. p. 244–255.

DEMSAR, J. Statistical comparisons of classifiers over multiple data sets. *J. Mach. Learn. Res.*, JMLR.org, v. 7, p. 1–30, dez. 2006. ISSN 1532-4435.

DHILLON, I. S. Co-clustering documents and words using bipartite spectral graph partitioning. In: *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. New York, NY, USA: ACM, 2001. (KDD '01), p. 269–274. ISBN 1-58113-391-X. Available from Internet: <http://doi.acm.org/10.1145/502512.502550>.

EDGAR, R. C. Search and clustering orders of magnitude faster than BLAST. *Bioinformatics*, v. 26, n. 19, p. 2460–2461, 2010. Available from Internet: <http://bioinformatics.oxfordjournals.org/content/26/19/2460.abstract>.

FABRIS, F.; KROHLING, R. A. A co-evolutionary differential evolution algorithm for solving min-max optimization problems implemented on gpu using c-cuda. *Expert Systems with Applications*, n. 0, 2011. ISSN 0957-4174. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0957417411015004>.

FERHATOSMANOGLU, H. et al. Constrained nearest neighbor queries. In: JENSEN, C. et al. (Ed.). *Advances in Spatial and Temporal Databases*. [S.l.]: Springer Berlin / Heidelberg, 2001. (Lecture Notes in Computer Science, v. 2121), p. 257–276. ISBN 978-3-540-42301-0. 10.1007/3-540-47724-1_14.

FICICI, S. G. *Solution concepts in coevolutionary algorithms*. Tese (Doutorado), Waltham, MA, USA, 2004. AAI3127125.

FRANK, A.; ASUNCION, A. *UCI Machine Learning Repository*. 2010. Available from Internet: <http://archive.ics.uci.edu/ml>.

FRANTI, P.; VIRMAJOKI, O.; KAUKORANTA, T. Branch-and-bound technique for solving optimal clustering. In: *Proceedings of the 16th International Conference on Pattern Recognition, 2002*. [S.l.]: IEEE, 2002. v. 2, p. 233–235.

GUHA, S.; RASTOGI, R.; SHIM, K. Cure: an efficient clustering algorithm for large databases. In: *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1998. (SIGMOD '98), p. 73–84. ISBN 0-89791-995-5. Available from Internet: <http://doi.acm.org/10.1145/276304.276312>.

GUSTAFSON, D. E.; KESSEL, W. C. Fuzzy clustering with a fuzzy covariance matrix. In: *IEEE Conference on Decision and Control including the 17th Symposium on Adaptive Processes*. [S.l.]: IEEE, 1978. v. 17, p. 761 –766.

GUTTMAN, A. R-trees: A dynamic index structure for spatial searching. In: *International conference on management of data*. [S.l.]: ACM, 1984. p. 47–57.

HARRINGTON, J.; SALIBIáN-BARRERA, M. Finding approximate solutions to combinatorial problems with very large data sets using birch. *Comput. Stat. Data Anal.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 54, n. 3, p. 655–667, mar. 2010. ISSN 0167-9473. Available from Internet: <http://dx.doi.org/10.1016/j.csda.2008.08.001>.

HE, Q.; WANG, L. An effective co-evolutionary particle swarm optimization for constrained engineering design problems. *Engineering Applications of Artificial Intelligence*, v. 20, n. 1, p. 89 – 99, 2007. ISSN 0952-1976. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0952197606000686>.

HE, Z.; XU, X.; DENG, S. Scalable algorithms for clustering large datasets with mixed type attributes. *International Journal of Intelligent Systems*, Wiley Subscription Services, Inc., A Wiley Company, v. 20, n. 10, p. 1077–1089, 2005. Available from Internet: <http://dx.doi.org/10.1002/int.20108>.

HJALTASON, G. R.; SAMET, H. Incremental distance join algorithms for spatial databases. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 27, n. 2, p. 237–248, jun. 1998. ISSN 0163-5808. Available from Internet: <http://doi.acm.org/10.1145/276305.276326>.

JAIN, A. K.; MURTY, M. N.; FLYNN, P. J. Data clustering: a review. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 31, n. 3, p. 264–323, set. 1999. Available from Internet: <http://doi.acm.org/10.1145/331499.331504>.

KANUNGO, T. et al. A local search approximation algorithm for k-means clustering. In: *Proceedings of the eighteenth annual symposium on Computational geometry*. New York, NY, USA: ACM, 2002. (SCG '02), p. 10–18. ISBN 1-58113-504-1. Available from Internet: <http://doi.acm.org/10.1145/513400.513402>.

KAUFMAN, L.; ROUSSEEUW, P. J. *Finding Groups in Data: An Introduction to Cluster Analysis (Wiley Series in Probability and Statistics)*. [S.l.]: Wiley-Interscience, 2005. Paperback. ISBN 0471735787.

KORN, F.; MUTHUKRISHNAN, S. Influence sets based on reverse nearest neighbor queries. In: *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2000. (SIGMOD '00), p. 201–212. ISBN 1-58113-217-4. Available from Internet: <http://doi.acm.org/10.1145/342009.335415>.

LECHEVALLIER, Y.; VERDE, R.; CARVALHO, F. Symbolic clustering of large datasets. In: BATAGELJ, V. et al. (Ed.). *Data Science and Classification*. [S.l.]: Springer Berlin Heidelberg, 2006, (Studies in Classification, Data Analysis, and Knowledge Organization). p. 193–201.

LIEW, A.; YAN, H. An adaptive spatial fuzzy clustering algorithm for 3-D MR image segmentation. *IEEE Transactions on Medical Imaging*, v. 22, n. 9, p. 1063 –1075, sept. 2003.

LIU, B. et al. A memetic co-evolutionary differential evolution algorithm for constrained optimization. *IEEE Congress on Evolutionary Computation. CEC 2007.*, p. 2996–3002, Sept. 2007.

LIU, D.-Z.; LIM, E.-P.; NG, W.-K. *Efficient k Nearest Neighbor Queries on Remote Spatial Databases Using Range Estimation*. 2002.

LUO, Y. et al. *Analyzing and Improving Clustering Based Sampling for Microprocessor Simulation*. 2008.

MACQUEEN, J. B. Some methods for classification and analysis of multivariate observations. In: CAM, L. M. L.; NEYMAN, J. (Ed.). *Proc. of the fifth Berkeley Symposium on Mathematical Statistics and Probability*. [S.l.]: University of California Press, 1967. v. 1, p. 281–297.

MAHER, M. L.; POON, J.; BOULANGER, S. *Formalising Design Exploration As Co-Evolution: A Combined Gene Approach*. 1996.

MARZOUK, Y. M.; GHONIEM, A. F. K-means clustering for optimal partitioning and dynamic load balancing of parallel hierarchical n-body simulations. *Journal of Computational Physics*, v. 207, n. 2, p. 493 – 528, 2005.

MICHALEWICZ, Z. Evolutionary computation techniques for nonlinear programming problems. In: *International Transactions in Operational Research*. University of North Carolina, USA: Elsevier Science, 1994. v. 1, n. 2, p. 223–240. Available from Internet: <http://dx.doi.org/10.1111/1475-3995.d01-23>.

MICHALEWICZ, Z.; JANIKOW, C. Z. Genocop: a genetic algorithm for numerical optimization problems with linear constraints. *Commun. ACM*, ACM, New York, NY, USA, v. 39, n. 12es, dec 1996.

MICHALEWICZ, Z.; NAZHIYATH, G. Genocop III: A co-evolutionary algorithm for numerical optimization problems with nonlinear constraints. *Proceedings of IEEE International Conference on Evolutionary Computation*, IEEE Press, p. 647–651, 1995.

MOORE, A. *An introductory tutorial on kd-trees*. Pittsburgh, PA, 1991.

NA, J. P.; LOZANO, J.; NAGA, P. L. An empirical comparison of four initialization methods for the k-means algorithm. *Pattern Recognition Letters*, v. 20, n. 10, p. 1027–1040, 1999. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0167865599000690>.

NG, A. Y.; JORDAN, M. I.; WEISS, Y. On spectral clustering: Analysis and an algorithm. In: *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS*. [S.l.]: MIT Press, 2001. p. 849–856.

NG, R. T.; HAN, J. CLARANS: A Method for Clustering Objects for Spatial Data Mining. *IEEE Trans. on Knowl. and Data Eng.*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 14, n. 5, p. 1003–1016, set. 2002. ISSN 1041-4347. Available from Internet: <http://dx.doi.org/10.1109/TKDE.2002.1033770>.

OSTROVSKY, R.; RABANI, Y. Polynomial time approximation schemes for geometric k-clustering. In: *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*. New York, NY, USA: ACM, 2000. p. 349–358.

PAREDIS, J. Co-evolutionary constraint satisfaction. In: *Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature*. London, UK: Springer-Verlag, 1994. p. 46–55. ISBN 3-540-58484-6.

POON, J.; MAHER, M. Co-evolution and emergence in design. *Artificial Intelligence in Engineering*, v. 11, n. 3, p. 319 – 327, 1997. ISSN 0954-1810. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0954181096000477>.

PUDIL, P.; NOVOVICOVA, J.; KITTLER, J. Floating search methods in feature selection. *Pattern Recognition Letters*, v. 15, n. 11, p. 1119 – 1125, 1994. ISSN 0167-8655. Available from Internet: <http://www.sciencedirect.com/science/article/pii/0167865594901279>.

PUNJ, G.; STEWART, D. W. Cluster analysis in marketing research: Review and suggestions for application. *Journal of Marketing Research*, American Marketing Association, v. 20, n. 2, p. 134, 1983. Available from Internet: <http://www.jstor.org/stable/3151680?origin=crossref>.

REINELT, G. TSPLIB- a traveling salesman problem library. *ORSA Journal of Computing*, v. 3, n. 4, p. 376–384, 1991.

RUSSELL, D. et al. A grammar-based distance metric enables fast and accurate clustering of large sets of 16s sequences. *BMC Bioinformatics*, BioMed Central, v. 11, p. 1–14, 2010. ISSN 1471-2105. 10.1186/1471-2105-11-601. Available from Internet: <http://dx.doi.org/10.1186/1471-2105-11-601>.

SANDER, J. et al. Density-based clustering in spatial databases: The algorithm gdbscan and its applications. *Data Min. Knowl. Discov.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 2, n. 2, p. 169–194, jun. 1998. ISSN 1384-5810. Available from Internet: <http://dx.doi.org/10.1023/A:1009745219419>.

SHERLOCK, G. Analysis of large-scale gene expression data. *Current Opinion in Immunology*, v. 12, n. 2, p. 201 – 205, 2000. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0952791599000746>.

STEINHAUS, H. Sur la division des corp materiels en parties. *Bull. Acad. Polon. Sci*, v. 1, p. 801–804, 1956.

STREHL, A.; GHOSH, J. Cluster ensembles — a knowledge reuse framework for combining multiple partitions. *J. Mach. Learn. Res.*, JMLR.org, v. 3, p. 583–617, mar. 2003. ISSN 1532-4435. Available from Internet: <http://dx.doi.org/10.1162/153244303321897735>.

TANG, L. R. et al. Cluster analysis of simulated gravitational wave triggers using s-means and constrained validation clustering. *Classical and Quantum Gravity*, v. 25, n. 18, p. 184023, 2008. Available from Internet: <http://stacks.iop.org/0264-9381/25/i=18/a=184023>.

VEGA, W. F. de la et al. Approximation schemes for clustering problems. In: *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 2003. (STOC '03), p. 50–58. ISBN 1-58113-674-9. Available from Internet: <http://doi.acm.org/10.1145/780542.780550>.

WOLD, S.; ESBENSEN, K.; GELADI, P. Principal component analysis. *Chemometrics and Intelligent Laboratory Systems*, v. 2, n. 1-3, p. 37 – 52, 1987. ISSN 0169-7439. Available from Internet: <http://www.sciencedirect.com/science/article/pii/0169743987800849>.

WONG, T.-T.; LUK, W.-S.; HENG, P.-A. Sampling with hammersley and halton points. *J. Graph. Tools*, A. K. Peters, Ltd., Natick, MA, USA, v. 2, n. 2, p. 9–24, 1997. ISSN 1086-7651.

XIA, Y. et al. Image segmentation by clustering of spatial patterns. *Pattern Recogn. Lett.*, Elsevier Science Inc., New York, NY, USA, v. 28, n. 12, p. 1548–1555, set. 2007. ISSN 0167-8655. Available from Internet: <http://dx.doi.org/10.1016/j.patrec.2007.03.012>.

XIANG, Z.; JOY, G. Color image quantization by agglomerative clustering. *IEEE Comput. Graph. Appl.*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 14, n. 3, p. 44–48, maio 1994. ISSN 0272-1716. Available from Internet: <http://dx.doi.org/10.1109/38.279043>.

YAO, B.; LI, F.; KUMAR, P. K nearest neighbor queries and kNN-Joins in large relational databases (almost) for free. In: *26th International Conference on Data Engineering (ICDE)*. Long Beach, California: IEEE, 2010. p. 4–15.

ZHANG, T.; RAMAKRISHNAN, R.; LIVNY, M. Birch: an efficient data clustering method for very large databases. In: *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1996. (SIGMOD '96), p. 103–114. ISBN 0-89791-794-4. Available from Internet: <http://doi.acm.org/10.1145/233269.233324>.