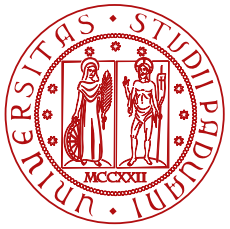

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAURA MAGISTRALE IN
COMPUTER ENGINEERING

Representative Itemsets Mining: A Clustering Approach

MASTER CANDIDATE

Giacomo Seno

Student ID 2054138

SUPERVISOR

Prof. Fabio Vandin

University of Padua

ACADEMIC YEAR 2022-2023
Date 27/11/2023

To my family and
my patient girlfriend

Abstract

In this thesis we study the problem of finding a good representation of a database of transactions. Previous works propose an approach that relies on a lossless compression of the database. This thesis focuses instead on a lossy compression of the database and studies a clustering approach. Given a set of transactions, the clustering model we will present tries to find the best representative itemsets by considering them as clusters. What defines the clustering model is an objective function that minimizes the number of clusters and tries to obtain the best clustering by assigning to each representative itemset some subtransactions taken from the input database. In this document we will present our algorithm and its results on synthetic datasets.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

In this chapter we will briefly introduce the context in which we will operate and we will present the general idea of the thesis and its organization.

1.1 Context

The elements that characterize the context of our work are essentially the following:

- a set of items $\mathcal{I}$,
- a set db of transactions t where each transaction is a set of elements taken from $\mathcal{I}$.

The most famous interpretation of the elements we just described is that:

- $\mathcal{I}$ represents the set of items for sale in a certain store,
- t represents the set of items a client bought at the store,
- db represents a set containing the different transactions on a specific day.

Our input is therefore the database db of transactions from which we want to extract some informations.
However, the database size may be very large and so those informations are not easy to extrapolate in terms of time, or easy to find at all.

1.2 General idea

Many different methods have been proposed to find different kind of patterns in large databases. However, we approach the problem from a different point of view.
In our case, the informations we want to extract are just composed by a set of itemsets $\mathcal{D}$ where an itemset is simply a set of items taken from $\mathcal{I}$ that may or may not be a subset of a transaction t of the database db.
The goal of this thesis is to find a set $\mathcal{D}$ that is a good representation of the input database db keeping the size of $\mathcal{D}$ low.
To achieve this goal we mainly related to two very different works that helped us developing our procedure.

1.2.1 Itemsets that compress

The first work is "Item Sets that Compress" [1] and it faces the main problem of our work, that is, the explosion of the number of results. What they want to achieve, in fact, is to extract from the database the frequent itemsets, that are itemsets that occur in the database with high frequency.
In order to do so, they have to consider a very big number of itemsets and therefore they proposed a different approach that involves a lossless compression of the database.

1.2.2 Subtrajectory Clustering

The second work is "Subtrajectory Clustering: Models and Algorithms" [2] that, even if it regards a different context, proposes a clustering method that can be well adapted to our problem.
The goal of this work is to find a set of trajectories that well represent a set containing some input trajectories. As it can be noted, our goal is very similar to what we just described and so we tried to adapt their clustering model to our problem.

In this thesis, we will firstly describe in detail the works we just presented, then we will present our method and finally we will comment the results we obtained on synthetic datasets.

2 Related Works

In this chapter, we present two works that are related with our goal in terms of procedure adopted and context.
The first section will concern the work "Item Sets that Compress" [1], whose main problem is somewhat similar to ours and so it describes a possible approach we could follow.
In the second section, instead, the work "Subtrajectory Clustering: Models and Algorithms" [2] is presented. This paper presents a clustering model to solve the subtrajectory clustering problem that we will adopt to achieve our goal.

2.1 "Item Sets That Compress"

"Item Sets that Compress" describes a procedure to code an entire database of transactions without any loss of informations in order to improve the frequent itemset mining.

2.1.1 Introduction

Frequent Item Set Mining

Frequent item set mining is one of the best known and most popular data mining methods and can be described as follows.
There are two main sets:

- a set of items $\mathcal{I}$

- a database db of transactions over $\mathcal{I}$

A transaction t is a set of items that belongs to the set $\mathcal{P}(\mathcal{I})$ of all the possible permutations of $\mathcal{I}$ and also to the database db.
We have that a certain item set $I \subset \mathcal{I}$ occurs in a transaction t if and only if $I \subseteq t$.
The last term to define is the *support* of I in db, that is the number of transactions of the database in which I occurs and is denoted by $supp_{db}(I)$.
At this point, we can state that the problem of frequent item set mining is to determine all item sets I such that the support of I is greater than a given threshold $min\text{-}sup$, i.e. $supp_{db}(I) \geq min\text{-}sup$.

MDL Principle

The major obstacle of this problem consists in the explosion of the number of results. In fact, the threshold $min\text{-}sup$ plays an important role: if its value is high, the resulting sets are well-known, while if its value is low, the number of frequent item sets is too large.
In order to face this latter problem, the paper "Item Sets That Compress" proposes a completely different approach based on the Minimum Description Length Principle (MDL). In this case, the interest of an item set no longer depends on a certain threshold. What really matters is if an item set yields to a good lossless compression of the whole database.
To determine if a set yields to a good compression, the paper uses the MDL principle that balances the size of the compressed database and the size of the code table needed to code it. Its definition is the following:

Definition 1 *Given a set of models $\mathcal{H}$, the **best model** $H \in \mathcal{H}$ is the one that minimizes the quantity*

$$L(H) + L(D|H) \tag{2.1}$$

where $L(H)$ represents the length of the description of H and $L(D|H)$ is the length of the description of the data D when it is encoded with the model H.

The set of models $\mathcal{H}$, in this case, is represented by sets of item sets that can describe the database and coding tables that can code them.

2.1.2 Minimal Coding Set Problem

In this section we will present the problem that has to be solved.
First of all we will define the properties of the item sets coding, then the models of the problem are determined and at the end the actual definition of the problem is presented.

Item Sets Coding

First of all let us define the cover of a transaction as the set of item sets C such that $t = \bigcup_{c_i \in C(t)} c_i$.
In order to code properly the database, the coding set of item sets must respect two simple conditions: it should be able to describe every transaction of the database and the item sets that cover a certain transaction should be mutually disjoint.
Below is the formal definition of the two conditions, in which C is a set of item sets and $C(t) \subseteq C$ is the set of item sets that cover a certain transaction t.

1. $t = \bigcup_{c_i \in C(t)} c_i$
2. $\forall c_i, c_j \in C(t) : c_i \neq c_j \rightarrow c_i \cap c_j = \emptyset$

At this point, to know which item set has to be used to cover a given transaction $t \in db$, we need a way to assign a subset $C(t) \subseteq C$ to a transaction.
This is done by a coding scheme CS which is defined by a pair (C, S) where C is an item set cover of the entire database db and S is a function $S : db \rightarrow \mathcal{P}(C)$ such that $S(t)$ covers t.
In order to use this coding scheme to actually encode the database, we have to assign a code to each element c_i of the item set cover C.
Since we are interested only in the size of the compressed database, what we need is to know only the length of the coding and not also the actual code. To do this we can exploit a nice correspondence between the codes length and the probability distribution P induced by a coding scheme (C, S).

Definition 2 *Let P be a probability distribution on C induced by a coding scheme (C, S), there exists a (unique) code on C such that the length of the code for an element $c \in C$, is given by*

$$L(c) = -log(P(c))$$

where

$$P(c) = \frac{freq(c)}{\sum_{d \in C} freq(d)} \quad and \quad freq(c) = |\{t \in db | c \in S(t)\}|$$

A nice property is that this code gives the smallest expected code size for data sets drawn according to P.
Now that we know the code length $L(c)$ for each item set c in the item set cover C we can define also the code length $L(t)$ for each transaction $t \in db$, that is

$$L(t) = \sum_{c \in S(t)} L(c)$$

Finally, we have that for a given coding scheme (C, S), the size of the coded database is given by

$$L_{(C,S)}(db) = -\sum_{c \in C} freq(c) \log(P(c))$$

Code Table

In the previous section, we defined how to compute the second term of the expression [2.1]. Now we need to understand how to compute the first one and to do it we have to define what actually is a coding table.
A coding/decoding table is a kind of physical representation of the coding scheme because it defines the item set cover and the assignment of each item set to a list of transactions.
Formally, a coding table is a table with three columns: Item Set, Code and Tuple List where this last column is composed by the list of the transactions in which the item set is used for encoding.
The code table should respect the following conditions:

- The columns Item Set and Tuple List form a coding scheme.
- The length of the code in the second column should correspond with the frequency of the itemset in the coding scheme.

The coding table is therefore represented as in Table 2.1.

Item Set	Code	Tuple List
c_1	$code_1$	$(t \in db \mid c_1 \in S(t))$
c_2	$code_2$	$(t \in db \mid c_2 \in S(t))$
...	...	(...)
c_i	$code_i$	$(t \in db \mid c_i \in S(t))$
...	...	(...)

Table 2.1: coding/decoding table

The problem with this table is that the *Tuple List* is not the most efficient way to represent the mapping from item sets to tuples.
To overcome this issue, we simplify our problem and we proceed as follows.
First of all, the main changes are essentially two:

- we take a fixed algorithm to code the database.
- we assume an order on a code table that now has only two columns, one for the item sets and one for the codes.

To understand the coding procedure we also need the notion of a *coding set*: an ordered set of item sets that contains all the singleton item sets I for $I \in \mathcal{I}$. The order of this set is called *coding order*.
The fixed algorithm, called **Cover** (see Algorithm 1) is very straightforward. If we want to obtain the cover (and consequently the code) of a certain transaction $t \in db$ we look into the table following the coding order and we take the first item set c_i such that $c_i \subseteq t$. At this point the procedure continues recursively on $t \setminus c_i$ until the remainder is empty.
We can formalise the situation with the following theorem which states that we can extended our simplified model to a coding/decoding table.

Let C be a coding set and db a database, then

1. *C and **Cover** induce a unique code table.*
2. *This code table can be extended to a coding/decoding table.*

At this point we know what our models really are: the code tables (Table 2.2) induced by coding sets. In fact, we can code and decode all the transactions of the database using simply the code table and the algorithm **Cover**.

Item Set	Code
c_1	$code_1$
c_2	$code_2$
...	...
c_i	$code_i$
...	...

Table 2.2: code table

Our goal however is to compute the first term $L(H)$ of the expression [2.1], that is, the length of the description of our model. To do that we simply have to add the length of the descriptions of the two columns of the code table. This is easy for the second column given that we already know the length of the code of a certain item set ($-\log P(c)$). For the first column instead what we do is use another encoding.
Although it may seem strange to use an additional code, this is needed to know the exact size of the code table and it has a different meaning respect to the code on the second column. The code on the second column in fact is the code used to describe the database while the code on the first one describes the item set itself.
This new encoding, termed *standard encoding*, can only be done in terms of the singleton item sets and its definition is the following.

Definition 3 *The standard encoding of an item set c_i for a given database db over $\mathcal{I}$ is the encoding induced by the coding set $\{I\}_{I \in \mathcal{I}}$.*

Note that the table that would perform the mapping from the standard encoding to the names of the individual items is not considered in the following analysis since it is the same for all the code table and so it does not have any effect on the choice of the best model.

Now that we know the length of the elements contained in the code table, we can define what its actual size is.

Definition 4 *The size of the coding table CT_C induced by a coding set C for a database db is given by*

$$L(CT_C) = \sum_{c \in C : freq(c) \neq 0} (L_{st}(c) + L_C(c))$$

where $L_{st}(c)$ is the length of the standard encoding of c and $L_C(c)$ is the length of the encoding induced by the coding set C.

Algorithm 1 Cover(C, t)

Require: a coding set C and a transaction t
Ensure: the item set cover for transaction t
  $S \leftarrow$ smallest element c of C in coding order such that $c \subseteq t$
  if $t \setminus S = \emptyset$ **then**
    $Res \leftarrow S$
  else
    $Res \leftarrow S \cup Cover(C, t \setminus S)$
  end if
  return Res

Problem Definition

In the previous sections we defined how to compute the terms of the expression [2.1] that defines which is the model that would yield to the best compression.
Using the previous results can in fact define the total size of the encoded version of the database db

$$L_C(db) = L_{(C,S_C)}(db) + L(CT_C)$$

and consequently we can give the formal definition of the **Minimal Coding Set Problem**: for a database db over a set of items $\mathcal{I}$, find a coding set C for which $L_C(db)$ is minimal.
There is also a simplification of this problem, called **Minimal Coding Subset Problem**: for a database db over a set of items $\mathcal{I}$ and J a proto coding set (a set that must contain the singleton item sets), find a coding set $C(J) \subseteq J$, for which $L_{C(J)}(db)$ is minimal.
In this case the goal is to find a subset of J for which the size of the encoded version of the database is minimal, where J is a set of item sets that is given to us and must contain all the singleton item sets.

2.1.3 Solutions

At the end of this section four heuristic algorithms for the **Minimal Coding Subset Problem** will be presented. These algorithms are based on the same greedy strategy and use two strategies for improvement: **Pruning** and **Denoise**.

The greedy strategy

The greedy strategy employed by the algorithms is very straightforward:

- At first we start with the code consisting of only the singleton item sets.
- Then we add one by one the other item sets keeping only the ones that produce a better compression.

The initial encoding of the singleton is simply the *standard encoding* that we already defined. The actual order of the coding set that induces this encoding does not matter since each possible order of the coding set would produce a code of the same length. This is because the singleton item sets are obviously disjoint.
At this point two important points need to be defined:

- the order in which the item sets are place in the code table.
- the order in which we pick the item set to be checked for becoming part of the code table.

The answer to the first point is that we sort first by size and then by support so that the first elements in the code table are the smaller item sets and in case of two item sets of same size the one with the highest support in the database comes first. This order, defined *standard order*, is just a heuristic but it is justified from the properties of the item sets and the way the **Cover** algorithm works.

Definition 5 *Let $C \in \mathcal{P}(\mathcal{I})$ be an ordered set of item sets. C is in **standard order** for db if and only if for any two $J_1, J_2 \in C$*

- $size(J_1) \leq size(J_2) \rightarrow J_2 \leq_C J_1$
- $size(J_1) = size(J_2) \wedge supp_{db}(J_1) \leq supp_{db}(J_2) \rightarrow J_2 \leq_C J_1$

The answer to the second point, called *cover order*, is more greedy: we pick the item set with the highest cover in the database.
Algorithm **Naive-Compression** (see Algorithm 2) describes the greedy strategy with the order choices that we just discussed.

Algorithm 2 Naive-Compression($\mathcal{I}$, J, db)

Require: the set of items $\mathcal{I}$, the code set J and the database db
Ensure: the best code set the algorithm has seen
  $CodeSet \leftarrow$ Standard($\mathcal{I}$, db)
  $J \leftarrow J \setminus \mathcal{I}$
  $CanItems \leftarrow$ Cover-Order(J, db)
  while $CanItems \neq \emptyset$ **do**
    $cand \leftarrow$ maximal element of $CanItems$
    $CanItems \leftarrow CanItems \setminus cand$
    $CanCodeSet \leftarrow CodeSet \oplus cand$
    if $L_{CanCodeSet}(db) < L_{CodeSet}(db)$ **then**
      $CodeSet \leftarrow CanCodeSet$
    end if
  end while
  return $CodeSet$

In the pseudocode:

- **Standard($\mathcal{I}$, db)** returns the code set in the *standard order*.
- **Cover-Order(J, db)** returns the version of J in the *cover order*.
- $\oplus$ means that $cand$ is added in the right position following the *standard order*.

Pruning

The first improvement that we present is the **Pruning** of the code set.
This strategy tries to alleviate some problems caused by the heuristic choice of following the *cover order* to check if an item set should be part of the code set.
In fact, it could happen that **Naive-Compression** does not even consider a code set that would lead to a better compression and for this reason we prune the code set in the following way.
Suppose that we just added a new item set to a code set because it has led to a

better compression. Now we remove one by one from this code set the elements whose cover has become smaller with the addition of the last item set and check if the resulting compression is better.
At this point we have to define again in which order we remove the elements: in line with the previous reasonments we take first the ones with the smallest cover.
In **Prune-on-the-fly** (see Algorithm 3) the pruning strategy is represented.

Algorithm 3 Prune-on-the-fly($CanCodeSet, CodeSet, db$)

Require: the code set with the new item set $CanCodeSet$, the previous code set without the last added item set $CodeSet$ and the database db
Ensure: the best code set the algorithm has seen
  $PruneSet \leftarrow \{J \in CodeSet \mid cover_{CanCodeSet}(J) < cover_{CodeSet}(J)\}$
  $PruneSet \leftarrow$ Standard($PruneSet, db$)
  while $PruneSet \neq \emptyset$ **do**
    $cand \leftarrow$ element of $PruneSet$ with minimal cover
    $PruneSet \leftarrow PruneSet \setminus cand$
    $PosCodeSet \leftarrow CodeSet \ominus cand$
    if $L_{PosCodeSet}(db) < L_{CanCodeSet}(db)$ **then**
      $CanCodeSet \leftarrow PosCodeSet$
    end if
  end while
  return $CanCodeSet$

Denoise

The second improvement tries to get to a better solution using another approach. This strategy is based on the following observation: if a transaction needs an "exotic" itemset in its cover, it is very well possible that this transaction is just *noise*.
The idea of the procedure is therefore remove such unusual transactions from the database in order to make the regularity pattern more visible since also the very infrequent item sets will be removed from the code set.
The first question to address is how we decide if a transaction is noise or not. In order to do that we rely on the *standard encoding* that we already used.
The principle is that it could happen that the code for some transactions becomes longer than its *standard code* (i.e. the code produced by the standard encoding) so it would be better to code these transactions using the standard encoding.
The noise is therefore defined as those transactions whose code is longer than

their standard code, i.e. $L_S(t) < L_{CS}(t)$, and they are removed by three simple algorithms:

- **Noise** [4] that determines the unusual transactions in the database
- **Denoise** [5] that removes such transactions from the database
- **Sanitize** [6] that removes the item sets with a zero cover in the denoised database

Algorithm 4 Noise($\mathcal{I}, CodeSet, db$)

```
Noise ← ∅
for t ∈ db do
  if L_S(t) < L_CS(t) then
    Noise ← Noise ∪ {t}
  end if
end for
return Noise
```

Algorithm 5 Denoise($\mathcal{I}, CodeSet, db$)

```
Noise ← Noise(I, CodeSet, db)
db ← db \ Noise
return db
```

Algorithm 6 Sanitize($\mathcal{I}, CodeSet, db$)

```
db ← Denoise(I, CodeSet, db)
for J ∈ CodeSet do
  if cover_db(J) = ∅ then
    CodeSet ← CodeSet ⊖ J
  end if
end for
return CodeSet
```

Once the algorithms are defined we can compute the size of the new compressed database: it is the sum of the size of the sanitized code set, the size of the denoised database and the size of the *Noise* set.

Therefore, let CS be a coding set for a database of transactions db over a set of items $\mathcal{I}$, we have that

$$LN_{CS}(db) = L_{CS}(\textbf{Denoise}) + L_{CS}(\textbf{Sanitize}) + L_{CS}(\textbf{Noise})$$

In the case of the last term, the size is simply:

$$L_{CS}(Noise) = \sum_{t \in Noise} L_{standard}(t)$$

Note that the definition of the generalized length LN_{CS} is an extension of the previous length definition. In fact, if there are not unusual transactions we have that $\textbf{Noise}(\mathcal{I}, CS, db) = \emptyset$ and thus $LN_{CS}(db) = L_{CS}(db)$ since the three components that form LN_{CS} are:

- $\textbf{Denoise}(\mathcal{I}, CS, db) = db$
- $\textbf{Sanitize}(\mathcal{I}, CS, db) = CS$
- $L_{CS}(\textbf{Noise}(\mathcal{I}, CS, db)) = 0$

Algorithms

Given the two improvement strategies just defined in the previous sections, we can extend the algorithm **Naive-Compression** creating three other algorithms:

- **Compress-and-Prune** [7]: **Naive-Compression** + **Prune-on-the-fly**
- **Compress-and-Sanitize** [8]: **Naive-Compression** + **Sanitize**
- **All-out-Compression** [9]: **Naive-Compression** + **Prune-on-the-fly** + **Sanitize**

Note that it is not guaranteed that these algorithms will perform better than the simple **Naive-Compression**.

Algorithm 7 Compress-and-Prune($\mathcal{I}$, J, db)

Require: the set of items $\mathcal{I}$, the code set J and the database db
Ensure: the final code set
  $CodeSet \leftarrow$ Standard($\mathcal{I}$, db)
  $J \leftarrow J \setminus \mathcal{I}$
  $CanItems \leftarrow$ Cover-Order(J, db)
  while $CanItems \neq \emptyset$ **do**
    $cand \leftarrow$ maximal element of $CanItems$
    $CanItems \leftarrow CanItems \setminus \{cand\}$
    $CanCodeSet \leftarrow CodeSet \oplus \{cand\}$
    if $L_{CanCodeSet}(db) < L_{CodeSet}(db)$ **then**
      $CanCodeSet \leftarrow$ Prune-on-the-fly($CanCodeSet$, $CodeSet$, db)
      $CodeSet \leftarrow CanCodeSet$
    end if
  end while
  return $CodeSet$

Algorithm 8 Compress-and-Sanitize($\mathcal{I}$, J, db)

Require: the set of items $\mathcal{I}$, the code set J and the database db
Ensure: the final code set
  $Result \leftarrow$ Naive-Compression($\mathcal{I}$, J, db)
  $Result \leftarrow$ Sanitize($\mathcal{I}$, $Result$, db)
  return $Result$

Algorithm 9 All-out-Compression($\mathcal{I}$, J, db)

Require: the set of items $\mathcal{I}$, the code set J and the database db
Ensure: the final code set
  $Result \leftarrow$ Compress-and-Prune($\mathcal{I}$, J, db)
  $Result \leftarrow$ Sanitize($\mathcal{I}$, $Result$, db)
  return $Result$

2.2 "Subtrajectory Clustering: Models and Algorithms"

"Subtrajectory Clustering: Models and Algorithms" proposes a clustering model and a specific algorithm to solve the subtrajectory clustering model. In the following sections we will present in detail the procedure adopted by this work.

2.2.1 Introduction

The purpose of the paper "Subtrajectory Clustering: Models and Algorithms" is to extract shared structures that encode much of the information contained in a large trajectory dataset, in which each trajectory is simply represented as a sequence of points.
The subtrajectory clustering problem therefore consists in partition the subtrajectories of the input trajectories into a small number of clusters. Each of these clusters is represented as a *pathlet*, i.e., a sequence of points that is not necessarily a subsequence of an input trajectory, and each pathlet has to be a good representation of the subtrajectories in the cluster.
The clustering model presented in the document aims at capturing the shared portions between the input trajectories by considering each trajectory as a concatenation of a small set of pathlets with possible gaps between one pathlet and another.
The utility of extracting the pathlets does not only consists in compressing the starting dataset. The pathlets provide semantic information that could be useful for trajectory analysis applications and they also reduce uncertainty in individual trajectories since they can reduce noise and fill missing points.

2.2.2 General Model

In order to properly define the clustering model that we are going to use, first we need to define some necessary terms:

- a trajectory T is a polygonal curve defined by a finite sequence of points in $\mathbb{R}^2$;

- $\mathcal{T} = T_1, ..., T_n$ is the trajectories dataset, i.e., a set of n trajectories;
- $\mathbb{P} = P_1, ..., P_b$ is a set of b candidate pathlets;
- $\mathbb{X} = \bigcup_{i=1}^{n} T_i$ is the set of all trajectory points with $m = |\mathbb{X}|$. Note that, for simplicity, we assume that the points in each trajectory are distinct;
- $T[p, q]$ is the subtrajectory of T that goes from the point p to the point q.

In the Introduction Section 2.2.1 we briefly described what the problem is. To proceed we also need to know what actually is the solution we are looking for, that is, a subtrajectory clustering.

Definition 6 *A subtrajectory clustering is a* ***pathlet dictionary****, that is, a (multi) subset $\mathcal{P} \subseteq \mathbb{P}$, along with an assingment of a subset $\mathcal{T}(P)$ of subtrajectories to each $P \in \mathcal{P}$ such that there is at most one subtrajectory $S \in \mathcal{T}(P)$ of each trajectory $T \in \mathcal{T}$. When a subtrajectory S belongs to $\mathcal{T}(P)$ we say that S is assigned to or covered by P.*

However, we are not looking for a general subtrajectory clustering, we want to find a really good one. In the clustering model, this translates in finding the subtrajectory clustering that minimizes an objective function consisting of three terms:

- a first term proportional to the number of pathlets $|\mathcal{P}|$.
- a second term that represents the fraction of points of each trajectory that are not covered by any pathlet, measuring in this way the size of the gaps left uncovered.
- a third term that represents how well the pathlets resemble the assigned subtrajectories.

The objective function is formalised in the following equation

$$\mu(\mathcal{T}, \mathcal{P}, d) = c_1|\mathcal{P}| + c_2 \sum_{T \in \mathcal{T}} \tau(T) + c_3 \sum_{P \in \mathcal{P}} \sum_{S \in \mathcal{T}(P)} d(S, P) \tag{2.2}$$

in which c_1, c_2 and c_3 are user-defined parameters that regulate the trade-off between some properties of the final clustering. The last term d, instead, is a distance function that we will describe in detail later on that in some cases can be equal to ∞ so as not to allow certain assignments.

At this point, we can define the two main problems that we will face in the following sections: **Pathlet-Cover** and **Subtrajectory Clustering**.

Definition 7 *Given a tuple $(\mathcal{T}, \mathbb{P}, d)$, the **pathlet-cover problem** consists in computing $\mathcal{P}^* \subseteq \mathbb{P}$ and permissible assignment of subtrajectories to pathlets, to minimize the objective function $\mu(\mathcal{T}, \mathcal{P}, d)$. We let $\pi(\mathcal{T}, \mathbb{P}, d) = \mu(\mathcal{T}, \mathcal{P}^*, d)$.*

Definition 8 *The **subtrajectory-clustering problem** is to solve the pathlet-cover problem but without having a set of candidate pathlets, that is, $\mathbb{P}$ is the infinite set of all point sequences.*

Both problems are difficult to solve and, in fact, by a simple reduction from the standard *Facility-Location* problem in Euclidean space, we have that Pathlet-Cover problem is *NP-Complete* while Subtrajectory Clustering problem is *NP-Hard*.

2.2.3 Pathlet-Cover

In this section we will describe an approximation algorithm for the pathlet-cover problem that relies on the reduction to the standard *Set-Cover* problem.
The first thing to define is the actual reduction from one problem to the other. To do that, we define a **set system**, that is, a pair $(X, \mathcal{S})$, where $X = e_1, ..., e_\ell$ is the ground set of elements and $\mathcal{S}$ is a family of subsets of X.
The optimization version of the *Set-Cover* problem, given a set system and a weight function $w : \mathcal{S} \to \mathbb{R}^+$ find a subset $C \subseteq \mathcal{S}$ such that the sum of all the weights of the sets in C is minimum and X is covered by C, i.e., $\bigcup_{C \in C} C = X$.
At this point we need to define what X and $\mathcal{S}$ actually are:

1. X is the set of all points present in $\mathcal{T}$.
2. $\mathcal{S}$ is a family of subsets that contains two kinds of subsets
   - for every $P \in \mathbb{P}$ and for any set of input subtrajectories $\mathcal{R}$ drawn from distinct trajectories in $\mathcal{T}$ such that $d(S, P) \neq \infty$ for all $S \in \mathcal{R}$, family $\mathcal{S}$ contains a set $S(P, \mathcal{R}) = \{p \in S | S \in \mathcal{R}\}$ with $w(S(P, \mathcal{R})) = c_1 + c_3 \sum_{S \in \mathcal{R}} d(S, P)$.
   - let $T^{(p)} \in \mathcal{T}$ be the trajectory containing the point p. Then for every $p \in X$, family $\mathcal{S}$ contains a singleton set $\{p\}$ with $w(\{p\}) = c_2/|T^{(p)}|$.

These two types of subsets have an important meaning.
The first type represents the trajectory points that are covered by the pathlets and, in the greedy algorithm that we are going to describe, choosing a set $S(P, \mathcal{R})$ will result in covering the trajectory points present in the subtrajectories $\mathcal{R}$ assigned to the pathlet P.
The second type, instead, represents the trajectory points that will remain uncovered and choosing a set $\{p\}$ will mean marking a trajectory point p as permanently uncovered.

Now that we have defined what the set system consists of, we can describe how the greedy algorithm looks like.
Until every element is covered, the algorithm simply picks the set that maximizes a certain quantity called **coverage-cost ratio**, that is, the fraction between the newly covered elements and the weight of the set.
In order to understand more in depth its functioning, let us suppose that the algorithm is at an intermediate iteration in which C is the family of sets chosen so far. To formalise the definition of the coverage-cost ratio we need two new terms:

- $\hat{X} \subseteq X$ is the set of points still not processed by C, that is, the set of points that do not belong to any set $S(P, \mathcal{R})$ or singleton chosen in C.
- $\hat{S} = S \cap \hat{X}$ is the set of unprocessed points in a substrajectory S.

At this point, we can define the **coverage-cost ratio** ρ of the two kinds of sets in $\mathcal{S}$:

- for a family of subtrajectories $\mathcal{R}$ and a pathlet P we have that

$$\rho(P, \mathcal{R}) = \frac{\sum_{S \in \mathcal{R}} |\hat{S}|}{c_1 + \sum_{S \in \mathcal{R}} c_3 d(S, P)} \tag{2.3}$$

  with

$$\mathcal{T}_P = \underset{\mathcal{R}: S(P,\mathcal{R}) \in \mathcal{S}}{\operatorname{argmax}} \rho(P, \mathcal{R}) \qquad P^* = \underset{P \in \mathbb{P}}{\operatorname{argmax}} \rho(P, \mathcal{T}_P)$$

- while for each unprocessed point $p \in \hat{X}$ we have

$$\rho(p) = \frac{|T^{(p)}|}{c_2} \tag{2.4}$$

with

$$p^* = \underset{p \in \hat{X}}{\operatorname{argmax}}\ \rho(p)\ .$$

In the next iteration, the algorithm will pick the set with the higher coverage-cost ratio between $S(P^*, \mathcal{T}_{P^*})$ and $\{p^*\}$ and add it to the set C. Then the set $\hat{X}$ of unprocessed points, the values $\rho(P, \mathcal{R})$ and the sets $\mathcal{T}_P$ will be updated.
This algorithm has an approximation ratio of $O(\log m)$ when run on the set system $(X, \mathcal{S})$ and the main challenge in implementing it consists in the computation of $\mathcal{T}_P$ for each pathlet P given that $\mathcal{S}$ contains an exponential number of sets $S(P, \mathcal{R})$.
For this reason, we now describe an efficent procedure to compute $\mathcal{T}_P$ that requires two new terms in order to be introduced:

- $S_P(T)$ is the set of subtrajectories S of T where $d(S, P) \neq \infty$;
- for a set $S(P, \mathcal{R})$, $x_{S,T} \in \{0, 1\}\ \forall\ S \in S_P(T), T \in \mathcal{T}$ is a set of variables that indicate if $S \in \mathcal{R}$.

With this terms, we can rewrite the previous coverage-cost ratio equation 2.3 and the problem of computing $\mathcal{T}_P$ for a fixed pathlet P is equivalent to solving the following optimization problem:

$$\max \frac{\sum_{T \in \mathcal{T}} \sum_{S \in S(T)} |\hat{S}| x_{S,T}}{c_1 + c_3 \sum_{T \in \mathcal{T}} \sum_{S \in S(T)} d(S, P) x_{S,T}} \tag{2.5}$$

s.t.

$$\sum_{S \in S(T)} x_{S,T} \leq 1 \quad \forall\ T \in \mathcal{T}$$
$$x_{S,T} \in \{0, 1\} \quad \forall\ S \in S(T), T \in \mathcal{T}$$

If there exist feasible values of the variables x that satisfy the equation

$$\sum_{T \in \mathcal{T}} \sum_{S \in S(T)} (|\hat{S}| - c_3 \gamma d(S, P)) x_{S,T} \geq c_1 \gamma \tag{2.6}$$

we have that the maximum objective value is $\geq \gamma$.
At this point, for a fixed value of γ, the goal is to maximize the left hand side and therefore for each T we should pick the subtrajectory $S \in S(T)$ that maximizes the quantity $|\hat{S}| - c_3 \gamma d(S, P)$ (provided that is > 0). To do that, we define the function $\mathcal{F}_{P,T} : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ for each pathlet P and trajectory T as

$$\mathcal{F}_{P,T}(\gamma) = \max\{0, \max_{S \in S_P(T)} f_{S,P}(\gamma)\}$$

where $f_{S,P}$ is a function defined as

$$f_{S,P}(\gamma) = |\hat{S}| - c_3 \gamma d(S, P)$$

Finally, we define

$$\zeta_P(\gamma) = \sum_{T \in \mathcal{T}} \mathcal{F}_{P,T}(\gamma)$$

that is a monotonically non-increasing, piecewise-linear and convex function with at most $\sum_{T \in \mathcal{T}} |S_P(T)| + m$ pieces.
The optimal point γ^* is therefore the intersection point between ζ_P and the line passing through the origin with slope c_1. With this result we then have $\mathcal{T}_P = \{T \in \mathcal{T} | \mathcal{F}_{P,T}(\gamma^*) > 0\}$.

2.2.4 Subtrajectory Clustering

In this section the approximation algorithm for the subtrajectory clustering problem will be described. The algorithm consists in an adaptation to the procedure used for the pathlet-cover problem.
During the explanation, we will assume the trajectories to be κ-packed, that is, the length of the intersections of a trajectory with any disk of radius r is at most κr.
The algorithm relies on two main ideas: the first one represents the adaptation to pathlet-cover problem while the second one is a further improvement of the algorithm performances.

- First of all we construct a small set $\mathbb{S}$ of candidate pathlets exploiting the fact that the discrete Fréchet distance is a metric. In this way we obtain an optimal pathlet cover of $\mathcal{T}$ with respect to $\mathbb{S}$ that has a cost close to the cost of an optimal subtrajectory clustering of $\mathcal{T}$.

- Then, exploiting again the properties of the Fréchet distance but also the fact that the trajectories are κ-packed, we compute an approximation of the Frechét distance.

Candidate Pathlets

The goal of this paragraph is to find a way to obtain a candidate set of pathlets in order to be able to use the procedure presented for solving the pathlet-cover problem. There are mainly two ways that lead to the following results:

- A candidate set $\mathbb{S}$ of $O(m^2)$ pathlets such that $\pi(\mathcal{T}, \mathbb{S}, fr) \leq 2\pi(\mathcal{T}, \mathbb{P}, fr)$.
- A candidate set $\mathbb{C}$ of $O(m)$ pathlets such that $\pi(\mathcal{T}, \mathbb{C}, fr) = O(\log m)\pi(\mathcal{T}, \mathbb{P}, fr)$.

In the definitions just presented, the term fr refers to the discrete Fréchet distance of which we now present the definition.
Given two point sequences A and B, a *correspondence* is a subset $C \subseteq A \times B$ such that for all $a \in A$ ($b' \in B$), there exists $b \in B$ ($a' \in A$) such that $(a, b) \in C$ ($(a', b') \in C$).
A correspondence between two point sequences $T_1 = \langle p_1, p_2, ..., p_k \rangle$ and $T_2 = \langle q_1, q_2, ..., q_l \rangle$ is monotone if for $(p_i, q_j), (p_{i'}, q_{j'}) \in C$ with $i \leq i'$ we have $j \leq j'$.
The discrete Fréchet distance between T_1 and T_2 is defined as

$$fr(T_1, T_2) = \min_{C \in \Xi} \max_{(p,q) \in C} ||p - q||$$

where Ξ is the set of all monotone correspondences between $langlep_1, p_2, ..., p_k\rangle$ and $\langle q_1, q_2, ..., q_l \rangle$.

At this point, we can explain how to obtain the results presented above.
First of all, suppose that $\mathcal{P} \in \mathbb{P}$ is an optimal pathlet dictionary of $\mathcal{T}$. For all $P \in \mathcal{P}$, we have that the cost of assigning the subtrajectories in $\mathcal{T}(P)$ to P is computed as

$$d(P, \mathcal{T}(P)) = c_3 \sum_{S \in \mathcal{T}(P)} fr(S, P)$$

Thanks to the triangle inequality property of the Fréchet distance, we can replace the pathlet P with a subtrajectory of $\mathcal{T}(P)$ while only doubling the cost. This is formalised in the following lemma.

Lemma 1 *There exists a subtrajectory $S' \in \mathcal{T}(P)$ such that $d(S', \mathcal{T}(P)) \leq 2d(P, \mathcal{T}(P))$.*

Therefore, the procedure that allows us to obtain the first of the two results reported above, consists in replacing all dictionary pathlets with input subtrajectories in any solution of the pathlet-cover instance $(\mathcal{T}, \mathbb{P}, fr)$. This change

will increase the total cost by a factor of at most 2.

However, to further restrict the candidate set, we need to use another procedure that requires the notion of *canonical pathlets*. Consider a trajectory $T \in \mathcal{T}$ and assume for simplicity that $|T|$ is a power of 2. Consider also a balanced binary tree over the interval $[1, |T|]$.
We have that each node of the tree correponds to a certain interval $[i, j]$. Moreover, its left and right children correspond respectively to the first half of the interval and to the second half, i.e., $[i, \lfloor (i + j + 1)/2 \rfloor]$ and $[\lfloor (i + j + 1)/2 \rfloor + 1, j]$.
From looking at the tree, we have that each node, i.e., each interval, induces a subtrajectory $T(i, j)$ of T.
The subtrajectories associated to the nodes of the binary tree are termed as *canonical pathlets* and are denoted by $\mathbb{C}(T)$.
At this point we can implement the same procedure as before and replacing a pathlet P by a set of $O(\log m)$ canonical pathlets while increasing the cost by at most a factor of $O(\log m)$.
If we perform this substitution for every pathlet in the dictionary we obtain the following result, that is the one reported in the second point above.

Definition 9 *Let $\mathbb{P}$ be the set of all point sequences in $\mathbb{R}^2$ and let $\mathbb{C}$ be the set of all canonical pathlets of $\mathcal{T}$. We have that*

$$\pi(\mathcal{T}, \mathbb{C}, fr) = O(\log m) \cdot \pi(\mathcal{T}, \mathbb{P}, fr)$$

Approximate distances

In this paragraph we introduce an algorithm to compute a distance function d, that just approximates the actual Fréchet distance, in order to improve the overall performances.
In fact, instead of computing the distance function directly, we compute an approximation d between a subset of pairs of canonical pathlets and subtrajectories (for the other pairs the distance is set to ∞ to mark them as not permissible) with the goal of doing that without decreasing the cluster quality by much.
This algorithm helps the one used to solve the pathlet-cover problem because it reduces the number of permissible assignments and of course the computation of the distance is much faster.
To present the actual procedure we need to introduce first the notion of *r-simplification*.

For any $r > 0$, an r-simplification of a trajectory T is a trajectory S consisting of a subsequence of points from T such that $fr(T, S) \leq r$.

To compute the *r-simplification* of a trajectory there is a very simple algorithm:

- We include the first point of T
- We iterate in order along all the points of T and we include each point that is at distance $\geq r$ from the previously included point.
- We include the last point of T, regardless of its distance from the last included point.

The trajectory obtained from this linear algorithm is denoted by $\overrightarrow{T_r}$. In the same way we can define $\overleftarrow{T_r}$, that is, the trajectory obtained from the procedure above by starting from the last point of the trajectory and proceeding in reverse. Now, by including each point that is present in either $\overrightarrow{T_r}$ or $\overleftarrow{T_r}$ following the original order of T, we obtain the *r-simplification* of T. The following results derive from the construction:

- Let p' and q' appear in T_r in order. We have $fr(T[p', q'], T_r[p', q']) \leq r$.
- Let p and q appear in T in order. There exist p' and $q' \in T_r$ such that $T[p, q]$ is a subsequence of $T[p', q']$ and $fr(T[p, q], T_r[p', q']) \leq r$.

At this point, we can construct a distance function d that gives a 4-approximation for some pathlet-subtrajectory pairs.
Let $\bar{r}$ be the maximum pairwise distance between trajectory points and be $\underline{r}$ the minimum. For each $T \in \mathcal{T}$ and $r \in \langle \bar{r}, \bar{r}/2, ..., \underline{r} \rangle$, we want to quickly decide which subtrajectories of T_r are within distance $O(r)$ of each pathlet P.
In order to be able to do that, the set of points contained in the *r-simplification* T_r are preprocessed into an approximate spherical range query data structure of size $O(|T_r|)$ in $O(|T_r|)$ time. This data structure is constructed so as to answer queries of the following form: let $\mathcal{B}(p, r) = \{x \in \mathbb{R}^2 | \ ||x - p|| \leq r\}$ denote a ball of radius r around p; given a point p, the data structure returns a subset of $\mathcal{B}(p, 2r) \cap T_r$ that includes all points in $\mathcal{B}(p, r) \cap T_r$ but no points outside $\mathcal{B}(p, 2r) \cap T_r$.
Each query requires constant time with an additional time proportional to the number of points returned by the data structure.

At this point, for each pathlet P in the candidate set $\mathbb{C}$, let p and q be respectively the start and end points of P and also define $Q^1_{TPr} = \mathcal{B}(p, 3r) \cap T_r$ and $Q^2_{TPr} = \mathcal{B}(q, 3r) \cap T_r$.
Now, compute Q^1_{TPr} and Q^2_{TPr} using the previous data structure and then, for every pair (p', q') such that $p' \in Q^1_{TPr}$ and $q' \in Q^2_{TPr}$, invoke a decision procedure that checks if $fr(T_r[p', q'], P) \leq 3r$ and if it is, we set $d(T[p', q'], P) = 4r$.
We do this procedure for all the triples (T, r, P) and we set the values of d that are not defined by the algorithm above as not permissible, i.e., equal to ∞.

3

Representative Itemsets

In this chapter we will first present the formal definition of our problem, that is, to find a set of itemsets that well represent an initial database of transactions. Then, we will explain how the problem can be solved by applying a procedure inspired by a work regarding the clustering of subtrajectories.

3.1 Introduction

The goal of the problem that we are going to describe is similar but not equal to the one of the paper "Item Sets that Compress" [1] presented in the previous chapter.
In fact, while in "Item Sets that Compress" the result contains, among the other things, a set of itemsets that compress without losses the whole database, in this case the result consists of a set of itemsets that represents the database but with losses in the informations.
Therefore, what we want to achieve is a set of itemsets that does not compress every transaction but gives a good representation of what the database contains.
To do that, we will apply a clustering method inspired by "Subtrajectory Clustering: Models and Algorithms" [2].
In the following section, we will present the general clustering model based on the following terms:

- a set of items $\mathcal{I}$;
- a database db over $\mathcal{I}$, that is, a set of transactions $\{t_1, t_2, ..., t_n\}$;

- a transaction t, that is, is a set of items taken from $\mathcal{I}$;
- an itemset $I \subset \mathcal{I}$, that is, a set of items that could occur in a transaction $t \in db$ if and only if $I \subseteq t$,
- a set $\mathbb{D}$ containing all the possible itemsets drawn from $\mathcal{I}$
- a set $\mathcal{D} \subset \mathbb{D}$ of representative itemsets $\mathcal{D} \subset \mathbb{D}$
- a set $\mathcal{T}$ containing all the possible subsets of the transactions in db
- a subset $\mathcal{T}(D) \subset \mathcal{T}$ that contains the subsets of the transactions associated to the representative itemset $D \in \mathcal{D}$. When an itemset T belongs to $\mathcal{T}(D)$ we have that T is covered by D.

3.2 Clustering Model

As most problem does, ours has an input and an output respectively represented by:

- the database of transactions db
- the set of representative itemsets $\mathcal{D}$ along with an assignment $\mathcal{T}(D)$ for each itemset $D \in \mathcal{D}$ such that there is at most one itemset $T \in \mathcal{T}(D)$ of each transaction $t \in db$.

We now present an objective function similar to the one presented previously (see Equation 2.2) but with some changes due to the different context. In order to obtain a good clustering, the model aims at minimizing the objective function that is, again, composed by three terms:

- the first term is proportional to the number of representative itemsets in order to regulate the cardinality of the set $\mathcal{D}$ and therefore not get a number of clusters that is too high or too low.
- the second term represents the fraction of items of each transaction that are not covered by any representative itemset so as to decide how much we can neglect of each transaction.

- the third term represents how much the representative itemsets are close to the subsets of the transactions that are assigned to them.

Therefore, the objective function is expressed as:

$$\mu(db, \mathcal{D}, d) = c_1|\mathcal{D}| + c_2 \sum_{t \in db} \tau(t) + c_3 \sum_{D \in \mathcal{D}} \sum_{T \in \mathcal{T}(D)} d(T, D) \tag{3.1}$$

where c_1, c_2 and c_3 are user-defined parameters that regulate the trade-off between the three terms just described, d is a distance function that we will discuss later on and $\tau(t)$ is the fraction of the transaction's t items that is not covered by any representative itemset.

3.3 Algorithm

At this point we can describe an algorithm that is strictly connected to the procedure associated with the Pathlet-Cover. In fact, as we previously did, we need to outline a reduction to the standard *Set-Cover* problem.
The main element of this reduction is, again, a **set system** $(X, \mathcal{S})$ which, however, this time is composed by

1. the set of all items present in the database db together with the index of the transaction to which they belong, that is, $X = \{(i, k) | i \in t_k\}$
2. and a family of subsets of X, termed $\mathcal{S}$, that contains two different kinds of sets
   - for every representative itemset $D \in \mathcal{D}$ and for any subset $\mathcal{R} \subset \mathcal{T}$ such that $d(T, D) \neq \infty$ for all $T \in \mathcal{R}$, family $\mathcal{S}$ contains a set $S(D, \mathcal{R}) = \{i \in T | T \in \mathcal{R}\}$ with associated weight $w(S(D, \mathcal{R})) = c_1 + c_3 \sum_{T \in \mathcal{R}} d(T, D)$.
   - for each transaction t and for each element i in t, family $\mathcal{S}$ contains a tuple (i, k) where $k \in [1, n]$ is the index of the transaction t. The weight of a tuple is $w((i, k)) = c_2/|t_k|$ where $|t_k|$ represents the number of items contained in the transaction that has index k.

In the definition just presented, we included also the definition of a weight for each set. This is because we are interested in the optimization version of

the *Set-Cover* problem that consists in: given a set system and a weight function $w : \mathcal{S} \to \mathbb{R}^+$ find a subset $\mathcal{C} \subseteq \mathcal{I}$ such that the sum of all the weights of the sets in $\mathcal{C}$ is minimum and X is covered by $\mathcal{C}$, i.e., $\bigcup_{C \in \mathcal{C}} C = X$.
As it was for the pathlet-cover algorithm, the two kinds of sets in $\mathcal{S}$ have a an important and practical meaning.
In fact, while the $S(\mathcal{R})$ sets represent the subsets of the transactions that are covered by a certain representative itemset, the tuples represent the items that will remain uncovered. Therefore, by choosing a set of the first type, the algorithm is actually deciding that the subsets that belong to $\mathcal{R}$ are covered by a representative itemset in $\mathcal{D}$. Instead, by choosing a tuple (i, k), the algorithm is marking that item i in that specific transaction t_k as permanently uncovered.

Now that we know what the set system represents, we can explain why it is the main element of the reduction to the *Set-Cover* problem. This is formalised by the following lemma and the consequent proof which derives from the one presented in the Subtrajectory Clustering paper [2].

Lemma 2 *There exists a bijection between set covers of $(X, \mathcal{S})$ and solutions to the representative-cover problem for $(db, \mathcal{D}, d)$ so that cost and weight remain equal across the bijection.*

Proof 1 *Let us consider a solution to the representative-cover problem, consisting of the set $\mathcal{D}$ and the assingments $\mathcal{T}(D)\ \forall D \in \mathcal{D}$. We are going to create a solution $\mathcal{C}$ to the Set-Cover instance $(X, \mathcal{S})$.*
For each uncovered item in each transaction, we add the singleton set $\{i\}$ to $\mathcal{C}$ and for each $D \in \mathcal{D}$, we add the set $S(D, \mathcal{T}(D))$ to $\mathcal{C}$. It can be easily verified that the total weight of $\mathcal{C}$ is equal to the cost of D.
In the same way, let us consider an optimal solution $\mathcal{C}$ to the Set-Cover instance. For each set in $\mathcal{C}$ of the form $S(D, \mathcal{R})$, we add the representative itemset D to $\mathcal{D}$ and assign all the itemsets in $\mathcal{R}$ to D. Every item i such that $\{i\} \in \mathcal{C}$ is left uncovered. Again, the cost of $\mathcal{D}$ is the same as the total weight of $\mathcal{C}$. □

At this point, we can describe how the greedy algorithm really works in detail (see Algorithm 10 for the pseudocode). The idea is the same as the one presented in Section 2.2.3: the algorithm iteratively picks the set that maximizes the **coverage-cost ratio** (formally defined later on) until every item is either marked as covered or uncovered.

Suppose that the algorithm is running and so it is in an intermediate iteration in which $C \subseteq \mathcal{S}$ is the family of sets chosen so far.
We define $\hat{X} \subseteq X$ as the set of items still not processed (neither marked as covered nor as uncovered, and so not present in any $S(D, \mathcal{R}) \in C$ or tuple) by C and $\hat{T} = T \cap \hat{X}$ as the set of unprocessed items that belong to a subset T of a transaction.
With these new terms we can provide the formal definition of the **coverage-cost ratio**, denoted by ρ, for both kinds of sets contained in $\mathcal{S}$:

- for a family of subtrajectories $\mathcal{R}$ and a representative itemset D we have that
$$\rho(D, \mathcal{R}) = \frac{\sum_{T \in \mathcal{R}} |\hat{T}|}{c_1 + \sum_{T \in \mathcal{R}} c_3 d(T, D)} \tag{3.2}$$
with
$$\mathcal{T}_D = \underset{\mathcal{R}: S(D,\mathcal{R}) \in \mathcal{S}}{\operatorname{argmax}} \rho(D, \mathcal{R}) \qquad D^* = \underset{D \in \mathbb{D}}{\operatorname{argmax}} \rho(D, \mathcal{T}_D)$$

- while for each unprocessed item $i \in \hat{X}$ that belongs to a certain transaction t_k we have
$$\rho(i, k) = \frac{|t_k|}{c_2} \tag{3.3}$$
with
$$i_k = \underset{i \in \hat{X}}{\operatorname{argmax}} \rho(i, k) \qquad k^* = \underset{k \in [1,n]}{\operatorname{argmax}} \rho(i_k, k)$$

In the next iteration, the set with the higher coverage-cost ratio between $S(D^*, \mathcal{T}_{D^*})$ and (i_{k^*}, k^*) will be added to the set C and finally, to conclude the iteration, the set $\hat{X}$ of unprocessed points, the values $\rho(D, \mathcal{R})$ and the sets $\mathcal{T}_D$ will be updated.
This algorithm, like the one in Section 2.2.3, has an approximation ratio of $O(\log m)$ when run on the set system $(X, \mathcal{S})$. The main challenge in implementing it consists again in the computation of $\mathcal{T}_D$ for each representative itemset D since the number of sets $S(D, \mathcal{R}) \in \mathcal{S}$ is exponential.

Algorithm 10 Greedy-Algorithm(db)

Require: a database db of transactions
Ensure: a set containing the representative itemsets
  $candidates \leftarrow$ compute_candidates(db)
  $subtransactions \leftarrow$ compute_subtransactions(db)
  $uncovered\_tuples \leftarrow$ uncovered_initialization(db)
  while $uncovered\_tuples$.size() > 0 **do**
    for $cand$ in $candidates$ **do**
      if is_ratio_changed($cand$) **then**
        $set\_T_D \leftarrow$ compute_T_D($cand, subtransactions$)
        $cand.ratio \leftarrow$ compute_coveragecost($cand, set\_T_D$)
        $cand.set\_T_D \leftarrow set\_T_D$
      end if
      if $cand.ratio > best\_cand.ratio$ **then**
        $best\_cand \leftarrow cand$
      end if
    end for
    if $best\_cand.ratio > uncovered\_tuples$.top().$ratio$ **then**
      cover_items($best\_cand$)
      $representatives$.add($best\_cand$)
    else
      $cover\_item$($uncovered\_tuples$.top())
      $uncovered\_tuples$.pop()
    end if
  end while
  return $representatives$

3.4 Computation of $\mathcal{T}_D$

In order to compute $\mathcal{T}_D$ we now describe the adaptation of the procedure previously presented for the computation of $\mathcal{T}_P$.
This requires two new terms:

- $T(t)$ is the set of subsets T of the transaction t;
- for a set $S(D, \mathcal{R})$, $x_{T,t} \in \{0,1\}\ \forall T \in T(t),\ t \in db$ is a set of variables that indicate if $T \in \mathcal{R}$.

Using the new terms, the coverage-cost ratio equation 3.2 can be rewritten and the problem of computing $\mathcal{T}_D$ for a certain representative itemset D becomes an optimization problem defined as follows:

$$\max \frac{\sum_{t \in db} \sum_{T \in T(t)} |\hat{T}|\, x_{T,t}}{c_1 + c_3 \sum_{t \in db} \sum_{T \in T(t)} d(T, D)\, x_{T,t}} \tag{3.4}$$

s.t.

$$\sum_{T \in T(t)} x_{T,t} \leq 1 \quad \forall\, t \in db$$
$$x_{T,t} \in \{0,1\} \quad \forall\, T \in T(t), t \in db$$

If there exist feasible values of the variables x that satisfy the equation

$$\sum_{t \in db} \sum_{T \in T(t)} (|\hat{T}| - c_3 \gamma d(T, D)) x_{T,t} \geq c_1 \gamma \tag{3.5}$$

we have that the maximum objective value is $\geq \gamma$.
From now on, the procedure is the same as the previous one but with the right adaptations: for a fixed value of γ, the goal is to maximize the left hand side of 3.5 and therefore for each transaction t we should pick the subset $T \in T(t)$ that maximizes the quantity $|\hat{T}| - c_3 \gamma d(T, D)$ (provided that is > 0).
In order to do that, we define a function $\mathcal{F}_{D,t} : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ for each representative itemset D and transaction t

$$\mathcal{F}_{D,t}(\gamma) = \max\{0, \max_{T \in T(t)} f_{T,D}(\gamma)\}$$

where $f_{T,D}$ in this case is a function defined as

$$f_{T,D}(\gamma) = |\hat{T}| - c_3 \gamma d(T, D).$$

Finally, we define another function

$$\zeta_D(\gamma) = \sum_{t \in db} \mathcal{F}_{D,t}(\gamma)$$

that is monotonically non-increasing, piecewise-linear and convex with at most $\sum_{t \in db} |T(t)| + m$ pieces.
With these elements, we have that the optimal point γ^* is at the intersection point between the function ζ_D and the line that has slope equal to c_1 and passes through the origin. This result leads directly to what was our goal since we have that

$$\mathcal{T}_D = \{t \in db \mid \mathcal{F}_{D,t}(\gamma^*) > 0\}.$$

3.5 Distances Definition

In the presentation of the clustering model and the greedy algorithm, we used a generic function d to evaluate the distance between two itemsets.
This function can be represented by various distances, among which we have:

- **Jaccard distance:** it is the complement of the Jaccard similarity.
  The Jaccard similarity between two itemsets I_1 and I_2 is computed as follows
  $$J(I_1, I_2) = \frac{|I_1 \cap I_2|}{|I_1 \cup I_2|}$$
  and so the Jaccard distance between the same two sets is
  $$1 - J(I_1, I_2)$$

- **Sørensen–Dice distance:** it is the complement of the Sørensen–Dice coefficient.
  The Sørensen–Dice coefficient between two itemsets I_1 and I_2 is computed as
  $$S(I_1, I_2) = \frac{2|I_1 \cap I_2|}{|I_1| + |I_2|}$$
  therefore the Sørensen–Dice distance between the same sets is equal to
  $$1 - S(I_1, I_2)$$

The main difference between these two functions is that the Jaccard distance is a metric. The Sørensen–Dice distance in fact, does not satisfy the triangle

inequality and this represents its main drawback.
Of course, these are only two very well known examples but other types of distances that measure how different two sets are can also be used.

3.6 Candidate Representative Itemsets

In the presentation of the problem, we defined the set $\mathbb{D}$ as the set containing all the possible itemsets drawn from $\mathcal{I}$.
However, if the set $\mathcal{I}$ is very large, the set $\mathbb{D}$ will contain too many candidates and the algorithm, since it is forced to try them all, may take a long time to execute. In order to avoid this problem, a sort of preprocessing could be applied to obtain a set of candidates that is not too large.
In this section, we are going to present an idea whose aim is to create a set of candidates that will contain fewer elements than $\mathbb{D}$ while remaining heterogeneous enough to obtain a good final result.

3.6.1 Candidates Generation

The procedure consists of two phases:

- Frequent itemsets mining
- Union of sets with high overlap coefficient

In the first phase, by using for example the Apriori Algorithm introduced in "Fast Algorithms for Mining Association Rules in Large Databases" [3] we want to obtain a set containing the most frequent itemsets in the database.
Apriori is a simple iterative procedure in which at each iteration two main steps are executed: *join* and *prune*. The result obtained by this algorithm is a set containing all the itemsets I such that the support of I is greater than a certain threshold $min\text{-}sup$, that is, $supp_{db}(I) \geq min\text{-}sup$.

In the second phase, we want to somehow add some diversification to the set just obtained by Apriori. One possible way to do so consists in adding to the set also the union of the itemsets that have a high **overlap coefficient**.

The overlap coefficient, also termed as Szymkiewicz-Simpson coefficient, between two itemsets I_1 and I_2 is defined below.

$$overlap(I_1, I_2) = \frac{|I_1 \cap I_2|}{\min(|I_1|, |I_2|)}$$

The final set of candidates will therefore contain the most frequent itemsets obtained by the first phase and the itemsets resulting from the second phase.

In this procedure, a crucial point is represented by the choice of the right value of the $min\text{-}sup$ threshold and the overlap coefficient.
Starting from $min\text{-}sup$ we can consider two different cases:

- If the threshold value is set too high, we have that the number of frequent itemsets may not be large enough to obtain a heterogeneous set of candidates;
- If instead the threshold is set too low, we have that the number of frequent itemsets may be so large as to not represent an improvement compared to the set $\mathbb{D}$.

For the overlap coefficient we can distinguish two other similar cases:

- If the coefficient is set too high, we may not obtain enough new itemsets from the second phase;
- If instead the coefficient is set too low, we may have to merge too many itemsets.

4 Implementation Details

In this chapter we will present how our code is organized and the data structures we used. All the code can be found at this https://github.com/giacomo6699/Representative-Itemsets-Clustering.git.

4.1 Code Structure

The programming language we adopted is c++ and the structure of the code is simple. The code is composed of four main files: **main.cpp**, **apriori.cpp**, **itemset_utilities.cpp** and **utilities.cpp** with the related header files.
Each file contains the code that concerns a different kind of tasks:

- **main.cpp** contains the functions regarding the main procedure, that is the Greedy-Algorithm (see Algorithm 10). This file contains functions like the one to compute the set $\mathcal{T}_D$ or the one to compute the starting set of candidates.
- **apriori.cpp** contains all the functions regarding the Apriori procedure to compute the frequent itemsets of the database db.
- **itemset_utilities.cpp** contains utilities functions that are more related to operations on sets and the definition of some behaviours of custom objects.
- **utilities.cpp** contains more general functions that mainly manage the parsing of the input and the creation of a synthetic database.

4.2 Data Structures

This section will regard two different types of data structures: **Priority Queue** and **Map**. We will present their performances and how they are used in the code.

4.2.1 Priority Queue

A **priority queue** is a data structure in which the elements must respect two conditions:

- the first element of the queue is always the greatest (smallest).
- the elements are in non-increasing (non-decreasing) order.

The methods of this data structure we used are:

- **push()** that adds an element to the queue
- **top()** that returns the greatest element
- **pop()** that removes the greatest element from the queue

and their time complexities are represented in the following table:

Methods	Complexity
push()	$O(\log N)$
top()	$O(1)$
pop()	$O(\log N)$

Table 4.1: priority queue methods

In the code, we used the priority queue for $uncovered\_tuples$ in order to collect the items of the database that are not covered yet. In this way, the element returned by **top()** is always the one with the highest coverage-cost ratio.

4.2.2 Map

A **Map** is a data structure that contains key-value pairs whose keys must all be distinct.
The methods we mainly used are:

- **insert(key, value)** that adds a new pair to the map
- **operator[]** that given a key returns the related value
- **erase(key)** that given a key removes the related pair from the map

and their time complexities are represented in the following table:

Methods	Complexity
insert(key, value)	$O(\log N)$
operator[]	$O(\log N)$
erase(key)	$O(\log N)$

Table 4.2: map methods

In the code we mainly used the map for:

- $subtransactions$ in which the key is a certain subtransaction and the associated value is itself a pair. This pair contains the number of tuples that the subtransactions covers (that is, $\hat{T}$) and the indexes of the transactions in which the subtransaction is present.
- $candidates\_T_D$ in which the key is the id of a representative candidate and the value is itself a pair. This pair contains the set $\mathcal{T}_D$ associated to the candidate and its coverage-cost ratio.
- $candidates\_indexes$ in which the key is a representative candidate and the associated value is itself a pair. This pair contains the id of the candidate and the indexes of the transactions in which at least one subtransaction $\in \mathcal{T}_D$ is contained.

4.3 Synthetic Data

In this last section we will explain how the synthetic data for the tests have been generated.
The steps for the generation of the database are essentially two:

- **clusters generation:** for every number between 0 and the max transaction length we add that number to the cluster with probability 0.5 and all the clusters must be at a certain distance between each other.
- **database generation:** for every transaction that we have to generate we casually pick a cluster and at this point we add noise to it. To add noise we iter through the numbers between 0 and max transaction length and we check if they are in the transaction. If they are in the transaction we remove them with a probability p while if they are not present we add them with probability p.

5 Results

In this chapter the results obtained by our algorithm are presented and commented. The graphs we will show are the results of different test on different measures.
The tests were executed on one thread of a linux server with the following characteristics:

- 1 TB of RAM
- Processor Model Name: Intel(R) Xeon(R) Gold 5220 CPU @ 2.20GHz
- Threads per Core: 1
- Cores per Socket: 18
- Sockets: 4

5.1 Introduction

The clustering of representative itemsets has a lot of parameters and variables to take into account. The results obtained and the time spent to find them strongly depend on different aspects of the starting database.
The list of the parameters regarding the functioning of our procedure are:

- **c1** that is the parameter that regulates the number of the representatives the algorithm finds. The higher it is, the fewer representatives there will be.
- **c2** that is the parameter that regulates the number of uncovered tuples in the database. The higher it is, the fewer uncovered tuples there will be.
- **c3** that is the parameter that regulates the distance between the representatives and the related subtransactions. The higher it is, the closer they have to be.
- **min_supp** that is the minimum support used to find the frequent itemsets (see Section 2.1.1).
- **overlap_threshold** that is the minimum value of the *overlap coefficient* for which we add the union to the candidates set (see Section 3.6.1).

while the parameters that concern the database complexity are:

- the number of transactions
- the number of clusters
- the maximum length of a transaction
- the noise percentage

For all our results we set the **min_supp** to 0.004 and **overlap_threshold** to 1.
To make these choices we performed numerous tests with $min\_supp = 0.002$, $min\_supp = 0.004$ and $min\_supp = 0.0075$ and we understood that the best choice was 0.004 and 1 because we noticed that for our case the apriori method produced a good set of candidates and there was no need to improve it by adding the union of the candidates with high overlap coefficient.
For **c1**, **c2** and **c3** we tried mainly two different configurations:

- we first started with $c1 = 3$, $c2 = 2$ and $c3 = 2$ in order to give more importance to having a small number of final representatives,
- we then switched to $c1 = 1$, $c2 = 1$ and $c3 = 1$ because the results of the previous tests were not satisfactory and the representatives were too few therefore we wanted a more equilibrated configuration.

As performances measures we chose 4 different values:

- **Precision**: is the ratio between the true positives TP and the sum between true positives and false positives $TP + FP$, that is, $\frac{TP}{TP+FP}$;
- **Recall**: is the ratio between the true positives TP and the sum between true positives and false negatives $TP + FN$, that is, $\frac{TP}{TP+FN}$;
- **Objective Function Ratio**: is the ratio between the objective function value obtained by the representatives computed by the algorithm and the objective function value obtained by the real starting clusters;
- **Time**: is the execution time of the algorithm

Finally, given that for each possible combination of parameters we executed the experiment three times of the algorithm, we presented in the graphs the *mean* and the *standard deviation* of the three results.

5.2 3-2-2 Configuration

In this section, the results regarding the $c1 = 3$, $c2 = 2$ and $c3 = 2$ configuration are presented.
This configuration gives more importance to minimizing the number of clusters rather than minimizing distances or uncovered items. In the following sections we will comment the results we obtained with a database of 100, 500 and 1000 transactions and a maximum transaction length of 10 and 15.
On the y-axis we will have the values associated with the performance measure while on the x-axis we will have the number of clusters that are 5, 10, 15 and 20.

5.2.1 Precision

In this section, the graphs presented show different values of mean and standard deviation of precision for each value of noise and number of starting clusters.

100 Transactions

In Figure 5.1 and 5.2 are presented the results obtained with 100 transactions and a maximum transaction length respectively of 10 and 15.

As expected the precision decreases as the noise and the number of clusters get higher. This is because the complexity of the database increases and the algorithm does not manage to find the exact starting clusters but instead returns clusters that are just similar to them.
Comparing the two graphs, we can also notice that, when the maximum transaction length is 15, for a high number of clusters the algorithm performs worse while for a small number of clusters the algorithm seems to perform slightly better.
The first consideration is probably caused by the fact that with a high number of clusters and a maximum transaction length of 15 the complexity of the database is too high compared to the number of transactions which is rather small.
The second one instead could be explained by the fact that with a greater maximum transaction length and a lower number of clusters, the algorithm had the opportunity to create more distant clusters which are therefore more recognisable.

500 and 1000 Transactions

For the graphs with a greater number of transactions, the considerations made above are still valid because we can notice the same general behaviour of the previous graphs.
For these graphs, however, we see that the precision is a bit higher when there is not noise and this is caused by the fact that now the size of the database is bigger.

In general, we can see that the algorithm does not reach good results in terms of precision.
Even if these graphs may make it appear that the algorithm does not work at all, we have to understand that the precision does not capture all the informations of the results and this is the reason why we chose to show different measures.
Moreover, the precision, just like the recall, base their value on the exact correspondence of the starting clusters and the final representatives. For our case this could be a problem because the objects are sets of numbers and two sets are equal if and only if they contain the exact same elements. Therefore a cluster and a representative are different even if they share all their numbers but one and so the representative will be not counted in the precision formula.

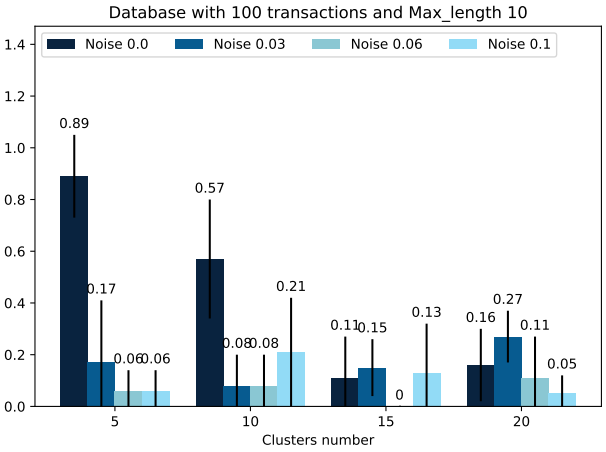

Figure 5.1: Precision plot of 100 transactions of max length 10

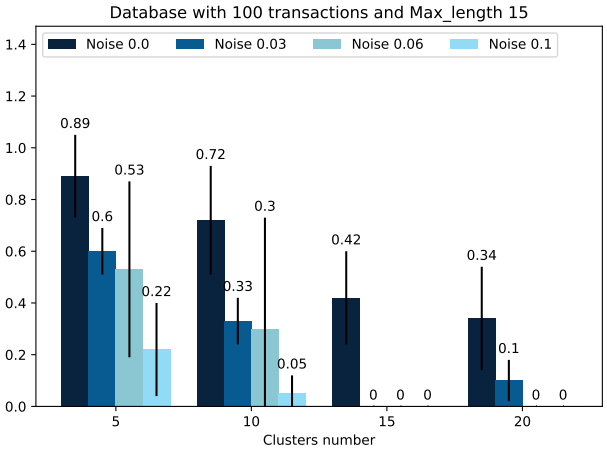

Figure 5.2: Precision plot of 100 transactions of max length 15

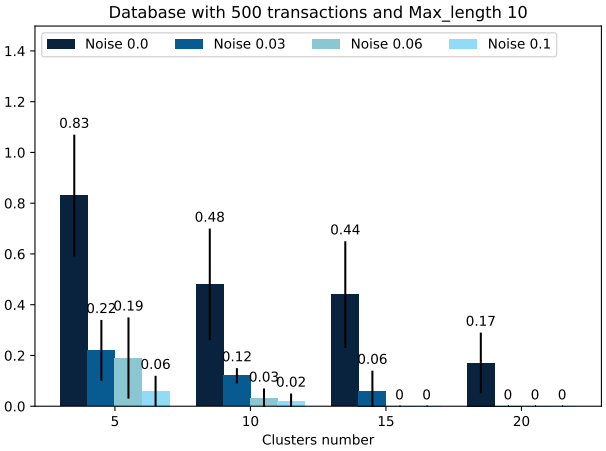

Figure 5.3: Precision plot of 500 transactions of max length 10

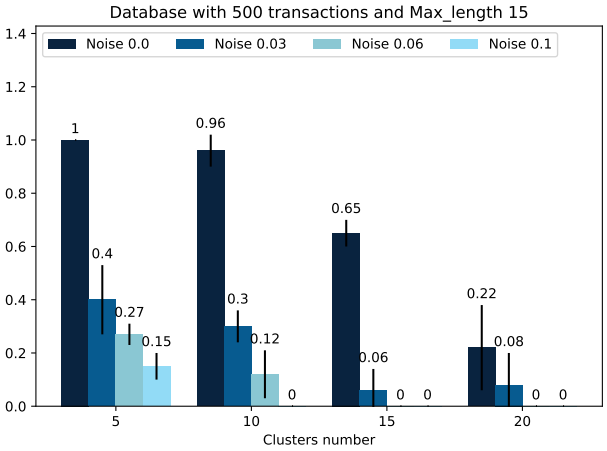

Figure 5.4: Precision plot of 500 transactions of max length 15

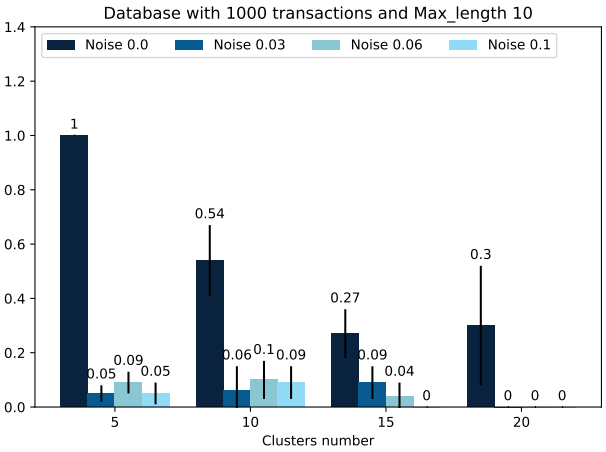

Figure 5.5: Precision plot of 1000 transactions of max length 10

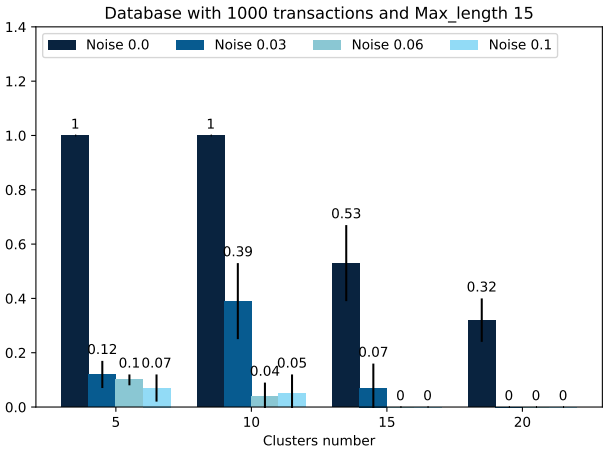

Figure 5.6: Precision plot of 1000 transactions of max length 15

5.2.2 Recall

In this section, the graphs presented show different values of mean and standard deviation of recall for each value of noise and number of starting clusters.

100, 500 and 1000 Transactions

The figures in the following page present the recall results for different database sizes and maximum transaction length.
As we saw in the previous section, also in this case the recall decreases as the noise and the number of clusters get higher.
The considerations made for precision can be valid also for the recall, because as we can see, the graphs concerning these two measures are pretty similar in terms of general behaviour.
One aspect that can be underlined is that the algorithm seems to reach good values of recall when the number of clusters is small. This is caused by the fact that the algorithm most of the times manages to find the starting clusters if they are not a lot even with different values of noise. As we can see, in fact, the results obtained on the graphs on the right are pretty good if we consider just the values corresponding to 5 clusters while they are not good at all as the number of clusters increases.
This is more visible on the graphs on the right mainly because of what we said on the previous section: with a higher maximum transaction length, the starting clusters are further apart and the algorithm performs better.

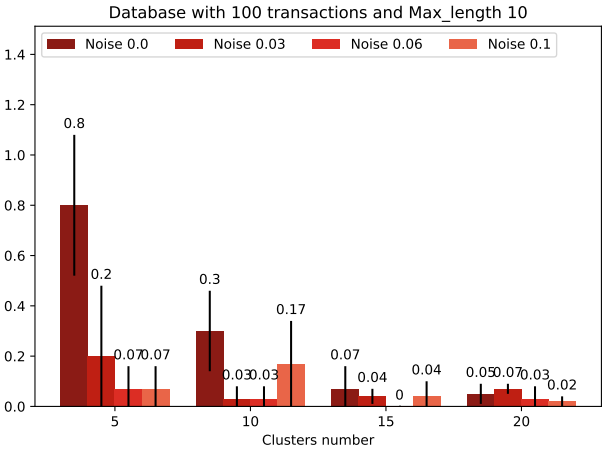

Figure 5.7: Recall plot of 100 transactions of max length 10

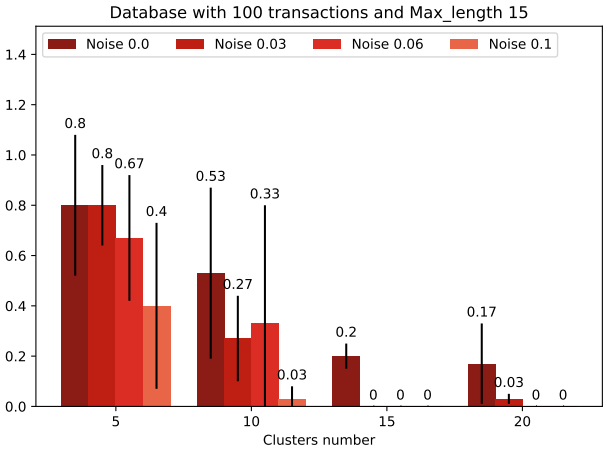

Figure 5.8: Recall plot of 100 transactions of max length 15

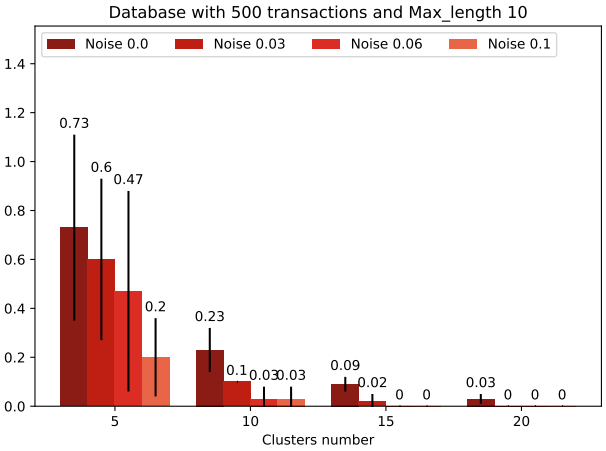

Figure 5.9: Recall plot of 500 transactions of max length 10

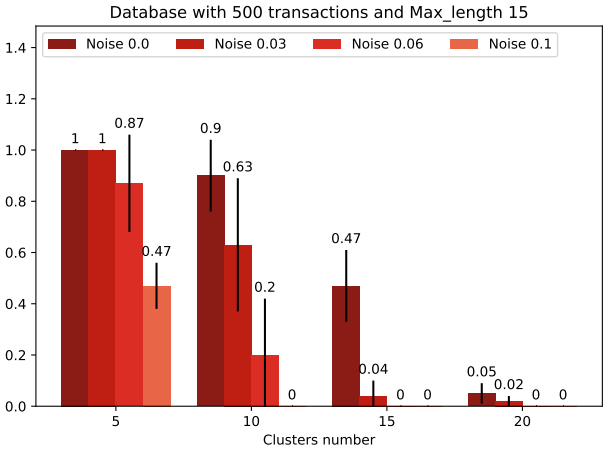

Figure 5.10: Recall plot of 500 transactions of max length 15

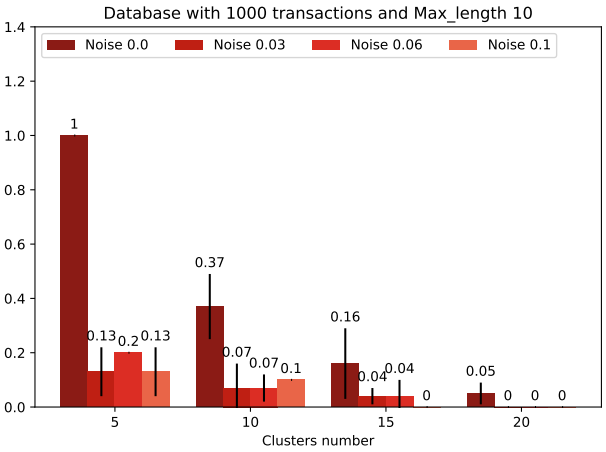

Figure 5.11: Recall plot of 1000 transactions of max length 10

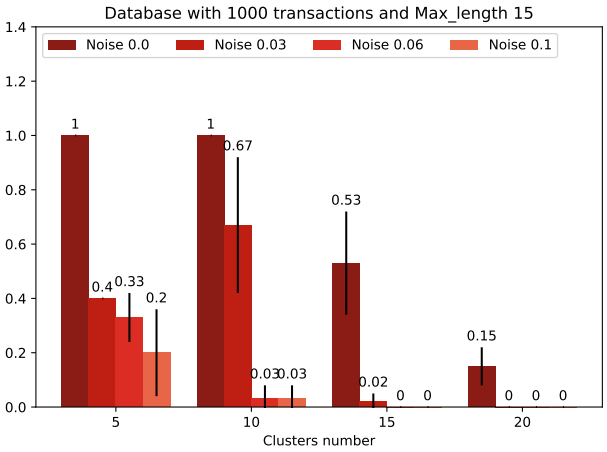

Figure 5.12: Recall plot of 1000 transactions of max length 15

5.2.3 Objective Function Ratio

In this section, the graphs presented show different values of mean and standard deviation of the objective function ratio for each value of noise and number of starting clusters.
This value is computed as the ratio between the objective function value obtained by the representatives found by the algorithm and the objective function value obtained by the starting clusters therefore we have that, in terms of this measure, the closer the ratio is to 1 the better the result.

100 Transactions

In Figure 5.13 and 5.14 are presented the results obtained with 100 transactions and a maximum transaction length respectively of 10 and 15.
As we can see, the results obtained are good since most of the times the ratio is close to 1 even for different number of clusters or value of noise. In particular, in the case of the graph 5.14 the values obtained are almost all very good, while we can notice that in the graph 5.13 we have worse value when the noise and the number of clusters get higher.

500 Transactions

The results are in Figures 5.15 and 5.16. We have that the values get worse than before even if they are still pretty close to 1 most of the times. Moreover, as happened before the results obtained when the maximum transaction length is higher are better.

1000 Transactions

The results are in Figures 5.17 and 5.18. We have a behaviour that is quite different between the two graphs. The graph on the left, in fact, is significantly worse than the one on the right. This could be explained by the fact that a maximum transaction length of 10 is too small for creating a noisy database with 1000 transactions that still could lead to a good clustering.

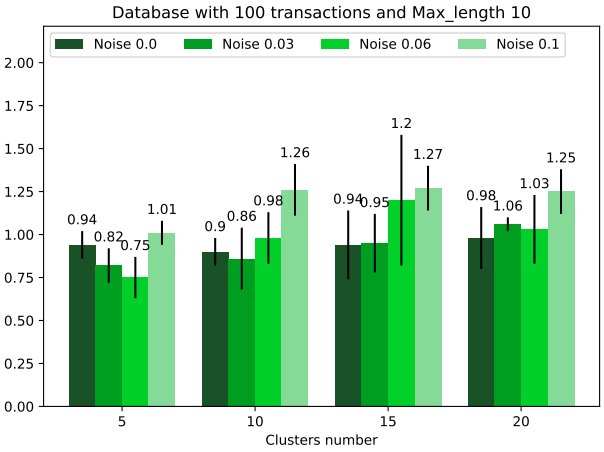

Figure 5.13: Ratio plot of 100 transactions of max length 10

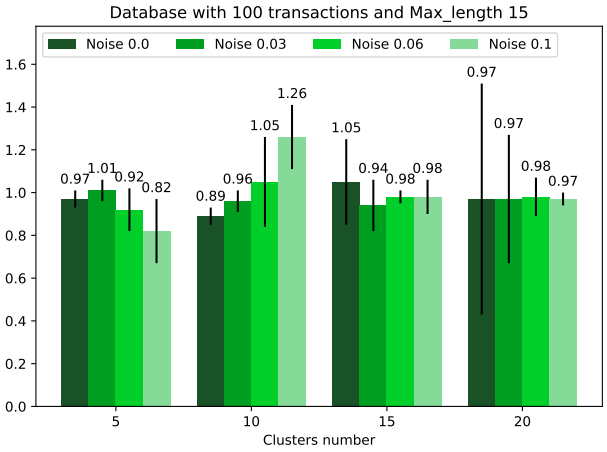

Figure 5.14: Ratio plot of 100 transactions of max length 15

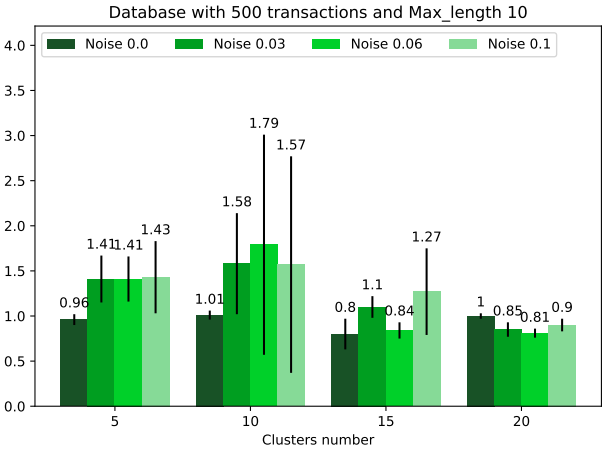

Figure 5.15: Ratio plot of 500 transactions of max length 10

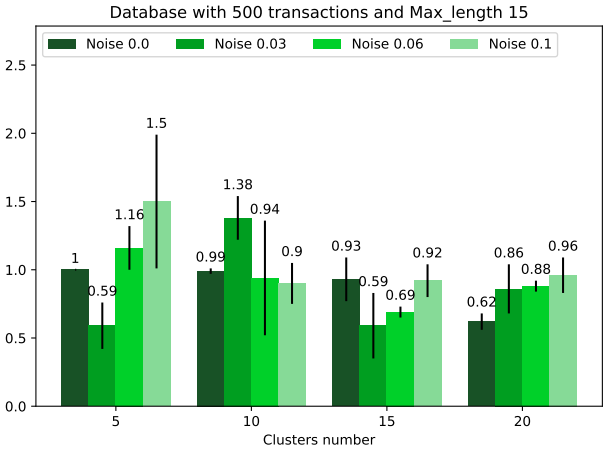

Figure 5.16: Ratio plot of 500 transactions of max length 15

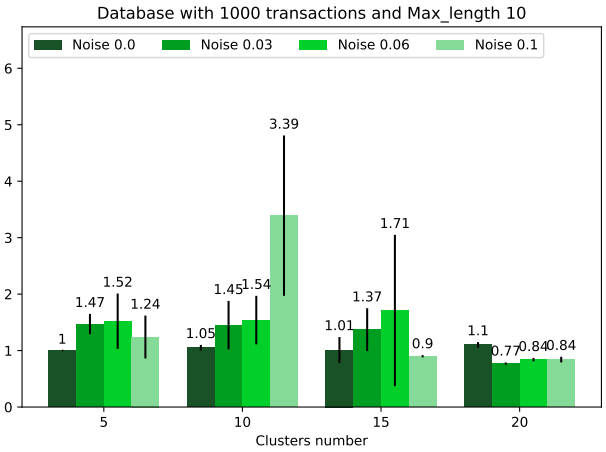

Figure 5.17: Ratio plot of 1000 transactions of max length 10

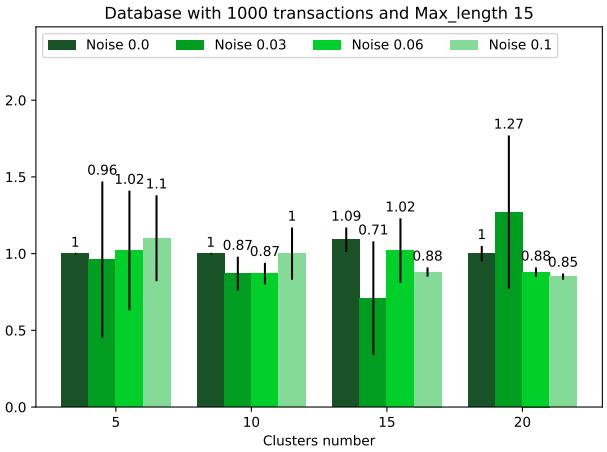

Figure 5.18: Ratio plot of 1000 transactions of max length 15

5.2.4 Time

In this section, the graphs presented show different values of mean and standard deviation of the computing time for each value of noise and number of starting clusters.

100, 500 and 1000 Transactions

Looking at the following graphs one of the first things that stands out is that the y-axis of the graphs on the right have values that are way higher that the ones on the left graphs. This is because the maximum transaction length has a significant influence on the number of subtransactions that can be generated and therefore the algorithm has to work on a bigger search space.
Another thing that may look strange is that the values seem to decrease as the number of clusters gets higher. This, however, can be explained by the fact that if the algorithm has a lower number of starting clusters, it will generate clusters that are more distant from each other and also more heterogeneous. This will lead to a greater number of possible subtransactions and therefore to a greater computing time.
One last thing to underline is that as the number of transactions gets higher, the computing time most of the times increases too as expected.

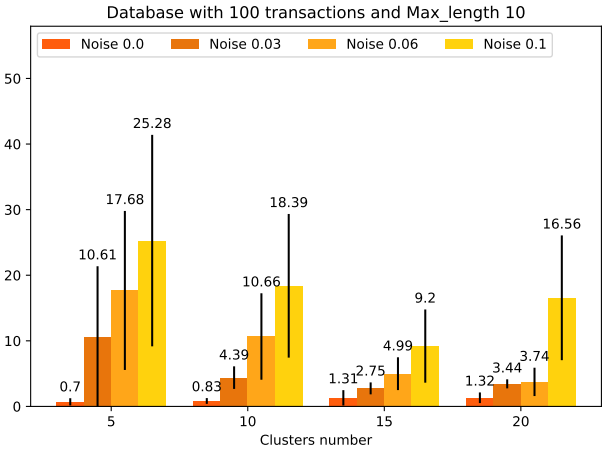

Figure 5.19: Time (s) plot of 100 transactions of max length 10

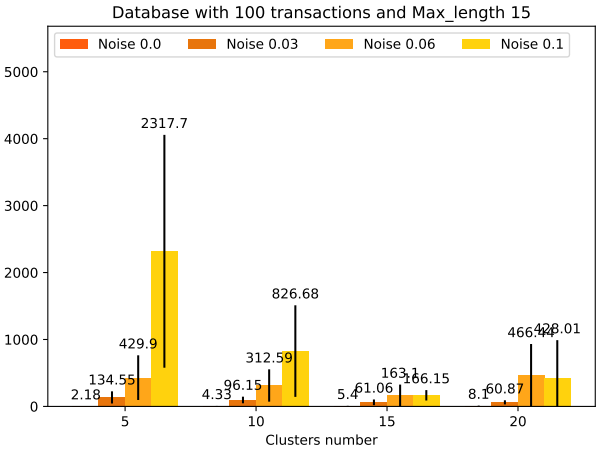

Figure 5.20: Time (s) plot of 100 transactions of max length 15

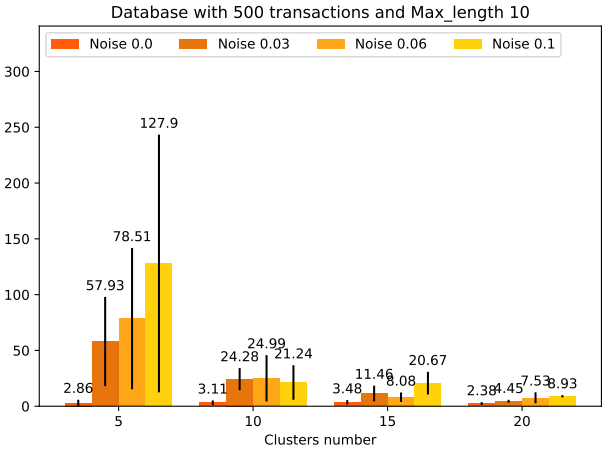

Figure 5.21: Time (s) plot of 500 transactions of max length 10

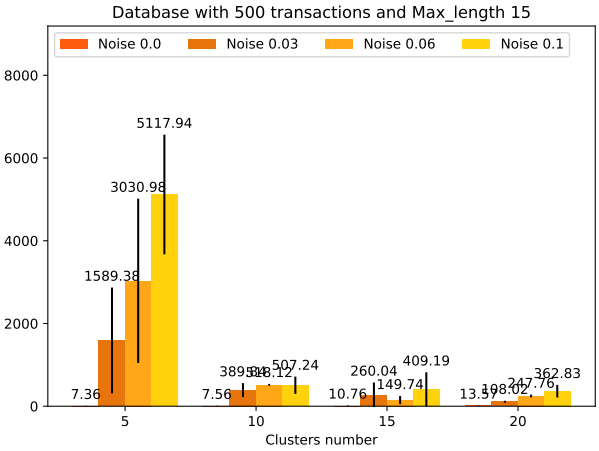

Figure 5.22: Time (s) plot of 500 transactions of max length 15

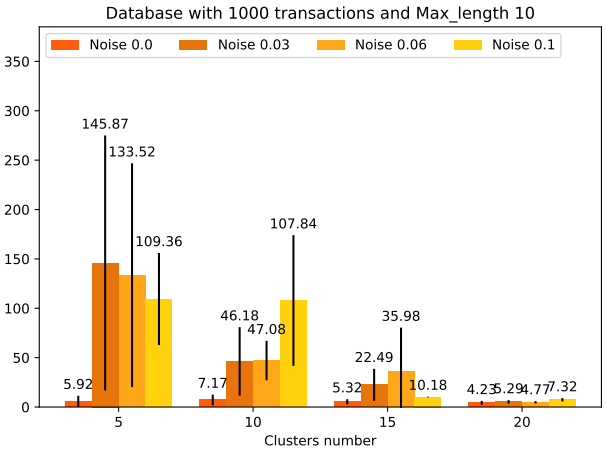

Figure 5.23: Time (s) plot of 1000 transactions of max length 10

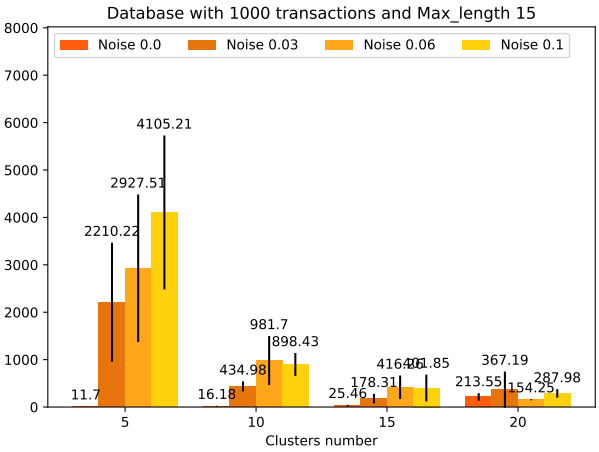

Figure 5.24: Time (s) plot of 1000 transactions of max length 15

5.3 2-2-2 Configuration

In this section, we will present the results regarding the $c1 = 2$, $c2 = 2$ and $c3 = 2$ configuration.
Once we saw the results of the previous configuration we were not really satisfied because most of the times we obtained a number of representatives that was too small. For this reason we decided to test a more balanced configuration to see how the results would have been.
In the following sections we will comment the results we obtained with a database of 100, 500 and 1000 transactions and a maximum transaction length of 10 and 15.
On the y-axis we will have the values associated with the performance measure while on the x-axis we will have the number of clusters that are 5, 10, 15 and 20.

5.3.1 Precision

In this section, the graphs presented show different values of mean and standard deviation of the precision for each value of noise and number of starting clusters.

100, 500 and 1000 Transactions

As we can see the 2-2-2 configuration has better precision results than the 3-2-2 configuration. In fact, the values are all generally higher regardless of the database parameters.
As it was for the previous configuration, the precision still strongly depends both on the number of clusters and the noise values because it gets worse as they get higher.
Moreover, the results obtained by this configuration do not differ much between the graphs but we can underline that the best results are probably obtained on a small database of just 100 transactions.
Finally, we can notice that when the maximum transaction length is equal to 15 the algorithm has pretty good performances especially when there is not any noise.

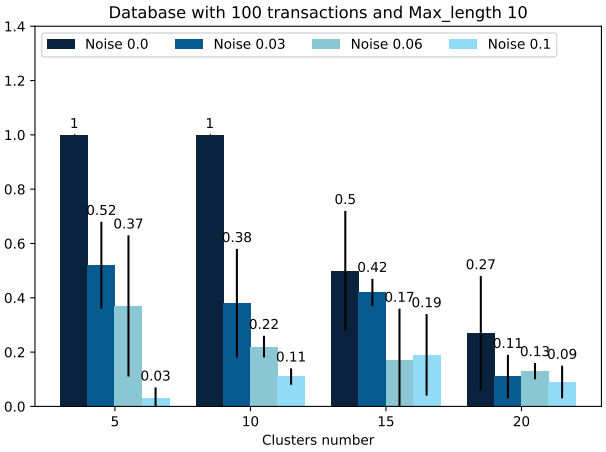

Figure 5.25: Precision plot of 100 transactions of max length 10

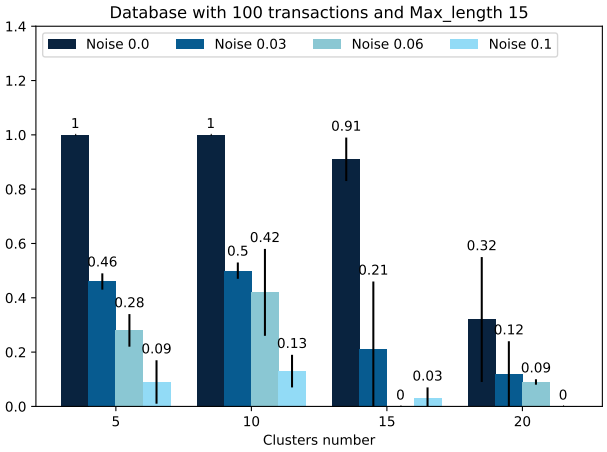

Figure 5.26: Precision plot of 100 transactions of max length 15

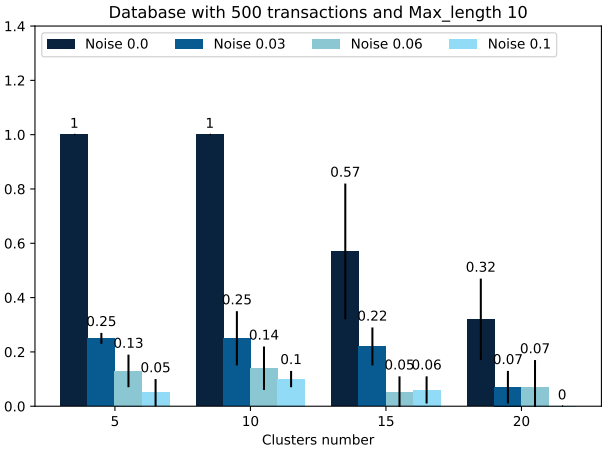

Figure 5.27: Precision plot of 500 transactions of max length 10

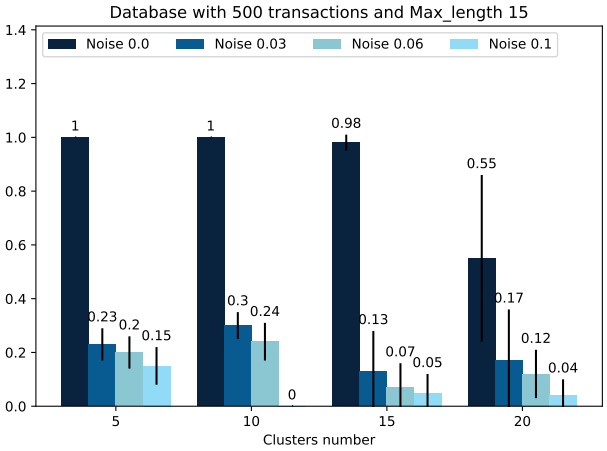

Figure 5.28: Precision plot of 500 transactions of max length 15

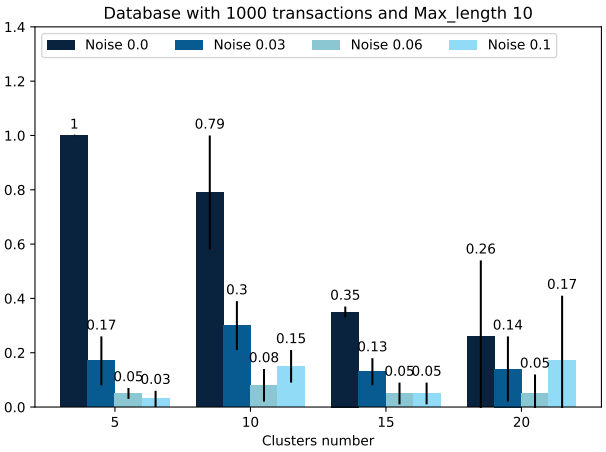

Figure 5.29: Precision plot of 1000 transactions of max length 10

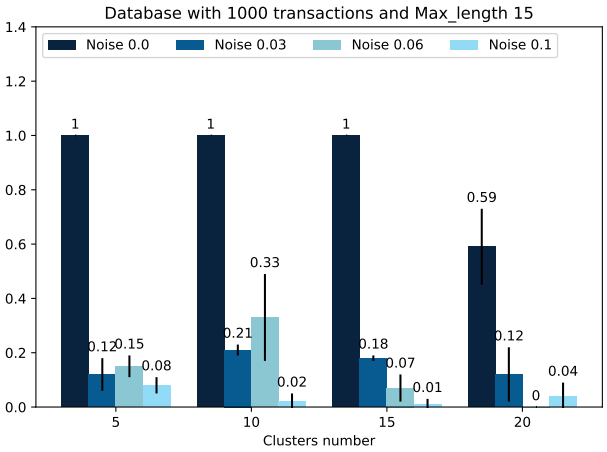

Figure 5.30: Precision plot of 1000 transactions of max length 15

5.3.2 Recall

In this section, the graphs presented show different values of mean and standard deviation of the recall for each value of noise and number of starting clusters.

100, 500 and 1000 Transactions

As it was for precision, we can see that the 2-2-2 configuration has better recall results than the 3-2-2 configuration. In fact, the values are again all generally higher regardless of the database parameters. The values still descrease as the number of clusters and the noise values get higher.
Moreover, one aspect that stands out is that the algorithm reaches really good performances when the number of clusters is low. In fact, especially when the maximum transaction length is 15, the algorithm obtains very good values even with high values of noise. This can be explained by the fact that the 2-2-2 configuration tends to find more clusters than the previous one and so the number of false negatives will be smaller.
Finally, we can notice that, as it was for precision, when the maximum transaction length is equal to 15 the algorithm has generally better performances especially when there is not any noise.

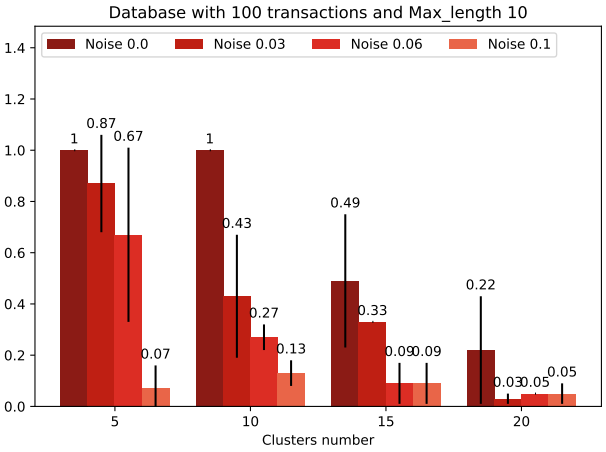

Figure 5.31: Recall plot of 100 transactions of max length 10

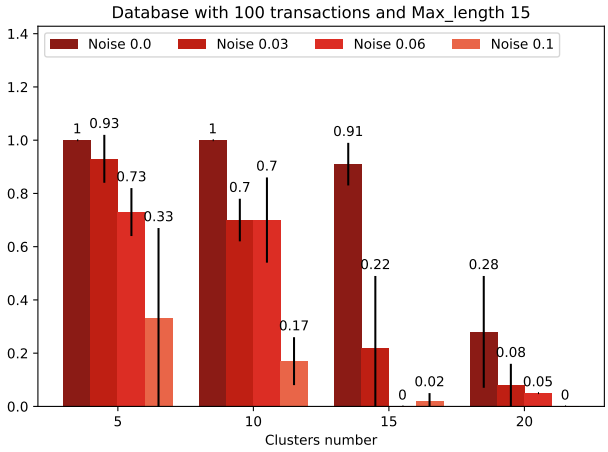

Figure 5.32: Recall plot of 100 transactions of max length 15

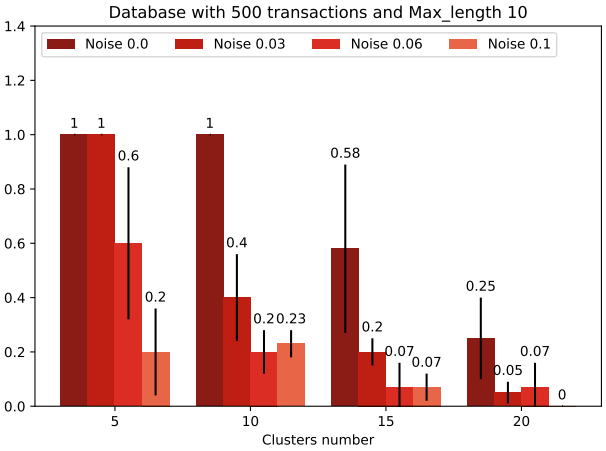

Figure 5.33: Recall plot of 500 transactions of max length 10

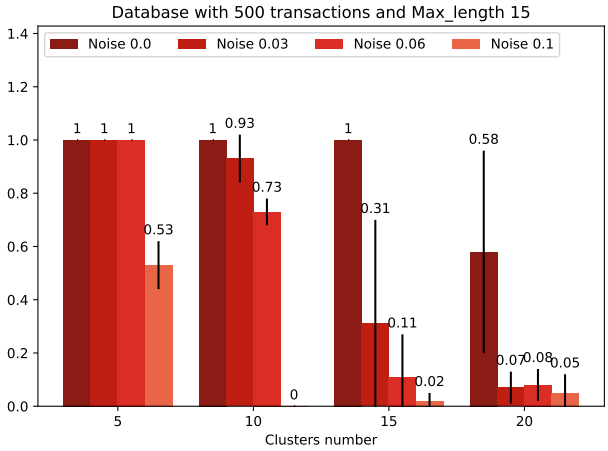

Figure 5.34: Recall plot of 500 transactions of max length 15

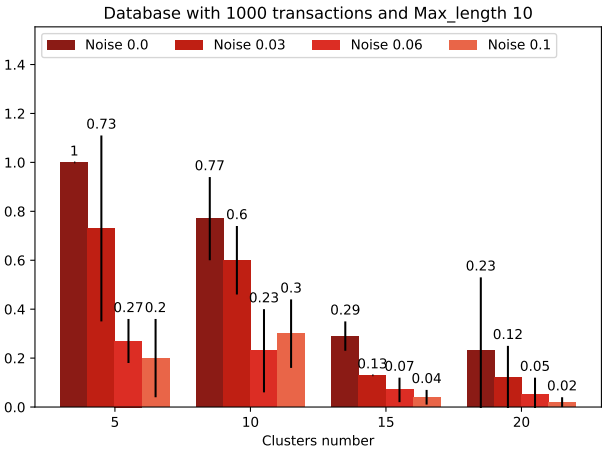

Figure 5.35: Recall plot of 1000 transactions of max length 10

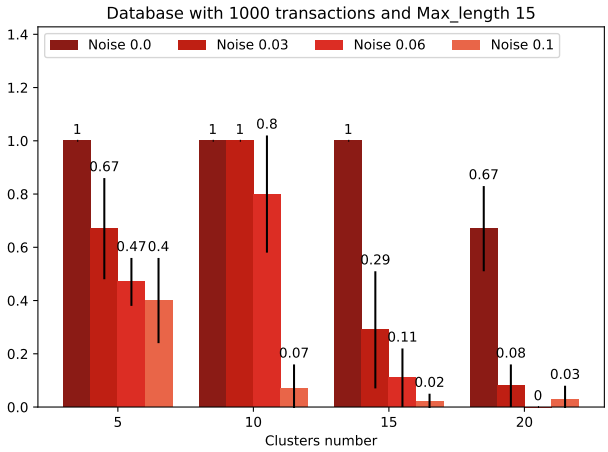

Figure 5.36: Recall plot of 1000 transactions of max length 15

5.3.3 Objective Function Ratio

In this section, the graphs presented show different values of mean and standard deviation of the objective function ratio for each value of noise and number of starting clusters.

100 Transactions

In Figure 5.37 and 5.38 are presented the results obtained with 100 transactions and a maximum transaction length respectively of 10 and 15.
The results in these graphs are all quite close to 1 and so the performances of this configuration do not differ much from the ones of the 3-2-2 configuration (that were already very good).

500 and 1000 Transactions

In these graphs instead we have that the values are a bit worse than the ones obtained by the previous configuration. Even if in general they are still pretty close to 1, this time there are more situations in which the value is greater than 1.3/1.4 especially when the maximum transaction lenght is 10. In fact, both on the database with 500 and 1000 transactions, we can see that the algorithm perfoms better when the maximum transaction length is 15.

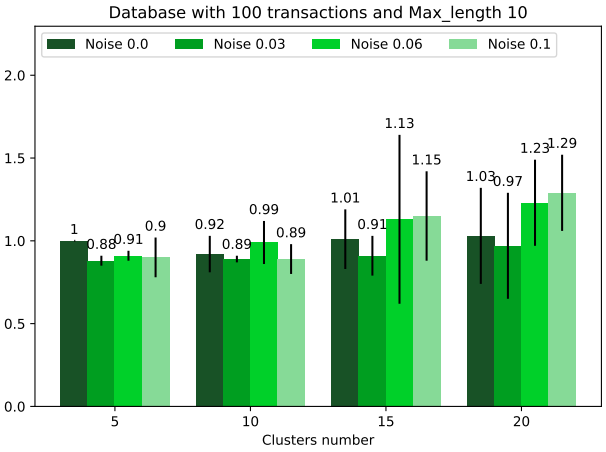

Figure 5.37: Ratio plot of 100 transactions of max length 10

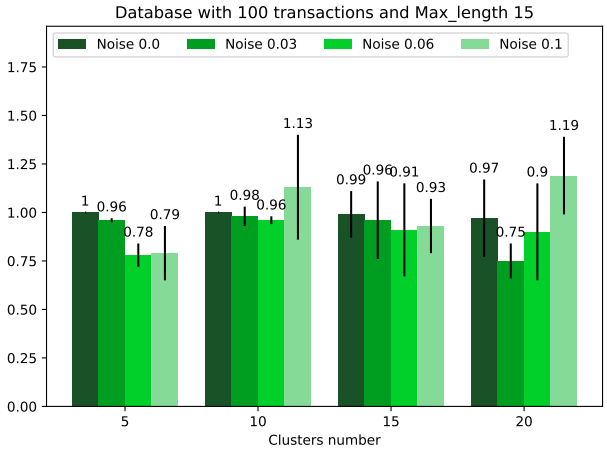

Figure 5.38: Ratio plot of 100 transactions of max length 15

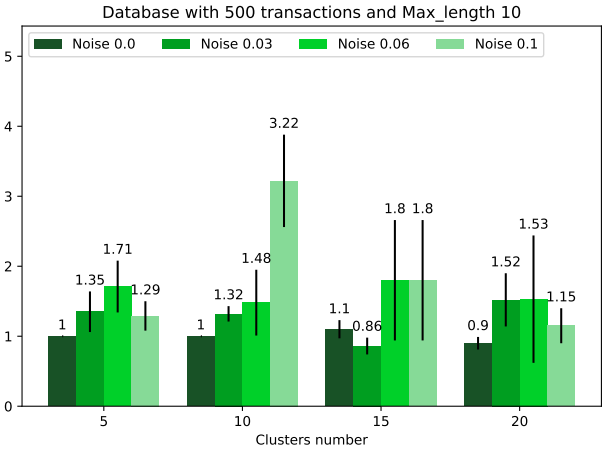

Figure 5.39: Ratio plot of 500 transactions of max length 10

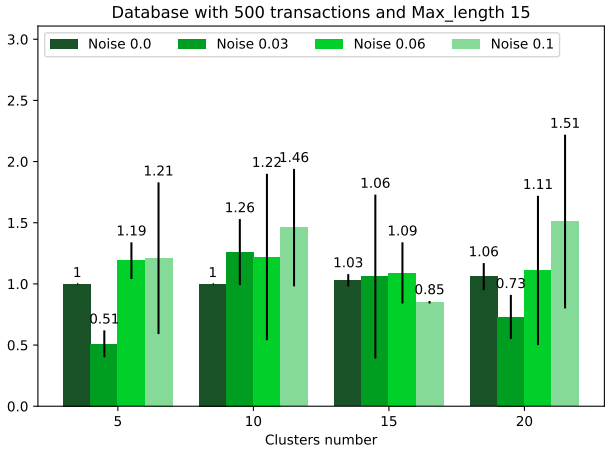

Figure 5.40: Ratio plot of 500 transactions of max length 15

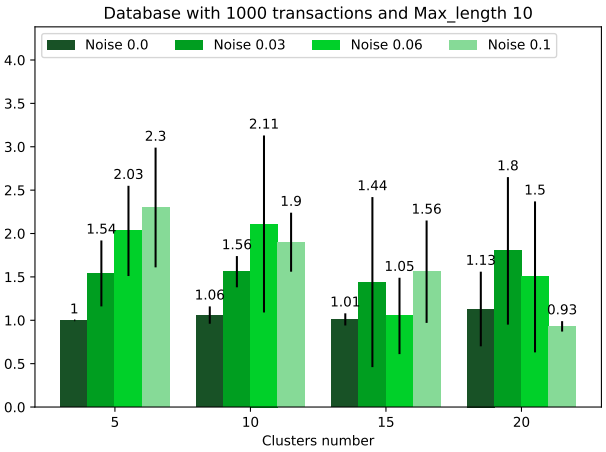

Figure 5.41: Ratio plot of 1000 transactions of max length 10

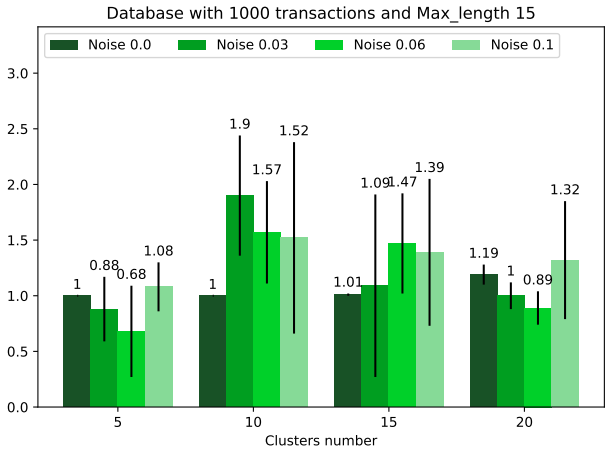

Figure 5.42: Ratio plot of 1000 transactions of max length 15

5.3.4 Time

In this section, the graphs presented show different values of mean and standard deviation of the computing time for each value of noise and number of starting clusters.

100, 500 and 1000 Transactions

In these graphs we can notice the same exact behaviours that we commented in the previous configuration. In fact, the distribution of the bars of these graphs is almost the same of the time graphs obtained by the 3-2-2 configuration.
The only thing that changes is which values are on the y-axis because the 2-2-2 configuration is significantly slower than the other one. This can be explained by the fact that generally the 2-2-2 configuration finds more representatives and so it does more iterations of the algorithm.

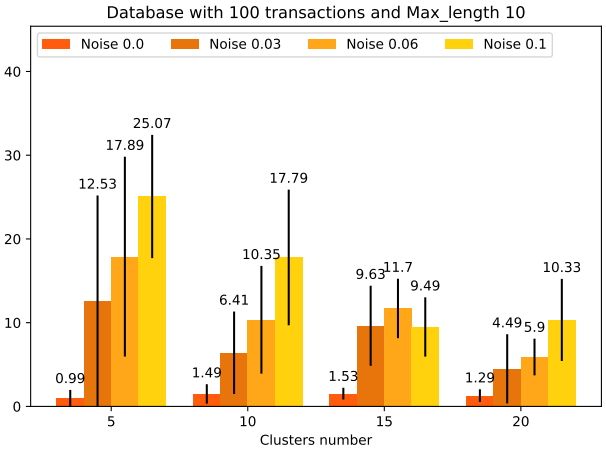

Figure 5.43: Time (s) plot of 100 transactions of max length 10

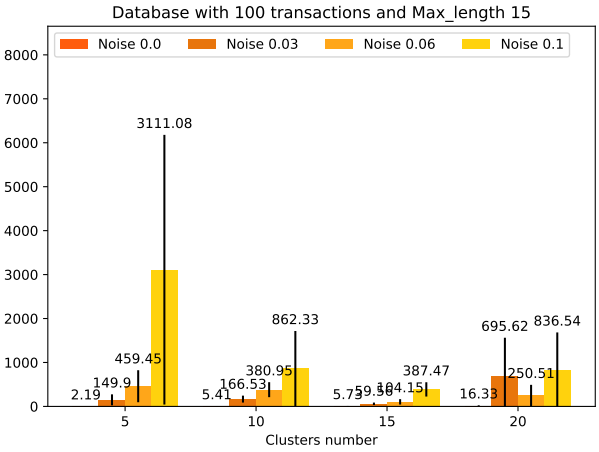

Figure 5.44: Time (s) plot of 100 transactions of max length 15

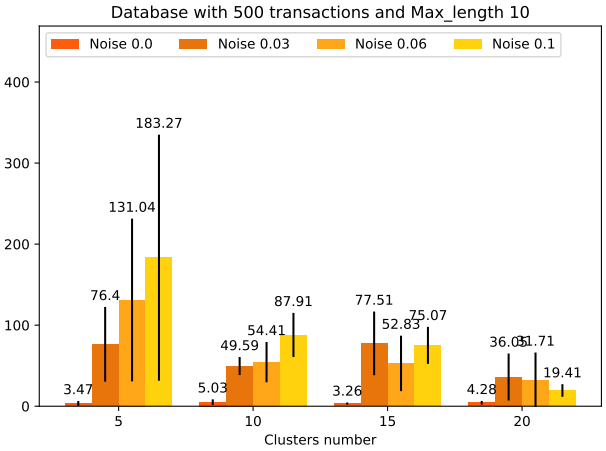

Figure 5.45: Time (s) plot of 500 transactions of max length 10

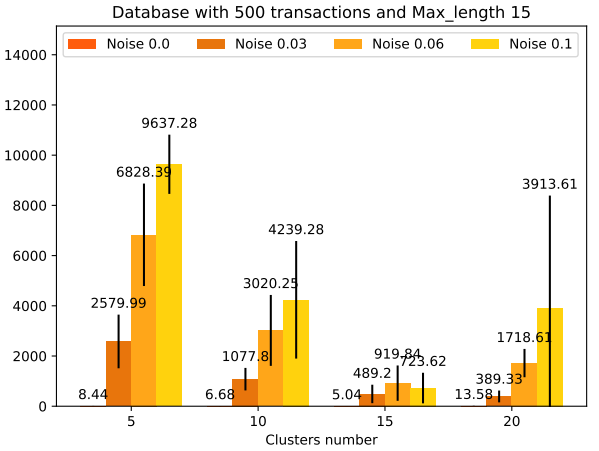

Figure 5.46: Time (s) plot of 500 transactions of max length 15

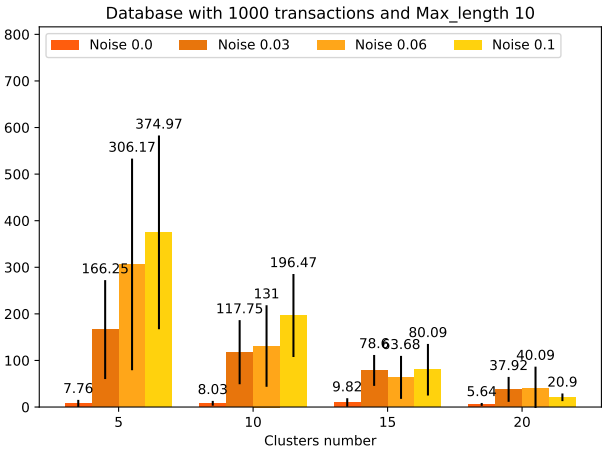

Figure 5.47: Time (s) plot of 1000 transactions of max length 10

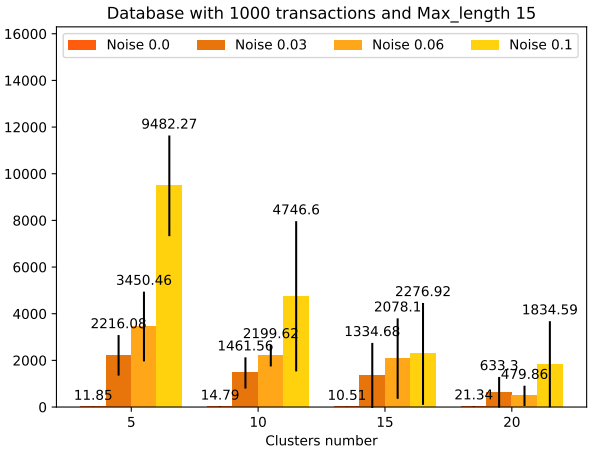

Figure 5.48: Time (s) plot of 1000 transactions of max length 15

5.4 Final Considerations

In general, both the configurations do not have good perfomances on precision and recall. However, we already discussed about why this does not totally affects the quality of the results.
Precision and recall, in fact, even if they are good performance measures, do not capture all the aspects of the results obtained. This being said, we cannot ignore their values on the different situations.
Regarding the differences between the two configurations, we have that they both have pros and cons:

- the 2-2-2 configuration has significantly better results on precision and recall,
- the 3-2-2 configuration is way faster,
- the 3-2-2 configuration obtained better results on the objective function ratio, even if the difference is not outstanding.

Given these informations, we cannot say with certainty which configuration is the best one because it could really depends on the input database and on what we want to achieve.

6

Conclusion

In order to draw some conclusions, it is necessary to remember what was the goal of our work: retrieve the main informations from the input database so as to work with more manageable data.
The approach we chose to use is a clustering model whose original version works very well with a database of trajectories. Our clustering approach adapted that model to our context in order to find the informations we needed in the different clusters we computed.
The results obtained with this approach were not completely satisfactory. As we have already seen in the Results chapter (see Chapter 5), in fact, the results on precision and recall of our tests were not as good as we desired.
This however does not necessary mean that what we obtained is not useful. Precision and recall, in fact, do not capture all the aspects of a result and so, for this reason, we presented also the graphs about the ratio of the objective function values that are more promising.
Moreover, we conducted numerous tests on different databases using always the same algorithm parameters. This obviously lowers the performances because parameters such as *min_supp* and *overlap_threshold* as well as *c1*, *c2* and *c3* may require a tuning for each input database in order to obtain the best result.
This being said, we still believe that a clustering approach may be a good solution to the representative itemsets mining problem especially if its parameters are tuned on purpose.

References

[1] Arno Siebes, Jilles Vreeken, and Matthijs van Leeuwen. "Item Sets that Compress". In: *Proceedings of the Sixth SIAM International Conference on Data Mining, April 20-22, 2006, Bethesda, MD, USA*. SIAM, 2006, pp. 395–406.

[2] Pankaj K. Agarwal et al. "Subtrajectory Clustering: Models and Algorithms". In: *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. PODS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 75–87. ISBN: 9781450347068.

[3] Rakesh Agrawal and Ramakrishnan Srikant. "Fast Algorithms for Mining Association Rules in Large Databases". In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 487–499. ISBN: 1558601538.