Frequent Pattern Mining in an Oriented Labeled Single Graph

[redacted]
1[redacted]
[redacted]

[redacted]
[redacted]
[redacted]

ABSTRACT

This report describe and show the realization and the application of an algorithm used to find relevant frequent subgraphs given a single huge graph as input.

1 INTRODUCTION

1.1 Graphs depict the world around us

Graphs are to be considered an useful tool in order to represent relationships among data and how strong they are, moreover with an immediate visual scheme of the given inputs/outputs. The core of the matter is: graphs are one of the most powerful instruments to depict the world around us. Over the last decades, the use of graphs has exponentially grown, this caused by the multiple applications in which it is necessary to have a simple model able to show all the complicated relations among the data[7]: an obvious example is the chemistry research, in which graphs are used to illustrate the type of atoms and bonds involved within a chemical compound. This compact way of representing information results useful in discovering the various connections among entities, such as relationships among social network accounts or documents within a web-site. In the light of this, it seems reasonable to think about a mining strategy over this special, simple and powerful structure: performing mining on graphs can help to understand properly and directly what kind of relation occurs among the elements and, as a result of this process, it could be possible to obtain useful informations in order to understand the true nature of these associations.

1.2 Who cares about it?

Through the years were developed several algorithms in order to efficiently recovery interesting patterns within a graph structure, thanks to the huge amount of possible implementations. This kind of mining is famous in the computer science literature as the *"frequent pattern mining on the graphs"*, useful to find out, as for instance, the interesting bonds among some proteins or discover the relevant connections among accounts within a social network. Combining all these aspects, it's clear that this topic will be central in the data mining field, leading to the developments of new frequent pattern algorithms.
Consequently, data scientists should be able to handle a great amount of data and research them if they have nothing available. Phone companies are the primary example of industries deeply interested in the development of data mining techniques: this is natural and obvious, most likely because they have access to daily generated data, having the necessity to apply many processing tasks on them[8]. Telecommunications companies can use data mining procedures in order to improve their businesses, identify possible frauds and more in general access customer base informations in order to try to forecast the future marketing movements.

1.3 Our Problem Interpretation

The algorithm presented in this paper should be useful and heavily linked with a real application: for this important reason, it arose the idea to adopt a realistic, specific instance for a frequent graph mining problem. The main idea is to conceive an input single graph as a description of the phone traffic occurring among different zones:

- The nodes represent different zones similarly to geographic areas: as a matter of fact, we can think about these nodes as region of 1/2 km^2. Each node then has a label describing what kind of zone is or what are the main activities related to that zone, e.g.: the label Movement can represent an area where there are airports, seaports, train stations; the label Commercial can describe zone where are present a lot of industries or shopping centers.
- The edges describe the most voluminous call flow from a zone to another at one specific hour. The distance between the two zones is no more than 10/15 km.

The discovery of frequent subgraphs related to these constraints:

- The zones reported in the candidate subgraphs are all different.
- The discrepancy between the hours (reported as weights on the edges) is no more than four.

can lead the users of this application (such as phone companies) to gain various advantages: the first one is that frequent subgraph with few edges can help to understand properly the relations among geographical zones, permitting a targeted design of new calling plans that could be proposed to the customers involved in the interested areas, using an already existing commercial base. The second one is that frequent subgraphs containing a lot of relations can be viewed as a "web of interests" existing among the various areas: it could help the phone company to understand how the various nodes influence each other, making sure that previously unobserved affiliations can emerge.

In the picture above, the circles represent the different areas which can be found on the map e.g: U is the label for an Urban zone, T stands for a Touristic area and C stands for a commercial one. The edges summarize the phone call traffic as previously described. The goal of our application is to find patterns like the graph

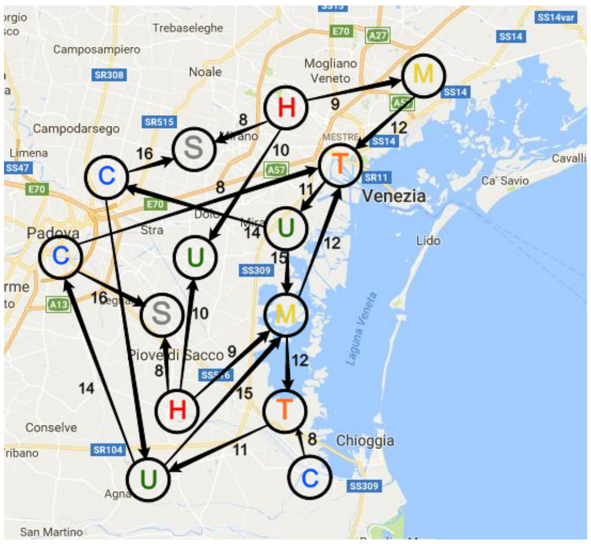


Figure 1: *Our problem Interpretation.*

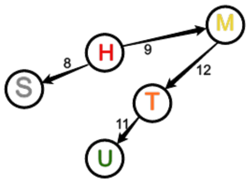


Figure 2: *Example of candidate subgraph, basing on the image reported as Figure 1.*

which appears two different times.

1.3.1 Privacy issues. A brief parenthesis should be open, in order to understand what is one of the main reason which brought telecommunication sector to be interested in Data Mining: as it told previously, phone companies hold a great quantity of data, and most of them are customer informations, sensitive data included. Legal restrictions about personal data processing are to be considered in order to understand how companies are supposed to behave with customer informations[8]: for this reason, the problem instance described in this report took into account this subject by describing generic geographical zones and call flows, without direct references to sensitive client data, considering the Italian privacy laws active nowadays [10].

2 RELATED WORKS

2.1 The foundations: the Apriori approach and the Threshold value

In this scenario, observing interesting connections among specific users or geographical areas could result very useful for the phone company: it would be possible to design new kind of subscriptions or services, with immediate benefits for the media industries as well as for user and other types of companies. The problem, in this case, was generalized in order to have an instance similarly to what happens in the reality: the algorithm presented in this report starts receiving in input a graph, having nodes and edges representing geographical zones and the hour in which occurs most of the calls in a certain time interval, respectively. The main challenge is to make the resulting outputs really useful for applications within the telecommunication business: how it can be possible to retrieve useful informations from a great mass of data? How the operator can realize to have found out something interesting, within a complicated knot formed by nodes and edges? At this point, everything will become clearer introducing two basic concepts of the algorithm presented in this report and more in general in the frequent pattern mining field: the Apriori approach and the threshold value.

The problem of frequent pattern mining is to discover relationships among a set of items within a database: for this reason, it is proper to define a value τ, defined more in general as **minimum support threshold**. Thus, if an item appears at least for a certain fraction τ, it should be considered frequent. This concept is basic in Data Mining, being central in the market basket problem, in which we are interested in finding frequent groups of items bought together. Consequently, it is important to proceed step-by-step, with the guarantee that the solution will be built on frequent items, which are in turn derived from other frequent entities too: so, it is fundamental to build, at each passage, $(k+1)$ frequent items which have to come from frequent k-patterns. The **Apriori approach** depicts perfectly this kind of procedure, which is largely used in the Join-based algorithm typology[9].

Algorithm 1 Apriori approach

```
1: procedure APRIORI(DATABASE D ,SUPPORT $\tau$)
2:     Generate frequent patterns for the step 1 and 2 of the algorithm.
3:     $k \leftarrow 2$
4:     while $F_k$ is not empty do
5:         Generate $C_{k+1}$ by using joins $F_k$.
6:         Prune the solution space
7:         Generate new candidates in $C_{k+1}$ respecting to the support $\tau$
8:         $k \leftarrow$ k+1
9:     return $\bigcup_{i=1}^{k} F_i$
```

Once the sets are scanned, the frequent items were retrieved, a prune of the solution space is performed and the new frequent item set is added to the total result. In this way, it is assured that all the solutions presented in the resulting set are frequent. This approach can be easily transposed to a frequent subgraphs generation problem, in which new candidates for the solution are created adding nodes or edges: if it is proven that a new subpattern is frequent, it is stored and will be used in order to built new candidates. This is the basis of the mining algorithm presented in this report: we will see it in detail later.
The **threshold value** is another important aspect in the mining of graph substructures: it's not only a value for which some candidates have to be rejected in case they can't overcome it, but moreover it

can be considered *a measure of the density of qualified intermediate solutions*[1]. Taking again the chemistry field as an example, the threshold can be assigned in order to describe the strength of the bonds among some atoms: thus, it can be possible to make emerge interesting recurring patterns, permitting to identify unseen chemical properties.

2.2 Further developments: the Pattern-Growth approach

On the basis of the Apriori approach, many algorithms were designed through the years, in order to discover all the frequent subgraphs within a graph. As we've seen previously, the Apriori is easily applicable on graph structures, and this is possible by extending frequent candidates using nodes or edges. An implementation using a node-based candidate generation process was accomplished through the **AGM algorithm**[7]: briefly, this algorithm, starting from a single node, increases the subgraph dimension adding one vertex each iteration. Simultaneously, an edge-based candidate generation approach was adopted for the **FSG algorithm**[7]: it uses a strategy in which subgraphs grow by adding one edge every time the procedure is called. However, both this algorithms suffer some issues derived from the Apriori approach[2]:

- The join of the candidates creates a considerable overhead, lowering algorithm performances
- Since subgraph isomorphism is a NP-complete problem, there's no algorithm able to solve it in a polynomial time. Thus, if we try to implement a search of the false candidates, the performances can be even worser.
- The generation of subgraph candidates is really expensive, especially when the threshold is low or the length of the pattern to be generated is too long.

For this reason, many of the newer algorithms are designed following a *Pattern-Growth approach* instead of the Apriori one.
Substantially, the **Pattern-Growth approach** preserves the data units for mining, in order to recall the basic frequent nodes/edges avoiding a continuous database re-scan[7].This method examines partitions of the data instead of doing it every time in their entirely, with each partition corresponding to a pattern to be analyzed: this is a *divide-et-impera* logic, which permits to reduce the solution space and increase the algorithm performance. The Pattern-growth philosophy can be adapted to graphs: a frequent subgraph mining algorithm following this logic adds new edges to a substructure in every possible way, without performing any expensive join operation. However, this type of subgraph candidate generation has an important drawback: it could be possible to generate the same candidate multiple times. The use of the *Rightmost extension technique* can solve efficiently the problem.

Among the various algorithms developed with this approach, it is useful to outline the **gSpan**, able to avoid the excessive cost mentioned above about the candidates generation and the build of same substructures multiple times: this is possible thanks to a strategy based on a *depth-first search visit (DFS)*, opposite to what happens in algorithms adopting an Apriori approach which use a *bread-fist search visit (BFS)*[7]. The duplicates creation is avoided through the Right-most Extension technique. Taking inspiration from the gSpan, our algorithm uses the **minimum DFS code**, which is a unique identifier for each candidate subgraph reaching the required threshold. The DFS code is explained in detail in the Solution section.

Algorithm 2 Pattern-Growth approach

```
    procedure Pattern-Growth(Database D ,support τ, frequent itemset p)
2:      Scan D, counting frequent items: $F = \{i_1, i_2, ..., i_n\}$
        Sort items in F in ascending order, basing on the frequencies
4:      for item $i \in F$ do
            $D_{p \cup \{i\}} = \Phi$
6:      for itemsets in D do
            Remove infrequent items from D
8:          Sort remaining items according to F
            Insert new itemsets into $D_{p \cup \{i\}}$.
10:     for item $i \in F$ do
            Output $s = p \cup \{i\}$
12:         Pattern-Growth($D_s$, τ, s)
            RightmostExtension($D_s$)
```

2.3 The Constraint problem: GraMi

Before to proceed to the algorithm presented in this report, it is appropriate to highlight one fact: most of the frequent graph mining procedures are performed over a set containing many graphs, while it is clear that the aim is to find all the possible substructures within a single large graph. For this reason, it could be difficult to define a proper support measure, since there could be many subgraphs overlapping each other. If overlaps are permitted, there could be the possibility that the anti-monotonicity property will be not respected. But, what is the **anti-monotonicity property**? It's what we said previously in many different ways, leading us again to the foundation of Data Mining: a certain constraint C results *anti-monotone* if all the subpatterns satisfy C, and all the superpatterns too[9]. Thus, making sure that all the previous candidates respect the property of being frequent is nothing less than making sure that an anti-monotone property is maintained valid for all the possible patterns.
Recent frameworks such as SiGram were developed in order to handle single huge graph, and these kind of algorithms follow a *grow-and-store method*: all the nodes appearing at least τ times are stored, then they are extended in order to build potential frequent subgraphs and finally this two steps are reiterated until it will be not possible to create other potential solutions. However, the creation and storage of the candidate substructures make this kind of algorithm really expensive in terms of computational costs[1].
GraMi, the framework which inspired the algorithm illustrated in this report, takes the advantage of memorizing only the templates of the candidates, reducing consistently the computational weight of the procedure and the whole frequency problem to a **constraint satisfaction problem (CSP)**: GraMi takes for every iteration the minimal set necessary to evaluate subgraphs frequency and, once

the CSP is solved, the remainings subgraphs are ignored. This CSP philosophy permits to GraMi to maintain an anti-monotone tendency, along with a proper support measure, since counting isomorphisms may be misleading due to the fact that a subgraph can appear less times than its extension. The measure adopted in GraMi is the **minimum image based support (MNI)**: the main idea is to discard simple overlapped subgraphs without harm the anti-monotonicity property[7]. In this way, the problem is merely reduced to the respect of some given constraints, rejecting harmless overlapped candidates and making sure that all the solutions are frequent.

3 PROBLEM STATEMENT

In this section a formal problem definition is given recalling the probelm approach that was followed for GraMi [1].
The problem that we analysed refers to looking for, inside a single huge labelled oriented graph, frequent subgraphs given in input two values:

- A *support threshold* τ that represents the number of occurrences for a specific subgraph to be considered frequent
- A *size threshold* K that is used to bound the number of edges present inside the frequent subgraph

Before defining the problem statement, it is appropriate to introduce the principal elements that are going to be treated.

Definition 3.1. **Oriented Graph** An oriented labelled graph can be represented by a triple **G**=(V,E,L) where **V** represents a set of nodes, each of them is univocally identified by an *id*; **E** is the set of edges describing binary sorted relations among nodes for which hold that $(u, v) \neq (v, u)$. **L** is the set of labels that are attached to nodes and edges.

Definition 3.2. **Subgraph** Given a graph S=(V_s, E_s, L_s) and a graph G=(V,E,L) S is a *subgraph* of G *if and only if*:

- $V_s \subseteq V$
- $E_s \subseteq E$
- $L_s(v) = L(v)$

The nodes and the edges are drawn from the original graph with label preservation.

Definition 3.3. **Subgraph Isomorphism** Given S=(V_s, E_s, L_s) and G=(V,E,L) an isomorphism of S in G is a bijective function $f : V_s \rightarrow V$ if and only if the sequent conditions are satisfied

- $L_s(v) = L(f(v))$
- $(f(u), f(v)) \in E$
- $L_s(u, v) = \mathrm{L}(f(u), f(v))\ \forall (u, v) \in E_s$

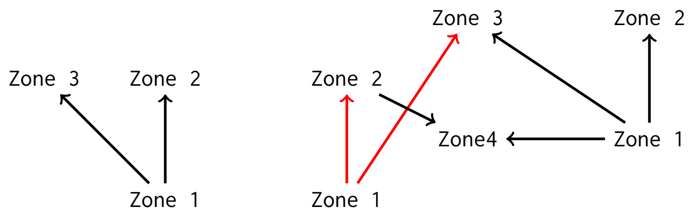


Figure 3: *Example of a possible subgraph isomorphism between the graph on the left and the graph on the right*

Since all the elements have been defined, the formal definition of the problem is the following:

Definition 3.4. **Problem Definition** Given a graph G, a *support threshold T*, *size threshold K* and a support function *C*, a frequent subgraph is a *subgraph S* of *G* such that $(C(S) = n) \wedge (n \geq T)$ and it has no more than K edges.

The aforementioned support function C is the function that allows to count the number of times that a candidate subgraph S appears in the super-graph G. This function defines implicitly the metric that is used to classify a candidate as frequent or not with respect to the *anti-monotone* property.
This property is used to obtain a correct result and prune the search space of possible solutions. It guarantees that, whenever a candidate subgraph S is identified as frequent, also all its subgraphs have to be frequent at least as much as S. The constraint imposed by the *anti-monotone* property doesn't allow us to simply count the number of isomorphisms that are present in G, because it is very likely that subgraph S is more frequent than the nodes that are part of S. Thus, to count correctly the number of occurrences, the *support function* is reduced to a CSP (Constraint Satisfied Problem) where the number of solutions of this new problem will be the exact number of appearances of a particular candidate even if there are some overlaps between them.
The CSP problem is made up of these elements:

- A set of variables X_s, one for each node v present in the candidate subgraph
- A set of domains D_s, one for each variable
- A set of constraints among the variables

Given a subgraph S, a number of variables for each node are created in the subgraph and, for each of them, a domain is defined. Each domain corresponds to a set of nodes in the super-graph G that have the same label of the node associated to a variable in S.

The frequency of a candidate will be counted in order to calculate the number of valid assignments for the variables relatively to their domains. An assignment is considered valid if it respects these constraints

- $\forall\ x, x' \in X_s\ x\ \neq\hat{}x'$
- $\forall\ (x, x') \mid (v, v') \in E_s \rightarrow (x, x') \in E$

- $\forall\ (x, x') \mid (v, v') \in E_s \rightarrow L_s((v, v'))\ =\ L((x, x'))$

The three aforementioned constraints says that an assignment for the variables in the CSP (so a possible valid map between a candidate S and the supergraph G) is valid if all the variables have an unique value, different respect to the others, and if it is respected the existence of the edges and labels associated to them. The number of valid assignments represents the instances candidate number in the supergraph G.
Now, it is appropriate to define the metric measure that it is used in order to maintain the anti-monotonicity of the results:

Definition 3.5. **Minimum Image Based** Let be:

- $f_1, ..., f_m$ the set of isomorphisms of a subgraph S(V_S, E_S, L_S) in a graph G.
- F(v) = $f_1(v), ..., f_m(v)$ the set that contains the (distinct) nodes in G whose functions $f_1, ..., f_m$ map a node $v \in V_S$.

The *minimum image based support (MNI)* of S in G, denoted by $\sigma_G(S)$, is defined as: $\sigma_G(S) = \min\{t | t = |F(v)|$, for all $v \in V_S\}$.

4 SOLUTION

The solution shown in this section recalls the approach presented in the GraMi framework but it is adapted and implemented from scratch with respect to the main context in which we conceived this work: mining on a directed graph representing calls from different zones into a particular geographic area.
The idea is to identify frequent oriented subgraphs given a size threshold. More specifically, we are looking for subgraphs for which their labels are all different among each other and the calls (relation) inside these graphs have no more than 4 hours of difference.
At the end of the algorithm, the frequent subgraphs that have been found are merged together using a specific similarity metric. This step is necessary, since we are dealing with oriented graphs and it is most likely to happen that the algorithm returns a set of subgraphs that are very similar to each other, maybe differing just for one label, or simply containing edges with reversed directions. To avoid the presence of hundreds different solutions with redundant informations, similar graphs are merged together building multi-edge graphs.

4.1 The Algorithm

The heart of the algorithm that solve the aforementioned problem is the following:
The philosophy that resides behind this algorithm recalls the *Apriori Approach* which, as have we said earlier, is one of the most important approach present in the literature for solving problems regarding the seek of *frequent itemsets*.
The first step of the algorithm (line 2) is the recognition of the frequent edges in the graph, exploiting the function `countFrequentEd` that takes in input the graph and explore the whole structure in order to find and count all its edges. A single edge inside this function is represented by a triple (*SourceLabel,DestLabel,weight*) and using a simple hash table we are able to count the exact number of time a single edge appears.

Algorithm 3 Main Algorithm

```
procedure COUNTFREQUENTSUBGRAPHS(GRAPH G1,INT THRESHOLD,INT SIZE)
    frequentEdges ← countFrequentEd(G1)
    candidateSet ← genCandidateSimple(frequentEdges)
    for candidate in candidateSet do
        if isFrequent(candidate) < thr then
            candidateSet.remove(candidate)
    while candidateSet ≠ ∅ ∧ sizeOf(candidateSet) ≤ size do
        candidateSet ←
            genCandidateComp(cand..Set,freq..Edges)
        if isFrequent(candidate,G1) < thr then
            candidateSet.remove(candidate)
    return mergingResultcandidateSet_temp
```

The return of `countFrequentEd` defines *frequentEdges* that will be used to construct new candidate subgraphs, extending them using frequent edges.
Each time the next candidates are generated, the count of their occurrences is performed using the function `isFrequent(candidate,G1)` (line 5/9). The function takes as parameters the candidate and the input graph and try to solve the CSP problem for these two elements. If the candidate does not respect the *support threshold*, it is removed.
Then, the entire process is iterated until the *size threshold* is reached or all the candidates in the last generated *candidateSet* haven't exceed the *support threshold*

4.2 Candidate Generation

The generation of the candidates is one of the most important and expensive part of the algorithm and it is possible to analyze two possible different cases:

(1) The first one is referred to function `genCandidateSimple`. This function takes as input the *frequent edges* set and try to combine couple of edges in order to create all the possible subgraph with at least two edges

(2) The second one, instead, regards genCandidateComplex that takes in input *frequentEdges* and a set of subgraphs. To obtain a set of candidates with an higher number of relations (dimension), for each candidate subgraphs we look for all the possible frequent edges that can extend them with the add of a new node or edge between two nodes that already exist.

Since we are dealing with labeled graphs, it's very likely to generate candidates that already exists. Briefly speaking, we need to verify the existence of isomorphisms among the candidates. To solve the problem, the *DFSCode* from gSpan is used [2].

4.3 DFS Code

The *DFS Code* is a technique introduced by *gSpan*[2] and reused by GraMi to verify if two graphs are isomorphic. During the candidates generation, this code is used to verify if a new created graph already exists. So, the DFS Code can be thought as an unique identifier of a labeled graph. To be more precise, during the elaboration of the DFS Code for a specific graph we are looking for the *minimum DFS Code* given a total order on the edges and a lexicographical order on the labels. As shown in the gSpan paper[2] if two graphs have the same *minimum DFS Code*, they are isomorphic.
A single *DFS Code* is a set of tuples (*time1,time2,Label,Weight,Label*) describing the edges of a graph discovered during the `DFS Visit` over that graph. The *DFS Codes* outline in which order the nodes, thus the edges, are discovered and then the edges are rearranged using a total order relation among *forward edges* and *backward edges* exploiting the discovery time of each edge. So given a graph, it is possible to obtain multiple *DFS codes*, since likewise multiple `DFS Visits` exist.
Given a graph G and all its possible DFS Codes, the minimum DFS Code is the smallest DFS Code obtainable by sorting the DFS Codes using both the total edge relation and the lexicographical order among nodes and edges label.

4.3.1 *Implementation.*
At first glance, the implementation of the *minimumDFSCode* is very expensive: the computation of all the possible DFS visits require a lot of time and, since we are dealing with oriented graphs, the problem of reachability rises. To solve these two problems we have followed the "advices " presented here [4][5]

(1) Instead of performing all the possible DFS Visits, we decided to consider to start the visit from the nodes that have the minimum value regarding the lexicographical ordering. Similarly when we reach a specific node, we sort its adjacency list considering again the lexicographical ordering of the labels and their edge-values. If two or more nodes had the same label and edge-value, different sorted adjacency lists would be created considering the permutations among " the same nodes".

(2) To resolve the reachability problem, before computing the *minimum DFS Code*, the input oriented graph is transformed into its non oriented version: the nodes are the same but the relations among them are no more oriented. This little trick resolves the possible unreachability of some nodes given a source root in the graph and makes the visit procedure more easy and direct. Then the direction of the original edges is used to classify a discovered edge as *backward edge* or *forward edge.*

This two simple advices reduce the number of possible visits to perform since in our scenario the label nodes in the candidates subgraphs are all different from each other so it reduces the time required to compute the *minimum DFS Code.* For all the sorting operations, an implementation of the merge sort algorithm is used

4.4 CSP and isFrequent

The CSP, given a candidate subgraph and the input graph G, is the heart of the algorithm and is performed by the function `isFrequent`. The main purpose of this function is to count the number of appearances of a subgraph inside the input huge graph. For every candidate the procedure computes the following steps:

Algorithm 4 Is Frequent

```
   procedure isFrequent(subgraph $S_G$,graph G,hashTable domains)
       for edges ∈ $S_G$ do
           restrictedDomains ← checkEdges(edge,G)
4:     return countFinal(restrictedDomains,G)
```

The first step computed by `isFrequent` is to use the function `checkEdges` for each edge in the candidate subgraph.
For each edge, the last mentioned procedure verifies which possible values the variables, associated to the nodes present in the edge, can take. To do this, we work with the domains related to each label. Given a label, its domain is the set of nodes in the input graph that have that label. The *restricted domains* will result to be an hash table having the same structure of the *domains* but with the valid values that permit to map correctly each edge in the subgraph with the equivalent edge in the super-graph. The steps computed by `checkEdges` are these:

- If both labels in the edge are seen for the first time, the procedure uses their "original domains" to construct all the couple derived from the *cartesian product* among the two domains. Then the correct existence of each couple and the related edge weight are verified in the input graph. The couple of values that pass the test populate the *restricteDomains*, the others are discarded.
- If one or both labels are already seen, the list of couples is built using the restricted domains instead of the original one. Then the couple of values is verified in the same way of the previous point.

Once all the restricted domains have been elaborated, the `countFinal` procedure is called. This function counts exactly how many times a subgraph appears. To do this, the cartesian product among all the restricted domains is performed, generating a list of possible assignments that will be checked in the super-graph G. For each possible assignment, we make sure that there will be no equal values in the related tuple, and then the super-graph structure is partially visited using the values present in the possible assignment. If at least one value is not mapped correctly in the input graph, the tuple is not counted as valid solution. At the end, the number of valid tuples will correspond to the number of occurrences in the input graph.

4.4.1 *How does isFrequent satisfies the constraints in CSP ?.*
First of all we make sure that each tuple, so each possible assignment having all different values, respects the first constraint. Then the existence of the edge both in the subgraph and in the graph is checked two times, the first with the the use of `checkEdges`, the second

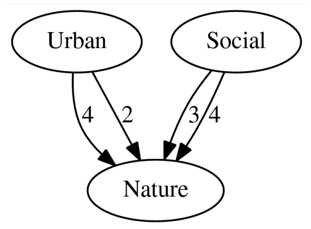


Figure 4: *A sample subgraph result generated by the algorithm*

with `countFinal`. At the the end the correspondence between the labels is assured by the fact that the possible values, for a variable, are generated from the *domains* and *restricted domains*.

4.5 Merging the frequent subgraphs

As previously mentioned, the set of the frequent subgraphs could be copious, and it is very likely that a lot of graphs could be very similar to each other. In order to reduce the number of redundant informations, it was decided to merge similar graphs using the well-known **Jaro similarity** among the *minimum DFS Codes* (transformed as strings) of the frequent subgraphs. If the size of the solution is 3, the similarity threshold used is 0.9 (90%), otherwise for bigger size the threshold used is 0.8 (80%). The merge operations is performed unifying the adjacency among nodes with the same labels preserving the structure of the merged graphs.

4.6 Technical Implementations Details

The algorithm was totally implemented in Python 2.7 in an imperative way. The main libraries that have been used are : `re` for the regular expressions, `numpy`, and `itertools` for the generations of the permutations list and cartesian products.
The graphs have been described using the *adjacency list* and, in order to be more specific, at each graph is associated a python `dictionary` structure which works as an hash table:

```
graph g1={n1:[(n2,w1),(n3,w2)..], n2:[..], ...}
```

In the above `dictionary` the keys values are the *id* of the nodes, while the value associated to it is the list of couples, where the first element correspond to another nodes in the graph and the second element is the hour associated to that edge. The use of an hash table to represent a graph facilitate the immediate access to a specific node and to all its connected neighbors. Related to the aforementioned structure, there's another dictionary, called `graphLabel`, which maps each node in the graph to its corresponding label:

```
graphLabel g1={v1:[Label1], v2:[Label2], ...}
```

4.7 Obtained results

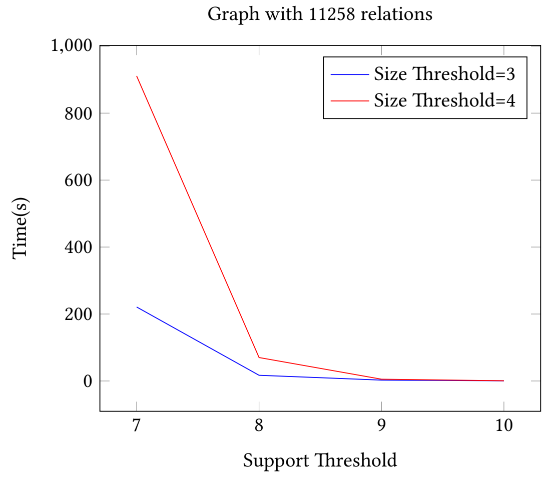
The Figure 4, displayed in this page, illustrates a possible result subgraph produced by the algorithm. Notice that the result is modeled under the problem described above, so it is supposed that the operator is strongly interested in having relevant results, able to show interesting interconnections among the geographical zones. E.g.: the figure illustrates an interesting call flows from two zones, labeled as Urban and Social respectively, directed to a natural zone (denominated like this due the presence of a natural park, as for instance). This subgraph could be seen as a "highlight" where a steady stream of calls occurs: it represents a relevant information for the phone company, which could be able to design and propose new "ad hoc" services and subscriptions in order to support and enlarge the customer base present in that area.
The presence of a multi-edges relation among the nodes means that this result was produced thanks to a merge of two similar subgraphs, which was detected thanks to the Jaro Similarity realized on two DFS Codes. Suppose to have among the results this subgraph:

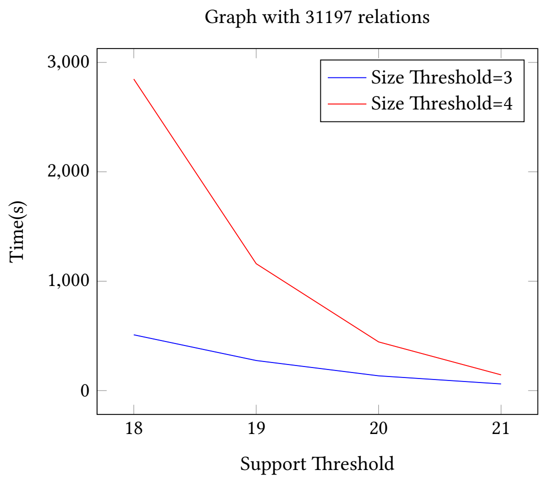
```
graph g1=v1:[(v2,10)], v2:[(v1, 8)], v3:[(v2, 12)]
graphLabel gL1=v1:[University], v2:[Nature], v3:[Commercial]
```

This result shows a strong correlation among three zones: University, Nature and Commercial, respectively. What it could be deduced is that the University and Commercial zones are strictly related to a Nature one. For instance, e.g.: it is possible to think the Natural zone as a central point for the University, probably having some interests due to a presence of a research center. Then, it is possible to suppose that some companies are interested in the research natural area which influences the University zone and probably they are supporting it through private investments. So, a final user, having this data in its hands, going beyond to a simple purpose of calling plans, can decide to fund a new start-up in order to exploit this relation among the University and Nature zones.
Obviously, in order to have the most reliable gateway possible, it's fundamental to know well the territory and have a good knowledge of the existing actors present within the geographical area.

5 EXPERIMENTAL RESULTS AND OBSERVATIONS - DATA MINING

The performance have been measured with the following specs: Intel Core i7-3632QM CPU @ 3.2GHz and 12 GB RAM. The tests for this application have been done with automated generated data: each input graph is the result of a probabilistic method which randomly creates edges between nodes and assigns labels to edges and nodes.





As we can see from the graphs plotted above, the execution times, with a fixed size threshold, lower with the increase of the supported threshold. This happens mainly because at the beginning of our algorithm there's a relevant prune of the edges which don't respect the support threshold. This implies a search space reduction, mainly due to the fact that the number of possible candidates to count and to merge is not so high. It's important because, since we are looking for relevant solutions, the proper usage of the threshold clearly impacts how much useful the results are: thus, using low thresholds will implicate an expensive computational time and the presence of useless results. Notice that: if we use the same threshold for two different size graphs, it implies an exponential time growth within the one with the larger dimension.
Taking in consideration the size of the solutions, using a high size threshold could mean to iterate multiple times the same steps: thus, repeating the whole process for bigger candidates could signify, as a direct consequence, an increase of the computational time. As a conclusion, it emerges that computational time is heavily influenced by two factors:

- the support threshold
- the size of the edges reported in the solutions

The worst case occurs when the support threshold is very low and the size threshold of the solutions is high: that's because the algorithm will execute a lot of iterations in order to perform building and counting tasks, whilst the number of candidates and frequent edges remains big.

5.1 Computational Cost

In this section we present our analysis about the computational cost of our implemented algorithm, the cost is split among the main steps which constitute our procedure.

- **count frequent edges**: $\Theta(V + E)$, since for this step we visit all the input structure.
- **candidate generation simple**: $O(fe^2 * DFSCodeCost)$, where *fe* represent the number of frequent edges
- **candidate generation complex**: $O(numberOfCandidate* fe * DFSCodeCost)$
- **minimum DFSCode**: $O((V_s+E_s)*|Perm(V_s)|*nlogn(E_s))+ nlogn(|Perm(V_s|)$, V_s and E_s represent the edge and the nodes of the candidate subgraph. $Perm(V_s)$ represent the number of all possible permutation of the nodes V_s
- **isFrequent**: $|cartesianProduct(restrictedDomains)|*\Theta(V_s+ E_s)$
- **restricted Domains**: $O(E_s)$, is a simple iterations over the subgraph edges.
- **merge of the frequent element**: $O(numberOfSolution^2)* \Theta(DfscodeString^2)$

The cost related to the calculation of the *minimumDFSCode* is theoretically very expensive due to the number of possible DFS Visits that can be obtained starting the visit using, at each time, a different source point and considering at same time all possible permutation of the adjacency lists. Reasoning about our application, the candidate subgraphs will have nodes with all different label. Forcing each DFS Visit to start only from nodes with the minimum lexicographical value and sorting the adjacency, reduce the number of possible DFS Visit thus the time required to calculate the *minimumDFSCode*. As can be viewed from the points above the time required to solve the problem is very high and it strongly depends on the *support threshold* used, that influences the number of edges and candidates considered, and on the chosen size (size threshold) for the frequent subgraphs.

6 A PARALLEL APPROACH: THE BIG DATA SOLUTION

6.1 A brief introduction

Nowadays, programming systems are designed to be parallel oriented thanks to a "cluster computing" approach: a large collection of hardware commodities, such as processors or computational nodes, are provided to programmer and they are exploitable through the usage of frameworks which are built on distributed file-system architecture. These kinds of file-system features much larger memory units than the usual ones used in a standard single machine architecture and also provide protection against the frequent data and hardware failures: corrupted data can be easily fixed through the exploitation of the redundancies and there's the chance to solve hardware issues thanks to the available network of nodes on which the environment is splitted. Moreover, there is the possibility to build a parallel environment, through a set of cheap hardware components that are designed into the form of nodes and racks [12]. Usually, all the nodes are coordinated by the "supervision" of a Master node, that is responsible of the distributed data management among the Workers network.

6.2 Apache Spark

Apache Spark is a platform for fast general purpose cloud-computing. It provides specific APIs for various programming languages such as Java and Scala, moreover having features such as easy usability, immediate implementation of complex algorithms and the possibility to combine different kind of computation tasks (e.g.: SQL queries and machine learning)[11]. Spark is based on Hadoop and, subsequently, on HDFS which is the current open-source implementation of a cloud parallel computing environment. Noticeable is the fact that Spark supports optimization for handling complex data structures, such as graphs (GraphX) and data streams (Spark Streaming). Every node has to manage a portion of an *RDD (Resilient Distributed Dataset)*: it consists in a set of elements that are subdivided among the nodes network in order to perform some specific tasks.

6.3 MapReduce

MapReduce is a typical parallel computing approach that is based on two main functions: the *Map* and the *Reduce* ones.

- The *Map* function is responsible of the *key-value* pairs generation: a set of input elements, distributed among the various nodes, are transformed into pairs formatted with a key and its respective value (e.g.: `(key, value)`). As for instance: given a set of a repeated elements, it could be possible to implement a count application for verifying how many times an element appears. So, in a MapReduce solution, the Map function generates couples in the form `(element, 1)`, where the element represents the key and the number 1 represents a temporarily initial count value.
- In the *Reduce* step, the Master node performs the task of elements collecting having the same key: it groups the data with an equal key within a bag, then for each key there is a node that performs the Reduce function on the basis of its input value. Taking again the previous example, for each element used earlier as key, there is a node performing the sum of the 1s which were mapped in the Map step.

6.4 Algorithm implementation

The algorithm previously explained has been developed using the Scala language with the Spark framework integration. Since Scala is an object oriented language, it was decided to represent the main elements of the algorithm as objects. So, the graphs processed by the algorithm are encoded with two main classes: `MyGraph` and `MyGraphInput`. Both these classes implement the graph data structure using the adjacencies list approach: every graph contains a list of `VertexAF` nodes, each of them having as attributes the label and the list of adjacencies to contiguous nodes. Since our candidate subgraphs have only distinct labels, the nodes that appear in the `MyGraphInput` have an attribute working as a unique reference.

```
class MyGraph() extends Serializable {
  var nodes: List[VertexAF]=List.empty[VertexAF];
  var dfscode: String=""
  var maxH:Int=0
  var minH:Int=0
  ...
}

class VertexAF (id:String) extends Serializable{
  var adj: List[(VertexAF,String)]=List[(VertexAF,String)]
  val label:String=id;
  ...
}
```

It was decided to design the input graph differently from the GraphX environment provided by Apache Spark. Since GraphX conceives graphs as two RDDs in order to handle nodes and edges, it was impossible to split among the various nodes the multiple computation of the subgraphs, due to our main purpose, that is the exploitation of the MapReduce approach.

Thus, the algorithm works through the usage of RDDs populated by objects produced by the previously described classes. It is possible to identify two main RDD instances: the first one describes a set of candidate `MyGraph` graphs, whilst the second one contains the frequent edges useful to perform the subgraph extension task. As it is noticeable, the main input graph is never subdivided among the nodes because it was used in the main MapReduce function.

Algorithm 5 Big Data

```
    procedure COUNTFREQUENTSUBGRAPHS(GRAPH G1,INT THRESHOLD,INT SIZE)
        freqEdges ← countFrequentEd(G1)
3:      candSet ← freqEdges.cartesian(freqEdges)
            .map(candSimpleGen(couple))
        candSet ← DFSMapReduce()
        candSet ← CSPMapReduce()
6:      while candSet ≠ ∅ ∧ sizeOf(candSet) ≤ size do
            candSet ← candSet.cartesian(freqEdges)
                .map(candGen(couple))
            candSet ← DFSMapReduce()
9:          candSet ← CSPMapReduce()
        return mergingResultcandidateSet_temp
```

Here above the *Big Data* pseudo code has been reported and it follows the same steps present in the *Data Mining*. The `candSet` and the `freqEdges` are both *RDD* containing respectively the candidate subgraphs and the frequent edges present in the input graph.

The *candidate generation* phase is splitted into two different steps:

- The first one, line 3, aims to generate the first set of candidates, the ones that have 2 edges among 2 or 3 nodes. At the beginning a *cartesian* product of the `freqEdges` is performed and then from all the couples that respect the prerequisites a set of new candidates graphs are built. Since the cartesian is performed between two equal sets, in order to reduce the number of couples that will be analysed from the matrix of the possible couples, it is kept only the lower triangular part.
- The second one, line 7, perform a cartesian product among `candSet` and the `freqEdges` and tries to extend in all possible way each *candidate* graph with one of the *frequent edges*

Then every time a new *candidate* is generated, its *minumum DFScode* is computed as its canonical representation and it is used in a *Map Reduce* approach to discard the duplicate generated candidates. At the end , the core of the algorithm, the CSP problem solving, is performed using a *Map Reduce* approach that is explained in the next section together with the other ones.

6.5 The implemented MapReduce functions

The MapReduce functions that have been used are the following:

The first one performs the delete of duplicate graphs resulting from the extension of the candidates using the frequent edges: as a matter of fact, each subgraph is mapped with its relative DFS code, since it is a canonical representation of the graph and it was used as key. Then, the `Reduce` function will simply keeps one element for each DFS Code

```
app=candidateGen.map(el=>(el.dfscode,el))
ris=app.reduceByKey((a,b)=>a).map(el=>el._2)
```

The second one is used to solve the CSP problem for each candidate, that is the *core* of the algorithm. Thus, it finds the candidates which overcome the threshold value and, as a matter of fact, they are most likely to be part of the solution. This MapReduce function is formatted as it follows:

First of all, on the RDD candidates a Map function is applied transforming it into a tuple where the first element is the candidate *dfscode*, the second one is the input graph, the third one is the candidate graph and the last is the result of a function, `CSP`, which takes as inputs a candidate graph and the input graph and returns a list of all of its possible domains that could satisfy the CSP problem. Then, the result is flatten and mapped into an RDD that is made up of tuples where the first element is a candidate DFS Code and the others are the input graph, the candidate graph and one candidate domain. At the last step of the Map phase, each element of the RDD has, at its second place, the result of the function, `checkG` that is responsible for the candidate domain validity verification: if the candidate domain is present in the graph, the aforementioned function outputs 1, otherwise 0.

```
a=cand.map(el => (el.dfsc,graphIn,el,(CSP(graphIn,el))))
a1=a.flatMap(el => list.map
   (listel =>(dfsc,graphIn,cand,listel)))
a2=a1.map(el => (dfsc,checkG(cand, input, dom)))
```

Subsequently, the reduced function simply sums all the values associated to the same DFS Code and the DFS Code that do not surpass the threshold value will be excluded through the usage of a straightforward boolean filter function

```
ris1=app2.reduceByKey((x,y) => x+y).filter(el=>el._2>= thr)
```

At the end, it was attempted a MapReduce implementation of a function used to merge together similar results, likewise to the Data Mining algorithm described previously. Differently from the serial approach, the same graph could be part of different solutions.

Initially, all the possible couples that could be generated from the solution set are computed. Then, the *Jaro Similarity* is applied among the couples and all of those who do not overcome the similarity percentage threshold are discarded.

It could be performed the MapReduced through the exploitation of the similarity relation transitivity property: if A is similar to B and B is similar to C, it is most likely that A is similar to C. Thus, an idea could be to map each couple `(A,B)` into two different couples `(A,List(A,B))` and `(B,List(A,B))`, then a reduce function could be performed to merge together the lists having the same key, represented by the first element within the couple. This step ideally represents the transitivity property of a similarity relation. The key could be represented by a *dfscode* candidate and the list subsequently should be populated with candidate graphs. At the end of this procedure, duplicated lists should be discarded and, for each remaining list, a multi-graph should be built exploiting the graphs contained within the list.

```
var jarJar = candSet.cartesian(candSet)
    .filter((c1,c2) => c1.dfsc<c2.dfsc)
var jj = jarJar.map((c1,c2) => (c1.dfsc,c2.dfsc,
  jaroS(c1.dfsc,c2.dfsc),c1,c2)).filter(thrsehold)
var jj1=jj.flatMap((tuple =>
  List((c1.dfsc, List(c1,c2)),(c2.dfsc,List(c1,c2))))
var rdbk = jj1.reduceByKey((x,y)=>x++y).map
  (el => merging(listofgraph))
```

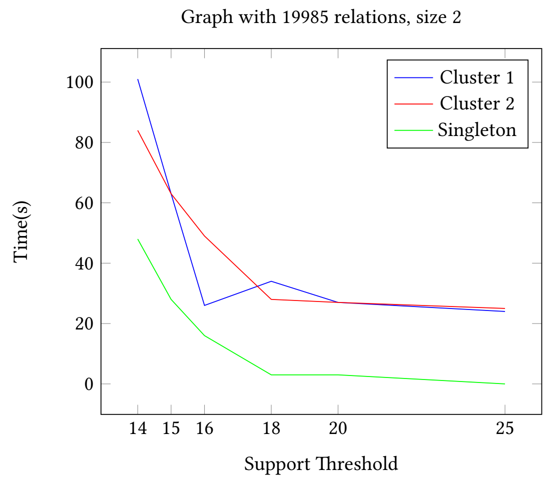
However this idea of *MapReduce* has led to some problems during the implementation phase, thus it has been decided to collect all the solutions that have been computed and to perform the merging in a serial approach.

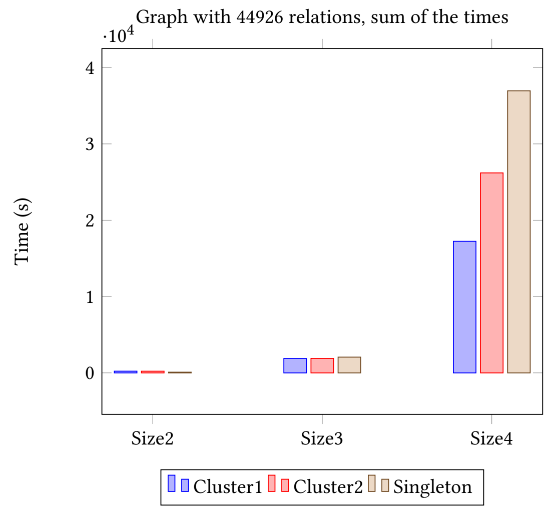
6.6 Experimental Results And Observations - Big Data

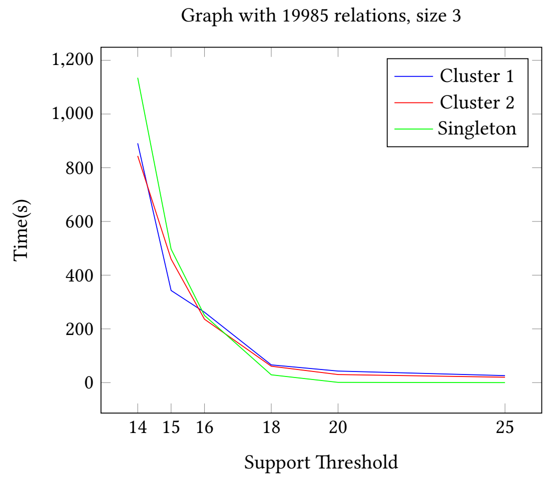
In order to show a comparison between a parallel environment and a sequential one, a series of test has been executed. The performances are evaluated with respect to the computational time, as it was previously done for the Data Mining algorithm. The battery of tests was performed on two different clusters and one single machine:

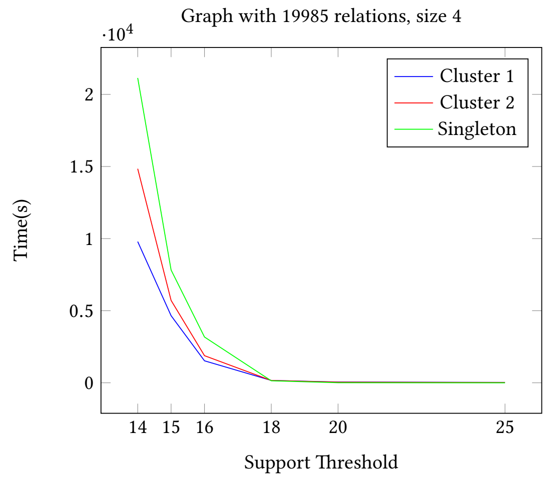
- The first cluster is made up of: 1 master node with 2vCPUs having 7,50GB of RAM, and 6 workers with 1vCPUs, 3,50GB of RAM.
- The second cluster has the following specifications: 1 master and 3 workers having 2vCPUs with 7,50GB of RAM.
- The machine which was used previously for Data Mining tests

All the test have been executed not considering the merging of the solution at the last step.







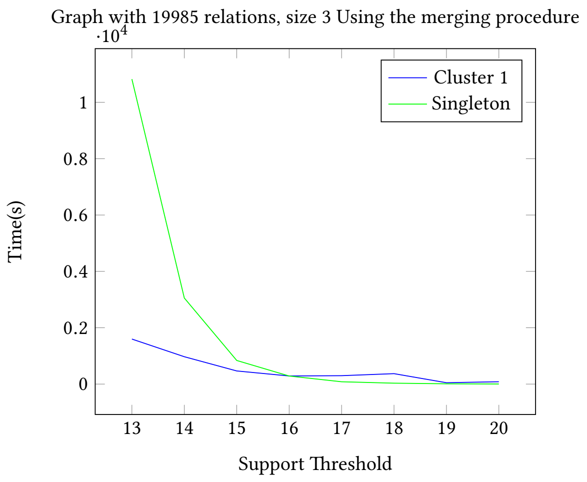


The first 3 plots report the execution time for the same graph, with 19985 edges, using different *thresholds size*. All of them has the same behavior described for Data Mining, the computational time decrease with the increase of the support threshold. The firs plot highlights how the the *serial version* of the algorithm perform better than the two cluster, increasing the gap of performance when the support threshold is very high.

In the last two plots the *clusters*, instead, have better performance for the first support thresholds but at the end the single machine performs equal or better with the respect of the two clusters. It is noticeable also that in the second plot the two cluster have similar performance until the *support* threshold turn to 18, but in the third one the difference between the bigger cluster and the smaller one is more clear.

When a high-density solution is handled by the algorithm, the two cluster report better performances than the single machine. On the other hand, the low-density solution requires much less computational time on the single machine than on the clusters: this because in a parallel environment it is necessary to keep in mind that nodes and master have to communicate and the synchronization system time does not affect the final result when the solution is so much dense. Vice versa, if the solution is not thick enough, the required “system” time influences negatively on the final computational period. This behavior can be highlighted also between the two clusters: it is possible to identify within the plot a “turning point” where the smaller cluster has better performances than the bigger one regarding the computation of “small” solutions

The fourth plot reports the summed time of difference tests, computed on a bigger graph, for a fixed *size*; the support threshold that have been used for the summation are: 23,24,25,26. This bar chart summarize the results obtained with the first three plot and what has been just said: when the solution is very dense the clusters have better performance, suppressing the synchronization cost among the different nodes. On the other side, with less dense solution the single machine perform much better than the other two cluster.



From the last plot it is possible to conclude that the inclusion of the merging method, computed in a serial way also for the parallel approach, does not affect consistently the behavior shown previously. As a matter of fact, the instances solved by the cluster scales well when the solution is much more dense, while the singleton still performs better with thinner instances. This is more evident observing the time tables attached to this project (*timetable.pdf*). To show the way in which the merging of the solution influences both the programs it is important to notice the shifting to the left of the *rendez-vous* point concerning the computational time among the two programs. E.g. : considering for example the instance 16 3, where 16 represents the threshold size and 3 the size of each subgraph in the solution. In the performed test, without the presence of the *merging procedure*, the *Big Data* solution performs 3 seconds better than the *Data Mining* one, but the inclusion of the *merging* allow the serial approach to overcome the parallel one of 2 seconds.

At the end is possible to verify that, in this particular problem, when it is required a lot of computational time, it is better to split the work among more nodes with less computational power than on few nodes with more computational power.

7 CONCLUSIONS

The algorithm illustrated in this paper takes inspiration from the GraMi framework: as a matter of fact, many other implementations try to follow the CSP philosophy, reducing the problem to the respect of some given constraints with the proper use of a metric, able to maintain the anti-monotonicity of the solutions. Nonetheless, this brand new algorithm previously illustrated comes up with an implementation which takes care of directed graphs, modeling a realistic usage within the telecommunication business, while many existing algorithm consider only sets containing multiple undirected graphs as input. The application of credible support threshold permits to the algorithm to report solutions which could represent important evidences for future marketing movements, making sure that there will be a gain of competitiveness for the phone company in the telecommunication market.

7.1 Further developments - Data Mining

One of the possible works that should be realized in order to exploit our application is to design a pre-processing procedure which transforms a possible detailed phone call graph into a more abstract one. In the first graph, the nodes correspond to points on the map that can describe either a source phone call point or a destination phone call one. The second graph, a generalized one too, can be obtained reducing the nearest phone call points and the correlated edges into single elements that will represent different zones on the map.

7.2 Further developments - Big Data

A logical subsequent development is to exploit much more the parallelization that could be gained by the usage of some specific frameworks: first of all, as a consequence of what it was previously illustrated, the algorithm should be adapted to exploit all the powerful functionalities offered by GraphX, leading to serious improvements through the placement of the candidate graphs and the input graph inside dedicated RDDs, able to properly handle them. Furthermore, it could be advisable an HashMap approach for adjacencies lists management, although this implementation would not be possible in the case of a GraphX development.

The algorithm presented in the parallel approach has a theoretical limit due to the fact that input graph is shared among all the nodes every time the CSP problem is solved: this sharing can lead to important space issue management in the case of a huge input graph, this because every worker could not have the possibility to manage such a big object.

REFERENCES

[1] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis *GraMi: Frequent Subgraph and Pattern Mining in a Single Large Graph*, 2014.
[2] Xifeng Yan, Jiawei Han *gSpan: Graph-Based Substructure Pattern Mining*, 2002.
[3] Cane Wing-ki Leung *Technical Notes on Extending gSpan to Directed Graphs*, 2010
[4] Adrien Basse, Fabien Gandon, Isabelle Mirbel, Moussa Lo *DFS-based frequent graph pattern extraction to characterize the content of RDF Triple Stores*, 2015
[5] Cong Yu *CS 6093 Lecture 8, Spring 2011 Advanced Data Ming*, 03/28/2011
[6] Brent E. Harrison, Jason C. Smith, Stephen G. Ware, Hsiao-Wei Chen, Wenbin Chen, Anjali Khatri *Frequent Subgraph Mining, gSpan, chapter 7 from Practical Graph Mining With R; Nagiza F. Samatova, William Hendrix, John Jenkins, Kanchana Padmanabhan, Arpan Chakraborty; Chapman and Hall/CRC, 2013* https://www.csc2.ncsu.edu/faculty/nfsamato/practical-graph-mining-with-R/slides/pdf/Frequent_Subgraph_Mining.pdf
[7] Hong Cheng, Xifeng Yan, Jiawei Han *Mining Graph Pattern, chapter 13, from Frequent Pattern Mining; Charu C. Aggarwal, Jiawei Han; Springer International Publishing, 2014*
[8] Gary M. Weiss *Data Mining in Telecommunications, chapter 56, from Data Mining and Knowledge Discovery Handbook; Oded Maimon, Lior Rokach; Springer International Publishing, 2005*
[9] Charu C. Aggarwal, Mansurul A. Bhuiyan, Mohammad Al Hasan *Frequent Pattern Mining Algorithms: A Survey, chapter 2, from Frequent Pattern Mining; Charu C. Aggarwal, Jiawei Han; Springer International Publishing, 2014*
[10] Decreto Legislativo 30 June 2003 n. 196 published in the Gazzetta Ufficiale n. 174, 29 July 2003 http://www.camera.it/parlam/leggi/deleghe/03196dl.htm
[11] Apache Spark Web Sites http://spark.apache.org/docs/latest/index.html
[12] Jure Leskovec Stanford Univ., Anand Rajaraman Milliway Labs, Jeffrey D. Ullman Stanford Univ. *Mining of Massive Datasets*