Optimal search strategies for significant subgraph mining

Manuel Gerardo Orellana Cordero

A thesis presented for the degree of
Master in Computer Science

Faculty of Science
University of Antwerp
Antwerp, Belgium
13 August 2017

Abstract

Graph data is very common among today's datasets, especially in the field of bio-medicine. Subgroup discovery of subgraphs in graph databases hold great potential for biological data sets. In this instance, the goal will be to search for those specific subgraphs that are highly enriched in a subset of vertices compared to the remainder of the graph. Recently, an algorithm was developed to tackle this problem. While the theoretical basis of the interestingness measure is solid, the search strategy is suboptimal resulting in long runtimes on proof-of-concept implementations. These long runtimes greatly hinder wider application of the strategy.
The goal of this master thesis will be researching the optimal search strategy for the discovery of enriched subgraphs. Therefore, it contains the following objectives:

1. Comparison of the complexity and performance of different search strategies, both in theory and in practice.
2. Development of a method that can dynamically tune the search strategy to best fit the input graph data.
3. Proof of concept implementation of the most efficient strategy, based on memory usage and runtimes.

Dedication

To my family, for all the support and motivation to fulfill my goals.
To my parents, the love I have received from them is beyond the limits of a dedication, without them, I would not have had the opportunity to be here and study this master.
To all the friends who have been around, encouraging me and providing a beer in the hard times.
To Andrés, who has been deeply thought and loved during all this time.
To Kamila, for being an angel whose support and love allowed me to accomplish this work

Acknowledgements

To SENESCYT (Secretaría de Educación Superior, Ciencia, Tecnología e Información), for funding my studies abroad, providing me the opportunity to live and study abroad.

To the Advanced Database Research and Modeling (ADReM) group of the University of Antwerp, for providing the opportunity of this master thesis, specially to Pieter Meysman, for his guidance and willingness to share his knowledge on the topic.

To the University of Antwerp, for all the support to international students, being of great help from day one. It has been a privilege to study in this institution.

Contents

Chapter 1

Introduction

Graph mining is a sub-field of data mining which has been of great interest for different domains. Graphs are basically a complex set of vertices interconnected to each other with edges. Therefore, there are a vast variety of applications for algorithms which discover the frequent subgraphs occurring within a specific graph. Domains of study such as circuits, images, chemical compounds, protein structures, biological networks, the Web, work flows, and XML documents obtain benefit from these algorithms since they can be modelled by graphs [1]. Figure 1.1 illustrates a protein to protein interaction in biology as one possible field of application of graph mining.

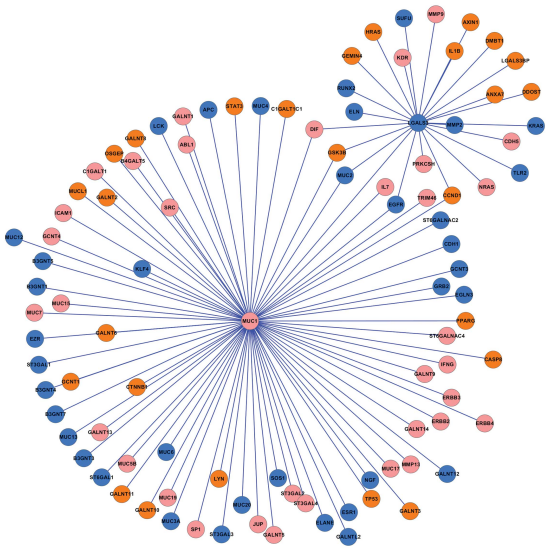


Figure 1.1: Protein-protein interaction (PPI) network of LGALS3. [2]

Graph mining has broad applications within the actual datasets available in all kind of domains but there are very different kinds of mining approaches in this field. Applications such as classification, clustering, and frequent subgraph discovery are some of the applications in which graph mining can be used.

Graphs present a complex structure which makes of this, a domain with many challenges in its exploration. Graph miners around the world have developed different strategies to discover subgraph patterns more efficiently in terms of time efficiency and memory usage. However, the wide majority of these approaches are based on graphs which do not take into account directionality and explores several graphs at a time. The present work explores two of the most popular algorithms to find frequent subgraphs and adapts them to the problem of significant subgraph mining in a single graph which contains a set of interesting vertices. This problem relies on finding frequent subgraphs which have as origin a set of interesting vertices selected before hand; this shows a difference with traditional subgraph mining since the latter, looks for frequent graphs in a dataset, without taking into account any vertex as specially important. The frequent pattern mining with a set of interesting vertices can have a variety of uses, for example, in biology can be used to find the associate structures related to a disease which are not known; with traditional pattern mining, mining algorithms show the frequent graphs that have no relevance for such interesting vertices associated with the disease.

This master thesis takes as a main topic the significant subgraph discovery, specifically exploring the possible solutions for subgraph discovery of patterns associated to a subset of specific vertices which are already known to be interesting; this type of subgraph mining was introduced in [3] and it is the first publication that looks over subgraph patterns with a set of enriched vertices. Before talking about this specific graph mining technique and its possible applications, it is necessary to introduce the frequent graph mining approach in general terms and the most important definitions in order to have a better understanding, this is further described in Chapter 2. After introducing the current domain, a review of the existing literature on subgraph mining is explored in Chapter 3, in this section, two algorithms are selected for the further applications to the current domain of interest. Chapter 4, shows the necessary adaptations to the algorithms to work in the current domain for directed and undirected configurations and single and multiple labels parameters. Chapter 5, execute the algorithms over different datasets showing the accuracy of their results, to later describe their performance in terms of time efficiency and memory usage and finally Chapters 6 and 7, gives the conclusions and the possible of future work on this topic.

Chapter 2

Problem Settings

2.1 The Graph Domain

Nowadays, data can be found with different forms such as texts, images, video, vectors, graphs and so on. The main focus of this master thesis is the mining of data so called graphs which are structured or semi-structured data. This structures can be found in multiple domains such as social networks, biology, and any field of science which has data represented as a network.
A graph *G(V,E)* is a finite set of vertices (vertices) *V={v1,v2,...}* and a set of edges *E={e1,e2,...}* connecting such vertices [4]. Figure 2.1, illustrate the presented definition with a set of vertices, and edges connecting those vertices. However, graphs are more complex than the definition presented above. Graphs may have directionality on edges, also labels on vertices and edges, these characteristics introduce a challenge on the treatment of this kind of data.

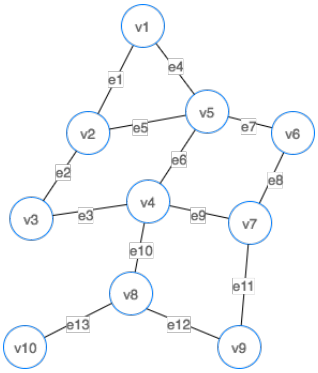


Figure 2.1: Example of Graph

Directionality of a Graph: the edges on a graph can have a direction

when connecting two vertices, a graph can be Directed or Undirected or Mixed.

- **Undirected Graphs:** In a undirected graph, the vertices are connected by edges without taking into account directionality; the concept of outgoing and incoming edges does not take any importance.
- **Directed Graphs:** In a directed graph, the edges have directionality, the vertices then have incoming and outgoing edges. A directed graph offers a richer and more complex structure, apart from the directionality, there is also the possibility of having self-edges.

Figure 2.2, shows a directed graph where the direction of the arrow indicates the origin and destination of a vertex. Graphs can also have a mixed directionality, this means that some vertices are connected with directed edges and others with undirected ones. This kind of graphs are not taken into account for the present work because they are not seen frequently and the present literature does not show algorithms for these kind of graphs.

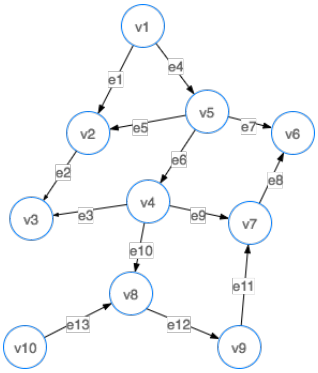


Figure 2.2: Example of Directed Graph

Labeling on graphs: A labeled graph can be represented as $G(V,E,L_V,L_E,\psi)$ [5], where V and E are the vertices and edges, L_V and L_E are the labels of the vertices and edges respectively, and ψ is a mapping function between vertices, edges and their labels. Figure 2.3, shows a graph with labeled edges $L_E\{a,e\}$ and vertices $L_V\{c,g,s,t\}$, the letters inside the circles correspond to the the vertices' labels and the ones in the middle of the connecting lines to the edges, by looking at the figure the reader can realize that the labels are finite sets for vertices and edges.
Graphs have three possibilities of classification according to their labels: unlabeled graphs, labeled graphs with maximum one label per vertex and multiple-label graphs in which the vertices can have more than one label or no label assigned to it.

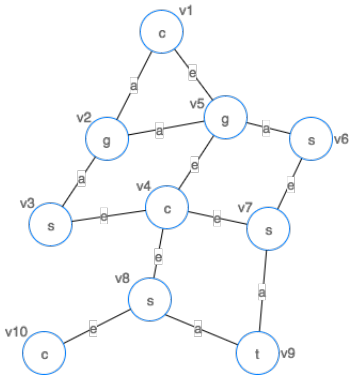


Figure 2.3: Example of Labeled Graph

2.1.1 Graph Properties

Additionally to the structure of the graph there are some properties that can be reasoned and are important to understand some of the algorithms described in Chapter 3. Some of the most important properties are described bellow [6]:

- Vertex Degree: It refers to the number of edges connected to a specific vertex.
- Shortest Path: The shortest distance (number of edges required) to connect two specific vertices.
- Network Diameter: The longest shortest path in a graph.

Apart from the properties described above, graphs can have properties such as: eccentricity centrality, closeness centrality, betweeness centrality, and clustering coefficient; these properties are not described in this work because they are not used on the algorithms presented on Chapter 3

2.2 Frequent subgraph mining

Frequent graph mining can be seen as the core of graph mining which main purpose is to identify frequent patterns of size greater than a given threshold in a dataset [5]. The number of times the subgraph is present inside a graph or group of graphs is called *support* and therefore, the number which denotes if the subgraph is interesting or not is called *support threshold*. Frequent graphs can be used in different graph operations such as classification, clustering, building of indices and other analysis. These operations can be used in multiple domains such as Biology, Chemistry, Web, Networks, Linguistics among others.

There are many different approaches to mine graph data because of the big diversity on the graphs' structure showed in Section 2.1, thus [7] presents a classification of these strategies with the three following factors:

1. Search Strategy: There are two main search strategies to find out frequent subgraphs, such are: Breadth First Search (*BFS*) and Depth First Search (*DFS*).

2. Nature of the input: The input can be defined as the number of graphs that will serve as input; if the input presents just one graph, the frequent subgraphs are searched inside this graph, when there are more than one graph in the input, then the frequent subgraphs are evaluated weather a graph has a subgraph or not, the algorithms with this kind of input are called *transaction based graph mining algorithms*, every graph in the set has a unique ID and subgraphs are count across every transaction.

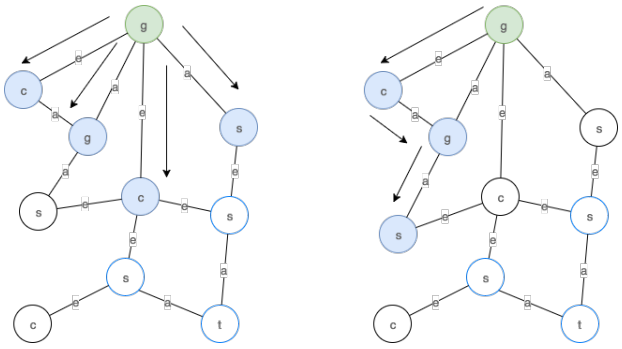
3. Completeness of the output: The output can return all the frequent subgraphs given a graph or a group of graphs, this is call a complete solution, or it can return a partial set of the results. The focus of the current problem is on complete solutions. Therefore, algorithms with a complete solutions are selected in chapter 3.

The present literature for subgraph mining algorithms presents a wide variety of approaches based in the parameters showed above. However, the present problem explores frequent subgraphs with one graph as input and a complete solution as output. The reader may feel that the amount of solutions for this configuration may get too narrowed given that the there are not many solutions specifically designed for one graph as input. In consequence, this work explores different algorithms with more than one graph as input, it analyzes the possibility to use them in the current problem, and then adapts them to get the desire output; chapter 3 introduces the algorithms that are going to be used, and chapter 4 discuses the technical implementation adding details of the necessary adaptations for the selected algorithms to work.

Literal 1 on the classification of the factors for graph mining strategies specifies two different search strategies for finding frequent subgraphs, these have a different approach to walk through the search space of a graph and are discussed bellow:

1. Breadth First Search (BFS): BFS is an algorithm to search subgraph structures in a graph, it starts by selecting a root vertex and then grows the search one level every time, this means that in every iteration, it finds the neighbors of the *current level* vertices and grows one level by adding the not visited neighbors; [8] explains how performance decreases when a graph does not fit into memory talking specifically on BFS strategies, BFS algorithms keep more objects into memory at the time than DFS algorithms. Figure 2.4a shows a small example of a BFS walk.

2. Depth First Search (DFS): DFS algorithms differ from BFS algorithms in the way they traverse the graph, instead of searching it level by level, it



(a) Breadth First Search (b) Depth First Search

Figure 2.4: Search Methods Graphs

explores a branch as far as possible and when it can not grow the pattern more, it starts backtracking to check different branches with unseen (not visited) vertices [9]. Figure 2.4b shows a small example of a DFS walk.

It is stated in [7], that algorithms which use a DFS search strategy are more optimal at finding frequent subgraphs than algorithms with a BFS search strategy; they compare the performance of two DFS algorithms and one BFS algorithm to get to that conclusion. However, the actual problem tries to find the best solution for significant subgraph mining based on interesting vertices, in consequence, the efficiency of these two strategies for this particular problem is yet to be proved, chapter 5.1 shows the results related to these strategies.

The classification of strategies proposed in [7] provides a first picture on the complexity of the frequent subgraph mining problem. However, this classification does not provide enough insight to understand the different ramifications that this problem can have, thus, an additional sub-classification based on fined parameters must be provided. This sub-classification is based on the structure that a graph can have which was introduced Section 2.1.

1. Directionality of graphs: It was stated in 2.1, that the directionality is one important characteristic of graphs, thus, the graph mining algorithms make a clear distinction if they work on a direct or indirect graph. As an example of this, [10, 11, 12, 13] only take undirected graphs as an input. Most of the researched algorithms focus on undirected graphs. However, this work focuses on directed and undirected graphs. Therefore, the selected algorithms of chapter 3 are adapted to work with the two settings if it is possible to do so.

2. Number of possible labels: The algorithms selected in chapter 3 tackles subgraphs with zero or one label on any edge or vertex, however, [3] creates an algorithm to mine significant subgraphs with interesting vertices

which handles graphs with multiple labels. Therefore, chapter 4 covers the strategies to use the selected algorithms with a multiple-label configuration.

2.2.1 Important Definitions

1. *Subgraph:* Given two graphs $G_1(V_1, E_1, L_{V1}, L_{E1}, \psi_1)$ and $G_2(V_2, E_2, L_{V2}, L_{E2}, \psi_2)$, G_1 is a subgraph of G_2 when the following conditions are satisfied[1]:

   i $V_1 \subseteq V_2$ and $\forall_v \in V_1, \psi_1(v) = \psi_2(v)$

   ii $E_1 \subseteq E_2$ and $\forall (u, v) \in E_1, \psi_1(u, v) = \psi_2(u, v)$

2. *Graph Isomorphims:* A graph $G_1(V_1, E_1, L_{V1}, L_{E1}, \psi_1)$ is isomorphic to another graph $G_2(V_2, E_2, L_{V2}, L_{E2}, \psi_2)$, if and only if a bijection $f : V_1 \to V_2$ exists such that: [1].

   i $\forall u \in V_1, \psi(u) = \psi_2(f(u))$

   ii $\forall (u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$

   iii $\forall (u, v) \in E_1, \psi_1(u, v) = \psi_2(f(u), f(v))$

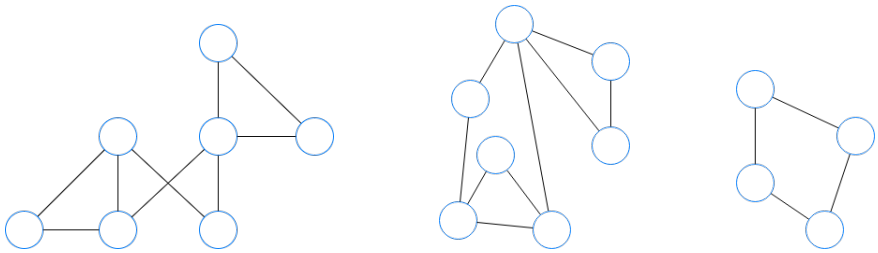
3. *Subgraph Isomorphims:* According to [1], an injective function $f : V_1 \to V_2$ is called a subgraph isomorphism from $g_1 = (V_1, E_1, L_{V1}, L_{E1}, \psi_1)$ to $g_2 = (V_2, E_2, L_{V2}, L_{E2}, \psi_2)$ if there exists a subgraph $g \subseteq g_2$ such that f is a graph isomorphism between g_1 and g.

4. Canonical labeling: Graphs present a complex structure and their representation assigning a unique code for a graph can result difficult. Canonical labels are used to compare graphs to each other, to define if a graph has been mined before, and to present results to the user. Canonical representation techniques are one of the main concerns of graph mining algorithms. The algorithms have different approaches to solve this such as: lexicographic order [14], vertex invariants [12].

Figure 2.5, illustrates the definitions of graph isomorphism and subgraph isomorphism showed above. [12] describes a special case of graph isomorphism where $g_1 = g_2$ called graph automorphism, which means that a graph g_1 has an exact mapping to a graph g_2.

2.3 Significant Subgraph mining

In statistics, the significance of a test refers to the fact that the result of a given experiment has not occurred by chance. In this case, P-value are the likelihood that the observed enrichments were the result of random chance. The lower the P-value, the less likely that the found subgraph association was the result of random chance. Therefore it is said that low P-values indicate that a found association are significant for further study [15].There are some tests available



(a) (b) (c)

Figure 2.5: Graphs (a) and (b) are isomorphic to each other (also called automorphic when $g_a = g_b$) and (c) is isomorphic to a subgraph of (a) or (b)

to measure the significance in statistics, [3] measures the significance of a given pattern using a method called Hyper-geometric Distribution and then uses the Bonferroni Correction to refine the results. These two methods are discussed bellow:

- **Hyper-geometric Distribution:** The most typical example of Hypegeometric distribution is the following: imagine you have an urn with N balls of two colors, M of them are white and N - M are black, we draw n balls uniformly without replacement. The hyper-geometric distribution tells the probability of obtaining k balls of color white given the n draws. The hyper-geometric distribution differs from a geometric one because of the *without replacement* condition; every time that we draw a ball, the probability of getting a ball of the same color is reduced because there is less balls of this color in the urn [15].

- **Bonferroni Correction:** The results of a significant subgraph mining can be a high number of subgraphs which are significant, this can result in the multiple testing problem. This problem reflects that the more inferences there are, the more erroneous inferences can occur. The Bonferroni Correction counteracts this problem by updating the significance value to a lower one, let the value be 0.05 and the number of inferences 100, the Bonferroni Corrected value is the ratio between the significance value and the number of significant results, for this case the corrected value is $0.05/100 = 0.5e^{-3}$. In consequence, less subgraphs will be significant enough providing an extra filter with finer results.

2.3.1 Significant subgraph mining of interesting Vertices

Significant subgraph mining of interesting Vertices is a new method to mine graph from a set of interesting vertices data developed by [3], this problems is specially important because with a traditional subgraph mining method it is not possible to find patterns associated with a set of interesting vertices; as an

example, in biology, the graph miner can investigate the related frequent non-know structures to a specific disease, in this example the interesting vertices are the those which are previously identified as disease. The provided example is one of the several applications of this mining method, applications can vary according to the domain of study and the properties of a graph to be investigated. This method uses the Hyper-geometric distribution to test the significance of the subgraphs and a Bonferroni Correction method to improve the final results. To clarify what significant subgraph mining with vertices of interest is, Figure 2.6, shows a graph with a set of vertices and edges, the vertices colored with light blue are the so called vertices of interest.

Figure 2.6, shows a graph with vertices $V = \{v_1, v_2, ..., v_{10}\}$, edges $E =$

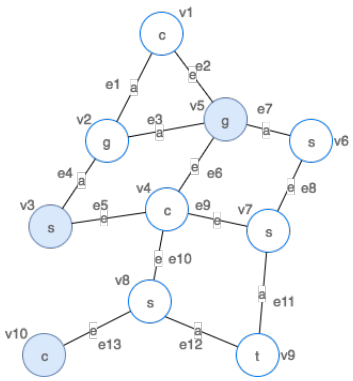


Figure 2.6: Example of Labeled Graph

$\{e_1, e_2, ..., e_{13}\}$, labels $L_E\{a, e\}$ and $L_V\{c, g, s, t\}$ and the new introduced concept of interesting vertices $I = \{v_3, v_5, v_{10}\}$. The algorithm developed by [3], focuses on searching the graph with the vertices of interest as root vertex. The root vertices are set as the starting vertex and from them the subgraphs are found. Developed algorithms do not take into account a vertex as root since they do not take into account any interesting vertex. Therefore, the algorithms selected in Chapter 3 and developed in Chapter 4 must be adapted to work with interesting vertices.

The methods for measuring the significance are kept from [3] because in order to make comparisons in Chapter 5, all the methods have to be measured in the same way.

Hyper-Geometric Distribution: The current problem explores the significance of a subgraph given a set of interesting vertices, thus, the Hyper-Geometric Distribution is measured as the probability of discovering a specific subgraph starting from these interesting vertices with the number of times this subgraph is discovered in the whole graph and the size of the search space itself.

Chapter 3

Theoretical Comparison

Graph mining techniques have proliferated over the years with many different approaches to solve different graph mining problems. Therefore, a revision of the available literature is necessary to determine the state of the art of graph mining techniques and to choose the algorithms that have the best results to use them for the current problem. Chapter 2, introduced the graphs' structure and a set of criteria to classify graph mining techniques. According to the classification criteria of chapter 2, graph mining techniques are first classified by their search strategy, their input, and their output; the current chapter explores the techniques that have the two different search strategies (BFS and DFS), one or more graphs as input and an exact result as output. As fine grained classification, the algorithms can be for a directed or undirected graphs and for single or multiple-label problems.

As it was stated before, one way to categorize subgraph mining techniques is through the search strategy, having BFS and DFS algorithms. However, [1, 5, 7, 16, 17] define that it is widely accepted to divide frequent subgraph mining techniques into two categories: (i) Apriori-based approaches and (ii) pattern-growth-based approaches; even though these two categories seem different from those described in subsection 2.2, the reality is that (i) are techniques based on a BFS search and (ii) are techniques based on DFS search. Section 3.1, explores the most popular Apriori-based algorithms while Section 3.2, focuses on those with a Pattern Growth-based approach.

3.1 Apriori-based Algorithms

Apriori-based algorithms for frequent graph mining are similar but more complex than item-set mining algorithms, because in the latter, the candidate generation is straightforward by generating just one candidate every two sub-sets. For example, suppose we have the two frequent items, *abc* and *bcd*, the candidate generation would suggest *abcd*; this shows that Apriori methods are simple

for item-sets. On the other hand, for the graph domain, this task becomes much more complex; as a starting point is important to mark out that Apriori-based approaches use a BFS to look up for frequent subgraphs, once all the subgraphs of size k have been discovered, the algorithms joins two similar but slightly different subgraphs of size k to generate candidates of size $k+1$. This process is more complex than frequent item-set mining because the two different subgraphs can generate not just one but several different candidates since there may be some ways to join two subgraphs. Algorithm 1, shows the basic approach for the Apriori-based subgraph mining.
There are several Apriori-based algorithms for mining graph data, the following

Algorithm 1: Apriori-based approach [5]

```
Input  : G = a graph data set, σ = minimum support
Output: F_1, F_2, ..., F_k set of frequent subgraphs of cardinality 1 to k
F^1 ← detect all frequent 1 subgraphs in G
k ← 2
while F_{k-1} ≠ 0 do
    F_k ← 0 C_k ← candidate − gen(F_{k-1})
    foreach candidate g ∈ C_k do
        g.count ← 0
        foreach G_i ∈ G do
            if subgraph-isomorphism (g, G_i) then
                g.count ← g.count + 1
        if g.count ≥ σ min G min ∧g ∉ F_k then
            F_k = F_k U g
    k ← k + 1
```

are the most popular ones cited and described by graph mining methods surveys such as [1, 5, 7, 16, 17].

- **AGM**: Apriori graph mining algorithm (AGM) [10] introduces an algorithm which uses a level wise approach also called BFS to find frequent subgraphs. This algorithm uses an adjacency matrix to represent graphs, it is important to point that this structure suffers of overheads when the graph is too sparse. The main idea of AGM is to generate candidates by adding one vertex at each iteration, it generates candidates of size k from subgraphs of size $k-1$, here the size of the graph refers to the number of vertices the graph has. Figure 3.1, shows two graphs joined by a core $k-1$, resulting in two candidate graphs of size k.

- **FSG**: Frequent subgraph discovery (FSG) [12] presents a candidate generation strategy which increments the size k of edges in a graph by combining two different frequent graphs of size k, in order to combine the graphs they must share a *core*. The combination of these two subgraphs of size k may result in one or many possible different subgraphs of size k + *1*. This

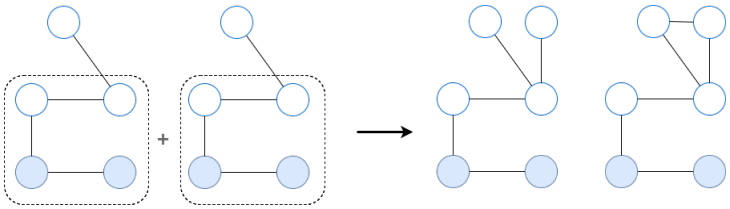


Figure 3.1: AGM [10]

method improves the AGM approach by focusing on creating candidates which add one edge instead of one vertex and also by representing the graph by an adjacency list instead of an adjacency matrix.

Figure 3.2, shows the simplest case of candidate generation; from two

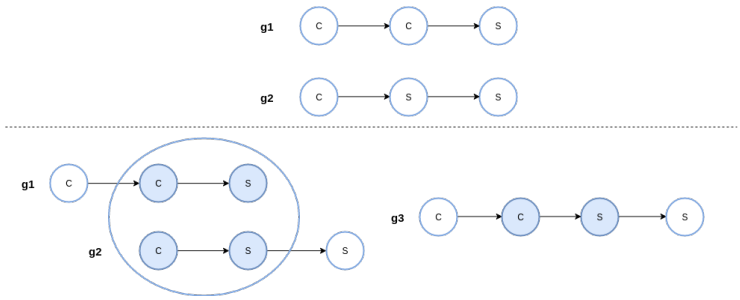


Figure 3.2: Candidate generation

frequent subgraphs of size 3, it takes as a core the edge *C-S* and generates a candidate graph of size 4. However, the candidate generation in reality given the complexity of the graphs domain, offers broader combinations than the one showed in Figure 3.2. The candidate generation can result on multiple automorphisms of a single core or multiple cores when joining two graphs.

It is stated in [14] that Apriori approaches suffer some problems because of their method to tackle the problem:

- Costly subgraph isomorphism test: Apriori algorithms have to test whether a candidate is frequent or not, thus the testing of false candidate decreases the performance.
- Costly candidate generation: Candidate generation is complicated and costly given that the algorithms have to check for subgraph isomorphism to identify common cores and the joining of two subgraphs requires to identify all the automorphisms of the subgraphs. These processes derive on complex and longer routines.

3.2 Pattern Growth Algorithms

Algorithms which are based in the pattern growth approach take as a principle that a graph G will be extended one edge at the time, this growth can add or not a new vertex. When the procedure adds one vertex the growth is said to be in a forward direction otherwise in a backward direction.
Pattern growth algorithms avoid costly candidate generation by growing the pattern incrementally and finding the patterns frequency in the same step. Algorithm 2 illustrates in a general way how pattern growth approaches increase their edges recursively until no more frequent graph can be found.
As it can seen, algorithm 2 is not efficient given that the same graph can be

Algorithm 2: PatterGrowth(g, D, min_support, S) [1]

Input : A frequent graph g,a graph dataset D, and min_support
Output: A frequent substructure set S
if $g \in S$ **then**
  return;
else
  insert g to S;
scan D once, find all the edges e such that g can be extended to $g \diamond_x e$;
foreach *frequent* $g \diamond_x e$ **do**
  Call PatternGrowth($g \diamond_x e$, D, min_support, S);

discovered many times because of isomorphism. The repeating discovery makes this algorithm simple but very inefficient [1]. The following approaches take as a base algorithm 2 and improve the process by duplicate generation avoidance strategies.

- **gSpan** [14]: gSpan is a pattern discovery algorithm that is designed to traverse different graphs and discover the frequent patterns in those graphs given a support threshold and a maximum number of vertices. The advantage of this algorithm is that it takes a deep-first search approach and that it goes through the graph with a right-most path extension strategy with a lexicographic canonical labeling which allows it to increment the pattern size avoiding duplicate graphs.

- **Gaston** [13]: This algorithm is based in a divide and conquer principle, it divides the mining process into different phases efficiently; this phases mainly focuses on paths, free trees and cyclic graphs; it takes into consideration that paths and free trees are much simpler structures and therefore mining these at first gives a quickstart to the process. A path is a graph in which two vertices have a degree of 1 and the rest a degree of 2; a free tree is a graph which does not contain any cycle and finally a cyclic graph is a graph which there are one of more edges which come back to an already visited vertex.

Table 3.1: Algorithms by classification criteria

Algorithm	Search Strategy	Input	Output	Labels	Directionality
AGM	BFS	Multi-graph	Exact	Single	Undirected
FSG	BFS	Multi-graph	Exact	Single	Undirected
gSpan	DFS	Multi-graph	Exact	Single	Undirected
Gaston	DFS	Multi-graph	Exact	Single	Undirected

3.3 Algorithm Selection

There are different approaches to solve graph data mining and Sections 3.1 and 3.2 shortly described the most known algorithms for BFS and DFS approaches respectively. This section focuses on choosing the algorithms that later will be used on Chapter 4. The selection takes into account one algorithm for each approach because one of the goals for this work is to decide on an empirical manner which technique best suites the current problem of graph mining with a set of interesting vertices.

- **Selection of the Apriori algorithm:** Section 3.1, talks about AGM and FSG algorithms; the first one has a candidate generation based on the increment of one vertex every iteration while the second one grows the graph by one edge at the time. By purely talking about the growth of the algorithms on their candidate generation, there is no clear way to define which algorithm is best for the current problem. However, FSG uses more sophisticated techniques to store and compare the graphs. AGM uses an adjacency matrix while FSG adjacency lists, the second one is proven to be more efficient by minimizing storage and computation. It is also stated on [12] that this technique in most of the experiments outperforms drastically the performance of AGM, it is also said that it incorporates various optimizations on the candidate generation and counting that allows it to perform better on larger datasets. In consequence, FSG is selected for the empirical algorithm evaluation. Section 3.3.1 discusses the FSG algorithm in length, and details its main concepts.

  The selected Apriori algorithm is FSG given that introduces several optimizations that makes it perform better on larger datasets and experiments show better performance than AGM.

- **Selection of the Pattern Growth algorithm:** Section 3.2 introduces two algorithms which use the DFS method to look over the graph, gSpan algorithm uses a Right-most path extension to grow the graphs and a minimum DFS code condition to avoid looking for graphs which not have a minimum DFS code. On the other hand, Gaston algorithm provides a

divide a conquer scenario, solving graphs from lower to higher complexity. Gaston first looks for paths, then for free trees and finally for cyclic graphs, giving a quickstart to the algorithm. These two algorithms have different approaches and a comparison from a technical point of view is complex. However, in [13] they compare the methods showing that both have advantages and disadvantages, gSpan is better when pruning backward edges while Gaston allows much more pre-pruning for free trees. In [13] it is concluded that the Gaston algorithm is promising in cases where the number of cycles and labels are not too large. Even though, it proves to be very efficient on the experiments, it is also true that it is graph dependent and performs worse on graphs with large number of labels and cycles. Therefore, for this work, gSpan is selected, given that there is non proof that its performance will decrease on complex graphs and that also [14] describes that in their experiments they improved FSG by one order of magnitude. Also [18] says that using Gaston algorithm in a directed configuration would require major changes, this reason also makes more suitable to choose gSpan as the working algorithm for this work.

The selected Pattern Growth algorithm is gSpan because it is an algorithm that improves FSG by one order of magnitude and even though it has disadvantages against Gaston, there is non proof that its performance is graph dependant.

The following subsections 3.3.1 and 3.3.2 describes the behavior and operation of the algorithms in detail.

3.3.1 FSG Algorithm

FSG is an algorithm designed to find frequent subgraphs inside a group of graphs with a single label, undirected configuration providing an exact solution. This algorithm creates subgraphs of size $k + 1$ based on graphs of size k which are similar but slightly different. This increase of size is done by adding one edge at every iteration in contrast to AGM which adds one vertex.
Since this is an Apriori algorithm, the general algorithm 3 is similar to Algorithm 1 with slight changes. This algorithm first generates the frequent subgraph with one and two edges to later generate the candidates by comparing two subgraphs that share a common core, finally it counts the occurrences of the given candidate.
This algorithm improves AGM candidate generation by adding a method called vertex invariants to create a canonical label of the graphs, this is the way the algorithm controls the isomorphisms over different graphs to not evaluate the same graph twice.

3.3.1.1 Canonical labeling

Finding a way to represent a graph is a complex task in graph mining because a single graph have many different ways to be represented. The task of find-

Algorithm 3: FSG(D, σ) (Frequent Subgraph) [19]

```
F^1 ← detect all frequent 1-subgraphs in D
F^2 ← detect all frequent 2-subgraphs in D
k ← 3
while F^{k-1} ≠ 0 do
    C^k ← fsg_gen(F^{k-1})
    foreach candidate g^k ∈ C^k do
        g^k.count ← 0
        foreach transaction t ∈ D do
            if candidate g^k is included in transaction t then
                g^k.count ← g^k.count + 1
    F^k ← {g^k ∈ C^k | g^k.count ≥ σ | D |}
    k ← k + 1
return F^1, F^2, ..., F^{k-2}
```

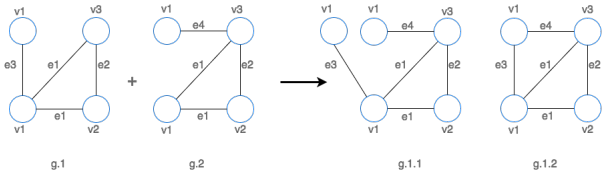
ing a unique way to represent a graph is called canonical labeling [20] and this process is not known to be P or NP-complete. In a graph is difficult to find a lexicographic order as in frequent item-set mining. However, there are some methods to do so, one can be seen in the gSpan algorithm described in section 3.3.2. This algorithm does not present a lexicographic order by itself to measure the canonical label but a representation of the graph in an adjacency matrix re-ordered by a method called Vertex Invariants.

Vertex Invariant: This method introduced by [21] reorders a matrix to form a unique representation. However, it is adapted in [12] to work with the specific domain. The approach creates partitions based on vertex degrees and the vertex' labels to finally make permutations inside the partitions to get the lowest possible representation; this is further explained by the next steps:

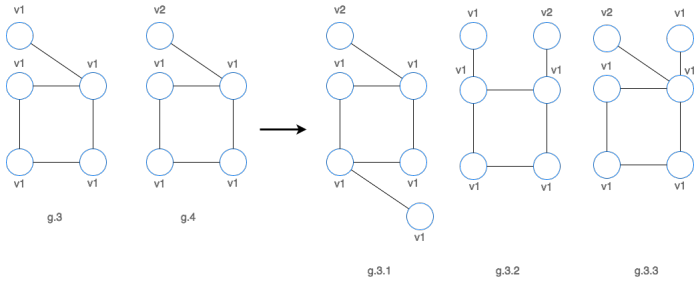
1. It calculates the degree of the vertices and partition them according to their degrees. The vertices are reordered based on the degree, vertices with a degree of 1 will be first, followed by higher degrees.

2. It creates new partitions inside the partitions created on the previous step; this partitions are based on the labels of the vertices and ordered with a lexicographic order.

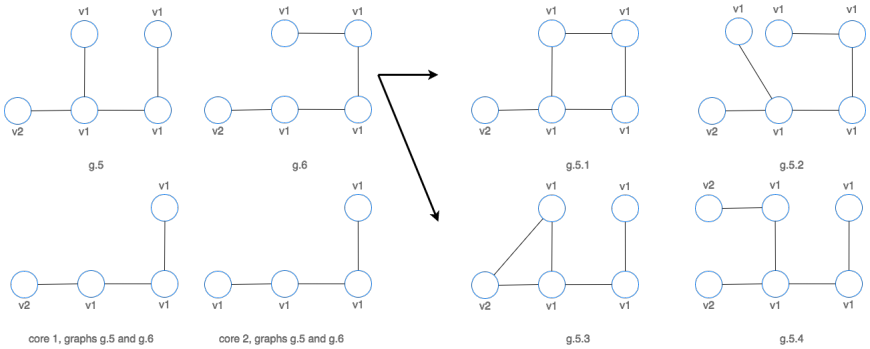
3. Given that this approach also works with labeled edges, it makes permutations inside the label partitions until it gets the edges with a lexicographic order as well. This is the last step and after calculating all the permutations the matrix is said to be canonical, giving a unique code to any graph.



(a) By vertex labeling



(b) By multiple automorphisms of a single core



(c) By multiple cores

Figure 3.3: Three different cases of candidate joining [19]

3.3.1.2 Candidate Generation

As stated before the size of the graph in the candidate generation is increased by one edge every iteration. Candidates of size $k+1$ are generated from a set of $k-subgraphs$. It is said in [19] that in order to select two graphs for joining, these must share a core, this core is a graph of size $k-1$.

Figure 3.3, shows the complexity of joining different graphs, illustrating three different scenarios where a pair of graphs of size k create a set of candidates of size $k+1$. Figure 3.3a, shows an example when two subgraphs differ in a vertex with the same label, Figure 3.3b, shows an example when a core has multiple isomorphisms and each one of them creates a different candidate of size $k+1$ and finally Figure 3.3c, shows an example when two graphs share more than one core in common, resulting in several candidates of size $k+1$.

Algorithm 4, shows the candidate generation approach; first it analyzes all the possible pair of graphs of size k and extracts an edge from them to check if they share a common core, if the case is true, it calls the join algorithm which generates the candidates; for every candidate it tests the downward closure property, if the property holds, it adds the candidate to the set of frequent graphs. The join algorithm is in charge of joining two graphs based on their

Algorithm 4: fsg_gen(D, F^k) (Candidate Generation) [19]

```
C^{k+1} ← 0
foreach pair of g_i^k, g_j^k ∈ F^k, i ≤ j such that cl(g_i^k) ≤ cl(g_j^k) do
    foreach edge e ∈ g_i^k do
        {create a (k-1) subgraph of g_i^k by removing an edge e}
        g_i^{k-1} ← g_i^k − e
        if g_i^{k+1} is included in g_j^k then
            {g_i^k and g_j^k share the same core}
            T^{k+1} ←fsg_join(g_i^k, g_j^k)
            foreach g_j^{k+1} ∈ T^{k+1} do
                {test if the downward closure property holds for g_j^{k+1}}
                flag ← true
                foreach edge f_l ∈ g_j^{k+1} do
                    h_l^k ← g_j^{k+1} − f_l
                    if h_l^k is connected and h_l^k ∉ F^k then
                        flag ← false
                        break
                if flag = true then
                    C^{k+1} ← C^{k+1} ∪ {g^{k+1}}
return C^{k+1}
```

core. Algorithm 5, shows this process; first it detects all the automorphisms of the shared core and secondly it detects the edges that are not present on the core from the two analyzed graphs; once the lacking edges from the core are detected, it generates all the possible candidates of size $k + 1$.

3.3.1.3 Frequency Counting

Calculating the frequency of each candidate in the entire graph set is an expensive operation and not feasible for large dataset. Therefore, FSG uses heuristics to short this operation as much as possible and improve the efficiency of the algorithm. For every found subgraph it saves a list of the graphs supporting the given subgraph, when the candidates of size $k + 1$ are generated, it does not look them up immediately in the dataset; first it calculates the intersection of the supporting transaction IDs of the graphs of size k that generated those candidates. If the intersection has a support bigger than the support threshold,

Algorithm 5: fsg_join(g_1^k, g_2^k, h^{k+1}) (Frequent Subgraph) [19]

```
M ← detect all automorphisms of h^{k-1}
{determine an edge e_1 ∈ g_1^k that does not appear in h^{k-1}}
e_1 ← NULL
foreach edge e_i ∈ g_1^k do
    if e_i ∉ h^{k+1} then
        e_1 ← e_i
        break
{determine an edge e_2 ∈ g_2^k that does not appear in h^{k-1}}
e_2 ← NULL
foreach edge e_i ∈ g_2^k do
    if e_i ∉ h^{k+1} then
        e_2 ← e_i
        break
G ← generate all possible graphs of size k + 1 from g_1^k and g_2^k, using M
```

then the algorithm proceeds to look for candidates only on the graphs which are in the transaction IDs. This method prunes the search space effectively for a given candidate limiting the search to only a few transactions.

3.3.2 gSpan Algorithm

gSpan is a graph mining algorithm created to optimize the mining process, at the time gSpan was created, the most efficient algorithm was FSG. However, gSpan's creators saw some faults on Apriori algorithms such as FSG that could be drastically improved; This algorithm was born to solve problems such as costly subgraph isomorphism test and costly candidate generation, the algorithm created in [14] solves this problem by implementing an algorithm without candidate generation and a DFS traversal approach.

According to the classification presented in Chapter 2 section 2.2, gSpan is an algorithm which uses DFS as search strategy, takes as input a set of graphs, gives an exact solution, and works for undirected and labeled graphs.
The main contributions of this algorithms are: the creation of a canonical labeling system that supports DFS called *DFS lexicographic order*, the control on the generation of subgraphs through a *minimum DFS code* and the combination of the subgraph isomorphism test and frequent subgraph growth into one procedure.

3.3.2.1 Lexicographic Order

Lexicographic order is the method that gSpan uses to compute the graph, establishing rules on how to grow a subgraph and also to compute a canonical representation. gSpan [14] presents a number of rules to construct a DFS code,

defining how the new vertices must be added, the restrictions these vertices have while being added, the order of this addition and finally the canonical representation also called *minimum DFS code*; these rules are described bellow and to facilitate their understanding, they are described based on Figure 3.4, which is a simple graph with five connected vertices and has labels just on their vertices.

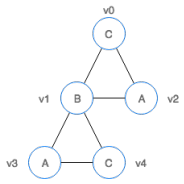


Figure 3.4: Graph example

3.3.2.1.1 DFS Code

The construction of a DFS Code starts from a single vertex, from this vertex a forward edge connected to the edge is added, after this, the algorithm looks for all the backward edges that connects this vertex to all the old vertices in the original DFS code. This procedure is repeated until there are no more edges to include or until the graph reaches the maximum vertex number. In order to clarify this procedure the definitions of froward and backward edge are defined bellow:

- **Forward Edge:** In the DFS Code a forward edge is an edge which was created by growing the graph to a vertex not visited before. As an example, when we start the construction of the DFS Code of Figure 3.4 from vertex $v0$, there are no edges on the code yet. Therefore the edge going from $v0$ to $v1$ is a forward edge given that $v1$ was not visited before.
- **Backward Edge:** A backward edge is an edge from an already visited vertex to another already visited vertex, continuing with the example of forward edges, from Figure 3.4, we have already visited $v0, V1$, and $v2$ with the code $((v0, v1), (v1, v2))$, we can see that there is an edge from $v2$ to $v0$, this edge is a backward edge given that the two vertices have been already visited.

For the graph in Figure 3.4, if we start to construct the DFS Code from the vertex $v0$ we get the following order: $((v0, v1), (v1, V3), (v3, v4), (v4, v1), (v1, v2), (v2, v0))$. At this point, the reader may have realized that there are several valid ways to represent a graph with a DFS Code. Table 3.2 shows three of the several possible representations of the graph on Figure 3.4.

The DFS Code generation is the core of the so called Right-most path approach used by gSpan. Figure 3.5, illustrates the order of the growth of the edges on a graph. The Figure shows that the algorithm first tries to grow the

Table 3.2: Three DFS Codes for Figure 3.4

edge no.	rep 1	rep 2	rep 3
0	$(0, 1, C, B)$	$(0, 1, C, A)$	$(0, 1, C, A)$
1	$(1, 2, B, A)$	$(1, 2, A, B)$	$(1, 2, A, B)$
2	$(2, 0, A, C)$	$(2, 0, B, C)$	$(2, 0, B, C)$
3	$(1, 3, B, A)$	$(2, 3, B, A)$	$(2, 3, B, C)$
4	$(3, 4, A, C)$	$(3, 4, A, C)$	$(3, 4, C, A)$
5	$(4, 1, C, B)$	$(4, 2, C, B)$	$(4, 2, A, B)$

graph with a backward edge, if this is not possible it tries to grow it with a forward edge and this is also not possible, it backtracks and tries to keep growing the graph until this is not possible. This subsection introduces the Right

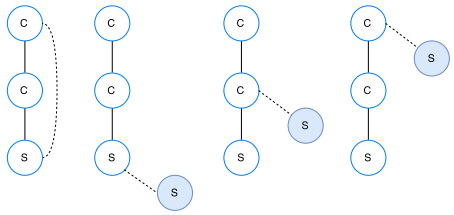


Figure 3.5: Right most path growing rule

Most Path Approach. However, not all the rules to grow a graph are seen here, subsection 3.3.2.1.5, details the rest of the necessary rules for the Right Most Path approach of gSpan.

3.3.2.1.2 Neighbor Restriction

When checking the rules to create DFS Codes, a graph can have several valid representations. Because of this, a set of rules called *Neighborhood Restriction* limits the ways of creating a DFS Code. The following are the rules summarized by [22] which tell if an edge is valid or not:

- When the *current edge* is a forward edge:
  - When the *next edge* is a forward edge:
    - If the first vertex of the *next edge* is less than or equal to the second vertex of the *current edge*,
    - And the second vertex of the *next edge* is equal to the second vertex of the *current edgee* plus one, then the *next edge* is considered valid.
    - Otherwise the *next edge* is considered invalid.
  - When the *next edge* is a backward edge:

    - If the first vertex of the *next edge* is equal to the second vertex of the *current edge*,
    - And the second vertex of the *next edge* is less than the first vertex of the *current edge*, then the *next edge* is considered valid.
    - Otherwise the *next edge* is considered invalid.

- When the *current edge* is a backward edge:
  - When the *next edge* is a forward edge:
    - If the first vertex of the *next edge* is less than or equal to the 1st vertex of the *current edge*,
    - And the second vertex of the *next edge* is equal to the first vertex of the *current edge* plus one, then the *next edge* is considered valid.
    - Otherwise the *next edge* is considered invalid.
  - When the *next edge* is a backward edge:
    - If the first vertex of the *next edge* is equal to the first vertex of the *current edge*,
    - And the second vertex of the *current edge* is less than the second vertex of the *next edge*, then the *next edge* is considered valid.
    - Otherwise the *next edge* is considered invalid.

3.3.2.1.3 DFS Lexicographic Order

The rules introduced by the generation of the DFS Code and the Neighbor Restriction constraint reduces significantly the amount of DFS Code that a graph can have. However, by looking at the graph on Figure 3.4, we still can find a great number of DFS Codes that are correct according to these rules, specially the DFS Code can start being constructed from different vertices not only from $v0$. Therefore additional constraints are set by taking into account the labels of the vertices to get a DFS Code with a Lexicographic Order. If any of the following rules summarized by [22] apply, we can say that a given edge e_1 is less lexicographic speaking than a given edge e_2:

1. e_1 is a backward edge and e_2 is a forward edge.
2. e_1 is a backward edge, e_2 is a backward edge, and the second vertex of e_1 is less than the second vertex of e_2
3. e_1 is a forward edge, e_2 is a forward edge and the first vertex of e_2 is less than the first vertex of e_1.
4. e_1 is a forward edge, e_2 is a forward edge, the first vertex of e_1 is equal to the first vertex of e_2, and the label of the first vertex of e_1 is less than the label for the first vertex of e_2.
5. e_1 is a forward edge, e_2 is a forward edge, the first vertex of e_1 is equal to the first vertex of e_2, and the label of the second vertex of e_1 is less than the label for the second vertex of e_2.

3.3.2.1.4 Minimum DFS Code

gSpan keeps a DFS tree with the different representations of a graph. Although the refinements made in subsections 3.3.2.1.1, 3.3.2.1.2, 3.3.2.1.3, there are still several ways to build a graph depending on the starting point and the growing of edges. A graph G has one minimum DFS Code which is said to be the canonical representation of the graph.
Given the two graphs, we can compare them by taking their minimums, the graph G^1 is said to be the minimum if and only if $min(G_1) < min(G_2)$. Also it is important to point that two graphs are isomorphic to each other if their minimums are the same $min(g_1) = min(g_2)$.
Algorithm 7, in its first line shows the *min* condition, thus, we can say that the problem of finding frequent patterns in gSpan is solve by finding the minimum DFS representation.

3.3.2.1.5 DFS Code's Parent and Child

gSpan defines the concept of parent and child vertex in order to grow the edges of the subgraphs. Given a DFS Code $\alpha = \{a_0, a_1, ..., a_m\}$, any valid DFS code $\beta = \{a_0, a_1, ..., a_m, b\}$ is called an $\alpha's$ child, and α is called $\beta's$ parent.

This restriction mainly points that the edge b cannot grow in an arbitrary position, given the restriction introduced on subsection 3.3.2.1.2. Therefore, this edge b has to grow from a vertex on the rightmost path. figure 3.6, shows a subgraph growing from the rightmost path, the reader can see the rightmost path as the vertices colored with light blue. Figure 3.6 literal (a), illustrates that first a backward edge is tried as said in subsection 3.3.2.1.1 (a backward edge can only grow from the rightmost vertex); if there is not such backward edge, then the algorithm tries to grow a forward edge from the rightmost vertex, Figure 3.6 literal (b), illustrates this growth. After trying these two edges, the algorithm in an orderly way tries to grow forward edges from the rest of the vertices in the rightmost path, Figure 3.6 literals (c) and (d), illustrates this behavior.

The enumeration is done by the lexicographic order shown in subsection 3.3.2.1.3. Figure 3.6 should be in the order 3.6(a), 3.6(b), 3.6(c), 3.6(d) with 3.6(c.1) and 3.6(c.2) being children of 3.6(c).

3.3.2.2 The gSpan Approach

Algorithm 6 shows the pseudo-code of the initialization of the algorithm, it first generates and counts all the 1-edge subgraphs and orders them with the DFS lexicographic order. The 1-edge codes that do not have the necessary support are pruned from the graphs, thus, gSpan optimizes the search space from the beginning of the algorithm.

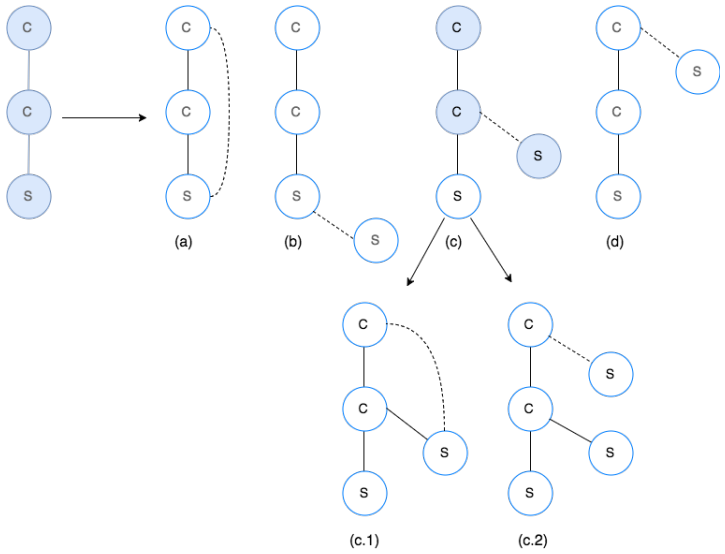


Figure 3.6: DFS Code/Frequent Pattern Growing

For every edge already sorted, it calls the Subgraph mining algorithm shown in Algorithm 7 which starts the growing process. For each subgraph, it calls the growing procedure, and after this finishes, the algorithm shrinks the edges that have already been found in the line with $GS \leftarrow GS - e$. For our example, the algorithm would find all the frequent subgraphs starting with $(0, 1, A, A)$, so this edge can be pruned from the graph dataset. This heuristic is done to optimize the search of the following 1-edge subgraphs by reducing the search space; the algorithm makes sure that all the graphs with the edge $(0, 1, A, A)$ are found and given the lexicographic order no other different pattern with this edge would be present in the graph dataset. The algorithm finishes when all frequent 1-edge subgraphs are covered.
The algorithm checks in every recursive call the minimum DFS condition, if the incoming graph is not minimum, it is not taken into account in the process. This condition is a fundamental improvement from the algorithm of pattern growth approach presented in 2 given that it minimizes the amount of patterns which are tested. It is also testing at the same time for subgraph isomorphism given that it looks for graphs which are minimum and does not look for duplicate graphs. This pruning of the search space does not affect the completeness of the result.

After checking the minimum condition, the algorithm generate all the potential children of the subgraph by growing one edge on the rightmost path, it enumerates the subgraph and if this subgraph count is equal or greater than the support threshold continues, otherwise it goes on by testing the children with one edge growth, this calls algorithm 7 recursively until there are no more edges

Algorithm 6: GraphSet_Projection(GS,S) [11]

```
sort labels of the vertices and edges in GS by their frequency;
remove infrequent vertices and edges;
relabel the remaining vertices and edges in descending frequency;
S^1 ← all frequent 1-edge graphs in GS;
sort S^1 in DFS lexicographic order;
S ← S^1;
foreach edge e_i ∈ S^1 do
    initialize s with e, set s.GS = {g | ∀ ∈ GS, e ∈ E(g)};(only graph ID
     is recorded)
    Subgraph_Mining(GS, S, s);
    GS ← GS − e;
    if | GS |< minSup then
        break;
```

Algorithm 7: Subgraph_Mining(GS, S, s) [11]

```
if s ≠ min(s) then
    return;
S ← S ∪ {s};
generate all s' potential children with one edge growth;
Enumerate(s);
foreach c, c is s' child do
    if support(c) ≥ minSup then
        s ← c;
        Subgraph_Mining(GS, S, s)
```

to grow or the maximum number or vertices has been reached.

3.4 Theoretical Comparison

It can be seen from section 3.3 that there are different approaches to solve the graph mining problem; a small comparison of the most popular algorithms is presented in Table 3.1 in order to choose from the existing algorithms, the ones to fully develop in this work, also a detailed specification of the two selected algorithms is presented in subsections 3.3.1 and 3.3.2. This section focuses on generalize some of the techniques from the two selected algorithms to offer the reader a more technical comparison.

Table 3.3, shows some comparisons of technical aspects between FSG and gSpan; this points are briefly explaining bellow:

- **Search Strategy**: The search strategy is well differentiated between

Table 3.3: FSG vs gSpan Technical Comparison

Algorithm	**Search Strategy**	**Graph Representation**	**Candidate Generation**	**Canonical Labeling**	**Growth Method**
FSG	BFS	Adjacency List	Core identification	Vertex Invariant	By candidates join
gSpan	DFS	Adjacency List	None	Minimum DFS Code	Rightmost path
	Isomorphism tests	**Duplicate Check**	**Support Computation**	**Enumeration**	
FSG	With candidates	Duplicates check before hand	Transaction IDs	Transaction IDs intersection	
gSpan	On the fly	No duplicates	Transaction IDs	Projection of DFS Codes	

gSpan and FSG while the first one takes a DFS approach, the second takes a BFS one.

- **Graph Representation**: Both algorithms represents the graphs with an adjacency list. However, for the storage the subgraphs, gSpan uses a DFS code and FSG an adjacency list. It is also important to note here that FSG, changes the adjacency list of the subgraph to an adjacency matrix, after the calculation, the graph is returned to an adjacency list form.

- **Candidate Generation**: FSG has a candidate generation with a 1-edge growing approach while gSpan has not candidate generation; gSpan avoids this by using a DFS approach.

- **Canonical labeling**: For calculating the canonical label, FSG use the Vertex Invariants method which makes a unique representation of the graph in a matrix, while gSpan works with DFS Lexicographic Order which introduces rules on the construction of DFS Codes and takes a minimum representation.

- **Growth Method**: In order to grow the subgraphs FSG uses a candidate generation approach from two subgraphs of size k which share the same core, while gSpan grows the graph by a rightmost extension approach.

- **Isomorphism tests**: FSG calculates the canonical label of the candidates and look up if they have been searched before, while gSpan checks the isomorphic graphs on the fly by removing those which do not have a minimum DFS Code.

- **Duplicate Checks**: FSG checks duplicate graphs after the candidate generation, with the canonical label it checks if there is a graph with the same canonical label already found, gSpan does not allow for duplicate candidates given that graphs with not minimum DFS Code are checked.
- **Support Computation**: FSG and gSpan save time when calculating the support of the graphs given that they both keep track of a transaction list with the graphs that support a given subgraph.
- **Enumeration**: FSG uses the intersection of the transaction list of the two graphs which generated a candidate and enumerates them by looking up with the BFS approach, gSpan keeps projections of all the graphs' subgraphs that supported a given DFS Code.

Chapter 4

Proof of Concept

The main algorithms for subgraph mining have been theoretically discussed and compared in chapter 3, for a deeper insight on how this algorithms work given the problem of subgraph mining of interesting vertices in a single graph, an implementation and adaptation of the selected algorithms to the actual problem is necessary. Therefore, sections 4.2 and 4.3 discusses the the implementations of the algorithms and details the adaptations to make the algorithms work in the current domain. These implementations are done in order to provide a platform for experimentation of Chapter 5.1, which allows to discover which is the best approach for the current domain. Because this work is based on the problem of significant subgraph mining with a set of interesting vertices described by [3], a brief introduction to this approach is necessary to further understand the differences between this and the other mining algorithms and why the implementation and experimentation of other approaches. Section 4.1 briefly introduces this algorithm.

4.1 Base Algorithm

The problem of significant subgraph mining with interesting vertices was first explored by this algorithm and therefore became the base for comparisons and validation checks over the other algorithms. The algorithm is constructed as a proof of concept. However, while the theoretical basis of the interestingness measure is solid, the search strategy is sub-optimal resulting in long runtimes on proof-of-concept implementations.

4.1.1 Candidate Generation

The algorithm takes a basic approach in the candidate generation step given that with the set of possible labels, it generates all the possible combinations of candidates. The algorithm offers the possibility of exploring directed and

undirected graphs and therefore the candidate generation differs in one case or another.

- Direct graph: A self, forward and backward motif are created in the first iteration with the combination of all the possible labels, these one edge subgraphs candidates give a start to begin the matching process in the graph.

    1. Self edge: 1 Lbl(1) - 1 Lbl(1), it generates a self edge, this is a vertex which is connected with itself.
    2. Forward motif: 1 Lbl(1) - 2 Lbl(2), it represents an edge starting from a vertex 1 with label Lbl(1) to a vertex 2 with label Lbl(2). This represents an outgoing edge which starts from the origin.
    3. Backward motif: 2 Lbl(1) - 1 Lbl(2), it represents an edge starting from a vertex 2 with label Lbl(1) to a vertex 1 with label Lbl(2). This represents an incoming edge from a vertex 2 to the origin.

- Indirect graph: As in directed graphs, it combines the possible labels into a set of edges. However, in an undirected configuration the directionality is not important, it creates just a forward motif.

    1. Forward Motif: 1 Lbl(1) - 2 Lbl(2), it is the only starting point for an undirected graph; all the subgraphs are generated from the origin vertex to the other possible vertices.

Algorithm 8, shows the start of the candidates generation for a directed graph. The candidate generation of this base approach is seen as a proof of concept

Algorithm 8: Subgraph_Miner(G, N, L) [3]

```
ℙ = 0, ℙ − 𝕊 = 0, 𝔼 = 0
Calculate Freq_minS
foreach label l_1 of L do
    𝔼 ← 𝔼 ∪ (1, l_1, 1, l_1)
    foreach label l_2 of L do
        𝔼 ← 𝔼 ∪ (1, l_1, 1, l_2)
        𝔼 ← 𝔼 ∪ (2, l_1, 1, l_2)
foreach single edge patter P of 𝔼 do
    Subgraph_Search(G,P,N,E_MAX,P_MAX,Freq_minS)
```

given that that the amount of possibilities that it generates in each step is large, and increases the consumption of resources.

4.1.2 Search Strategy

The starting vertex with ID 1 matches first with the vertices of interest, if the support over the interesting vertices is equal or greater of the given support threshold, then the algorithm continues to find the subgraphs through the rest of the graph in order to find the total support of the pattern. It is important to note that the algorithm keeps a mapping of the vertices of the candidate and the graph to keep track of how the matching is being done. In this specific case of graph mining, the search space is first set to the interesting vertices and if and only if the pattern's support reaches the support threshold, the algorithm continues to mine the rest of the graph.
Algorithm 9, shows the recursive procedure to measure the support of each

Algorithm 9: Subgraph_Search(G,P,N,E_{MAX},P_{MAX},$Freq_{minS}$) [3]

```
Calculate Freq_S for pattern P and vertices N
if Freq_S < Freq_minS then
    return
Calculate Freq_T for pattern P in graph G
Calculate P(Freq_S ≥ X)
if P(Freq_S ≥ X) < PMAX then
    ℙ ← ℙ ∪ P
if size(P) < E_MAX then
    foreach child pattern C of P do
        S ← Canonical(C)
        if S ∉ 𝕊 then
            𝕊 ← 𝕊 ∪ S
            Subgraph_Search(G,S,N,E_MAX,P_MAX,Freq_minS)
```

candidate and based on its canonical representation decide if the graph have been searched before.

4.1.3 Motif Optimization

The motif optimization calculates the canonical label of a subgraph in order to avoid searching the same graph twice. Given that a graph can have many different ways to be represented, this method creates a single representation of a graph based on the distance between the vertices and the origin, the number of their incoming edges, the number of their outgoing edges and finally the lexicographic order the their labels.
Figure 4.1 shows a graph with the mentioned parameters: distance to the origin (blue), number of outgoing edges (red), number of incoming edges (green). At the moment that the motif is reconstructed all these parameters are taken into account to re-order and re-structure the motif. For a better understandability of the reader, the Optimized graph of figure 4.1 shows only the reordering of the motif without changing the vertices' IDs. Although, the real algorithm changes

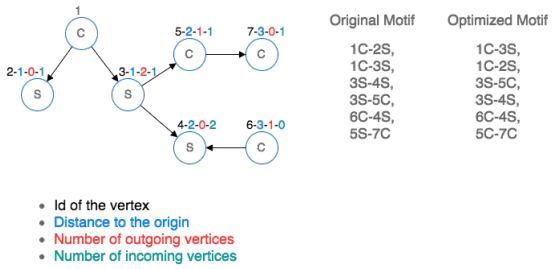


Figure 4.1: Motif optimization

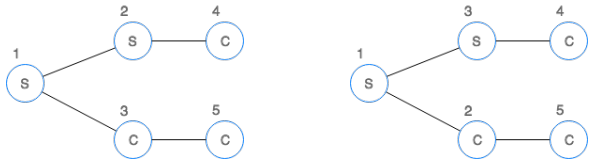
the IDs of the labels of the candidate to represent it in a better way. This approach is viable to discover isomorphic graphs for most of the subgraphs in a directed or undirected setting. However, this can still fail in the case of a tie in all the mentioned parameters between two or more vertices; in other words, this happens when two or more vertices have the same distance to the origin, number of outgoing and incoming edges and the same label; in this cases the algorithm would not be able to find only one solution. These ties are rare but still can affect the performance and results of the algorithm. Chapter 5.1, in its section 5.1.3 shows a tool to identify the duplicates results on a subgraph mining procedure, giving the fails of this canonical labeling strategy.

4.2 FSG algorithm implementation

FSG is an algorithm which uses a breath first search approach to traverse a graph, it is mainly designed for a dataset with multiple graphs with an undirected, single label configuration. Therefore, some changes must be done to first translate this algorithm to make it work in a single graph and later to add directed and multi-label support.

4.2.1 Canonical Representation

FSG uses a simple algorithm called vertex invariants in order to get a unique representation of a given graph; this canonical labeling method with the respective changes for the current domain is fully described bellow:
Vertex Invariant: The vertex invariant approach represents a graph in a matrix and reorders the vertices with a set of rules to obtain the canonical label of a subgraph, as it is shown in sub-section 3.3.1.1. However, this approach can not be applied as it is when looking for frequent subgraphs associated with interesting vertices. Some modifications to the approach have to be made in order to be able to use this canonical representation. The two new rules to construct a canonical matrix:



(a) Isomorphic graph example 1

(b) Isomorphic graph example 2

Figure 4.2: Example of a different construction of a subgraph

1. **Vertex Origin:** The original implementation of vertex invariants is not applicable because the origin vertex can be changed in the re-ordering of the vertices. As a consequence, this approach places the origin vertex in the first position of the matrix and keeps it in the same position during the whole process. Example 4.2.1 shows in*M1 and N1* this behavior, the origin vertex is fixed to remain in the same position during the entire process.

2. **Distances to origin:** This rule reorders the vertices which are in the same degree partition and have the same label. This rule is applied because the approach could give more than one possible representation for the same subgraph and, it is calculated by taking the mentioned vertices and counting from left to right, from the vertex zero in the matrix to the first value that shows a presence of an edge. In other words, it measures the distance from the origin to the first edge presented for each vertex, the reorder is done from the vertex which presents the minimal distance to the origin vertex to the largest distance. In case that the vertices have the same distance to the origin in the matrix the rule is not applied because the canonical representation would be the same. Example 4.2.1, shows in *M4 and N4* how the vertices 4 and 5 which have the same degree and label are exchanged given that 4 has a lower distance to the origin than 5.

Figure 4.2, shows the same subgraph constructed in two similar but different ways, 4.2a and 4.2b are isomorphic. However, if the representation would be taken as the subgraphs are constructed, they would give two different representations even though they are isomorphic. Example 4.2.1 takes these two subgraphs and builds the matrix of each with the vertex invariants approach, resulting in two identical matrices that are represented in the same way; this demonstrates that the vertex invariants approach with the new introduced rules is effective at giving a canonical representation of a subgraph.

Example 4.2.1.

$$M1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \left[\begin{array}{c|cccc} 0 & 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{array}\right] \end{array}, \quad N1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \left[\begin{array}{c|cccc} 0 & 1 & 1 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{array}\right] \end{array},$$

$$M2 = \begin{array}{c} \\ 1 \\ 4 \\ 5 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 4 & 5 & 2 & 3 \end{array} \\ \left[\begin{array}{ccc|cc} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ \hline 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{array}\right] \end{array}, \quad N2 = \begin{array}{c} \\ 1 \\ 4 \\ 5 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 4 & 5 & 2 & 3 \end{array} \\ \left[\begin{array}{ccc|cc} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ \hline 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{array}\right] \end{array},$$

$$M3 = \begin{array}{c} \\ 1 \\ 4 \\ 5 \\ 3 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 4 & 5 & 3 & 2 \end{array} \\ \left[\begin{array}{ccc|cc} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ \hline 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{array}\right] \end{array}, \quad N3 = \begin{array}{c} \\ 1 \\ 4 \\ 5 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 4 & 5 & 2 & 3 \end{array} \\ \left[\begin{array}{ccc|cc} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ \hline 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{array}\right] \end{array},$$

$$M4 = \begin{array}{c} \\ 1 \\ 5 \\ 4 \\ 3 \\ 2 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 5 & 4 & 3 & 2 \end{array} \\ \left[\begin{array}{ccc|cc} 0 & 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ \hline 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{array}\right] \end{array}, \quad N4 = \begin{array}{c} \\ 1 \\ 5 \\ 4 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 5 & 4 & 2 & 3 \end{array} \\ \left[\begin{array}{ccc|cc} 0 & 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ \hline 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{array}\right] \end{array},$$

Apart from the new rules introduced above, the approach remains the same; first the algorithm partitions the vertices based on their degree and locates the vertices in the matrix in order of the degree, Example 4.2.1, shows this in *M2 and N2*, then based on the vertices' label it reorders those which are not in lexicographic order getting a partially ordered matrix, this can be seen on *M3 and N3* in the example. This steps do not take into account the origin vertex as it is described in the first rule. After the application of these rules, the vertices which are in the same degree partition and have the same label are reordered based on the distance to the origin as described in the second rule and exemplified in *M4 and N4*.The result of the application of this approach with the new rules presented produce a matrix which is used to represent the canonical representation of the subgraph.

The construction of the vertex invariant based matrix differs on one step from

directed and undirected graphs. In fact, if we apply all the rules to a directed graph, there is still the possibility of finding different canonical representations when there is a tie on the degree, label and distance to origin, given that there is no differentiation in outgoing and incoming edges. Figure 4.3, shows this problem with two simple graphs which only difference is on the directionality on the edges. In order to overcome this problem an easy fix can be added to the

Figure 4.3: Vertex Invariant Tie

algorithm by simply sorting the vertices of the graph by degree, label, distance to origin and outgoing edges before incoming edges. However, this solution brings a decrease on the performance of the canonical labeling approach.

The vertex invariants approach is efficient at giving a canonical representation of subgraphs in the interesting vertices setting with the addition of two new rules: vertex origin and distances to the origin.

4.2.2 Initial Algorithm Setup

The algorithm has a starting setup in which the smaller subgraphs for the candidate generation are generated. However, [19] does not state which is the path to follow to calculate all the 1-edge and 2-edge subgraphs. Therefore, there are two possibilities learned from the base algorithm and the gSpan algorithm.

1. Base algorithm approach: This approach generates candidates from the first iteration and matches this candidate against the graph.

2. gSpan strategy: The gSpan algorithm first generate all the one edge graphs by exploring the graph directly finding the 1-edge subgraphs which reach the support threshold. In a next step it takes a right most approach shown in section 3.3.2.

Since there is not a theoretical base to get the 1-edge and 2-edges subgraphs in [19]. The gSpan method was selected for this matter. This selection is based on the premise that theoretically gSpan is more efficient than the base algorithm because it avoids the candidate generation and looks up on the fly the existing frequent patterns.

4.2.3 Candidate Generation

The candidate generation in FSG is in charge of identifying the possible candidates that are more likely to be frequent during the mining process. This process has three very well differentiated tasks which are: core identification,

joining of the two candidates which share a same core and finally checking the downward property to verify if the candidates will be a good match.

Core Identification: The core identification is the base of the candidate generation phase given that for every pair of frequent graphs of size k, it checks if they share a common core. The implementation of this task is computational intensive when it is done by brute force. However, a more efficient approach is done by simply saving the frequent subgraphs of size k with the cores of size $k-1$ that originated this subgraph.
Figure 4.4, shows a simple example of two graphs sharing a common core; as an

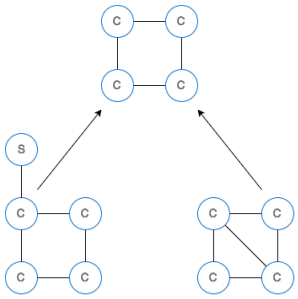


Figure 4.4: Core Example

example of the implementation, the algorithm would save the two graphs at the bottom of the figure with the core at the top. In this way when the candidate generation checks for the common core, it simply checks the canonical labels of the cores of the pair of graphs under analysis.
One important optimization to mark out by the algorithm is the measurement of the combined support of the subgraphs. On each iteration the algorithm saves the supporting vertices of every frequent subgraph, and when is about to join a pair of frequent subgraphs, first, it retrieves the supporting vertices of both and measures the size of the intersection between them. Therefore, when the combined support does not reach the support threshold, it does not continue to join such subgraphs.

Graphs Joining: Once a pair of graphs have been discovered to share the same core of group of cores, the algorithm proceeds to join them, calculating all the possible combinations of these two graphs. The first step for joining two subgraphs that have been detected to share a common core is to identify the edges which are not part of the core. Figure 4.5, shows the procedure to find the edge which does not belong to the core from figure 4.4. The algorithm removes one edge at the time comparing the resulting canonical label with the canonical label of the core and saving the edge once it matched the core. This is done to keep a track of the origin vertices to later use them to facilitate the joining process. As it can be seen from figure 4.5, this process is done in the two graphs that are supposed to generate the candidates.

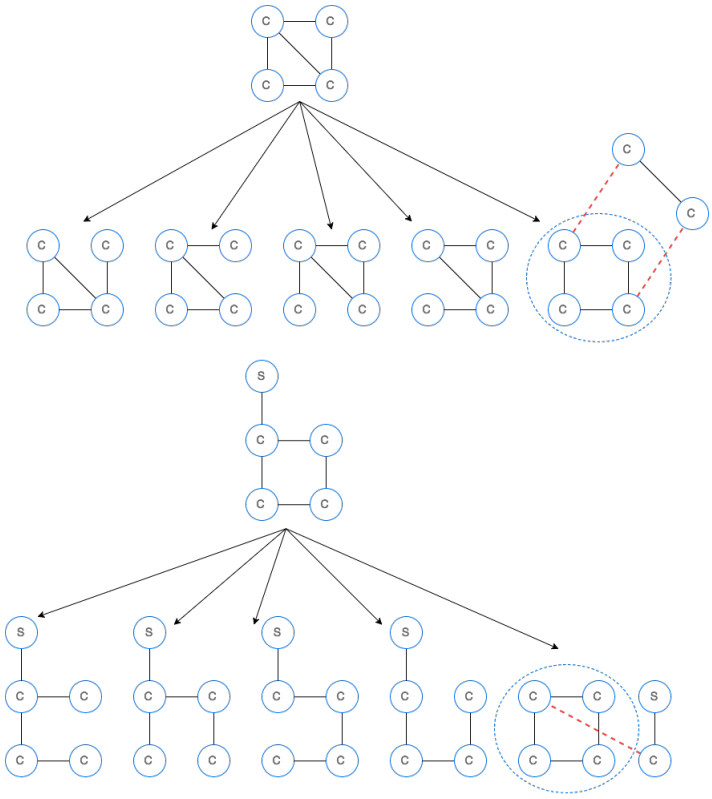


Figure 4.5: Edge identification

Downward Property: The candidates are analyzed to double check that they will be suitable for further examination. This analysis of candidates of size $k+1$ is so called the downward property, and it checks all the possible graphs of size k by erasing the edges of the candidate one at the time, checking if the resulting subgraph of originated from the candidate is fully connected and that it is frequent. The removal of edges takes into consideration that the graph must at least have one edge with the origin vertex present. The checking is done first by a function which checks the path of a graph from the origin and analyses if there is a hole interrupting the connectivity. Figure 4.6, show a graph at the top with four vertices and three edges; it exemplifies correct and incorrect subgraphs. Once the algorithm has checked the connectivity of the graph, it looks up the frequent graph table with the canonical label of the resulting graph; if all the conditions hold, the candidate is suitable for the matching procedure.

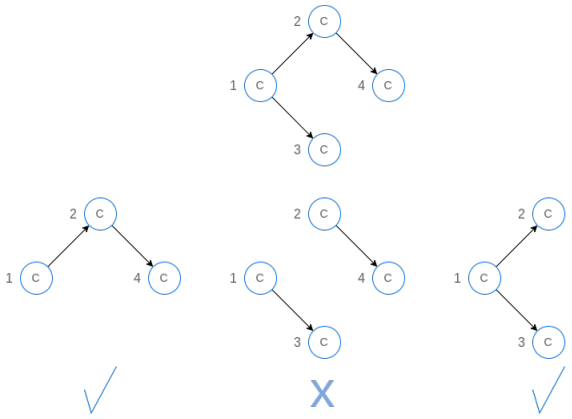


Figure 4.6: Graph Connectivity

4.3 gSpan algorithm implementation

gSpan explores a graph with a depth-first search approach in which the candidate generation is avoided by the implementation of the right-most-path approach. However, in the first iteration it checks all the possible first edges and checks their support among the graph. One of the main advantages of this approach is the pruning of edges that do not reach the determined support threshold from the very first iteration. However, gSpan is an algorithm designed to work with a multiple graph dataset and the support of a subgraph is done by counting the number of graphs in which the subgraph is present; the current problem aims to solve a frequent graph pattern mining among one single graph with a set of interesting vertices. The alternatives to implement gSpan in the significant subgraph mining problem over one single graph are the following:

1. **Interesting vertices approach**: This approach begins the exploration from only the interesting vertices, taking the interesting vertex as origin. However, one difficulty that this approach can bring is that the current problem tries to find the significance of a subgraph with a hyper-geometric distribution and the total support of the pattern among the graph is also needed. This is a difficulty because gSpan is designed to keep projections of the real graph of the discovered patterns and therefore if we explore in a first instance the interesting vertices, getting the projections of the rest of the vertices would not only become challenging to implement but also computationally more intensive.

2. **All vertices approach**: This approach differs from the Interesting vertices one by taking into account every vertex from the graph as an origin of a new graph, this solves the problem of having a two step approach as the Interesting vertices approach by saving every projection of a subgraph in only one step. However, this can result on a high memory consumption

because the algorithm would keep a very large amount of projections at the same time on memory. It is important to note here that this approach is closer from the original algorithm and that the implementation has a strong theoretical background.

The approach chose for the implementation of gSpan is the All vertices approach, given that it has a strong theoretical background and that a two step approach as the Interesting vertices one, presents many difficulties and the algorithm would have major changes, and could result in an long routine implementation. Sub-section 4.3.1 explain the approach in detail. The adjustment from the original algorithm greatly differ from the configuration of the problem, when the configuration is set to directed graphs, important adaptations to the algorithm have to be done, specially in the way the patterns grow respecting the right most path method, this is explain in detain in sub-section 4.3.2.

4.3.1 All vertices approach

The all vertices approach simulates having multiple graphs by using each vertex as a starting point, each vertex is taken individually. The rule of the thumb for this approach is that the vertices are the origin of every subsequently found subgraph "once an origin always an origin". Figure 4.7, shows the differences on the resulting subgraphs if we would be using a pure based gSpan algorithm against the base algorithm developed in [3].

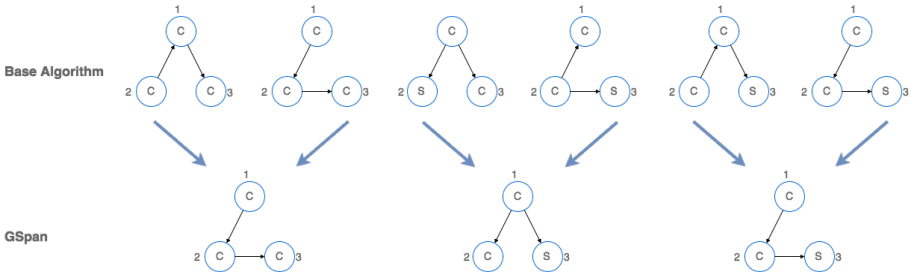


Figure 4.7: Differences of the base algorithm against an original gSpan representation

One equally important difference to note by analyzing Figure 4.7 is that in a pure based gSpan implementation, there is not certainty that the vertex with ID 1 is the original origin vertex that started the pattern. This is because gSpan reorders the edges in order to have always the minimum lexicographic representation of such pattern. For analysis purposes the algorithm is divided in three different steps which are discussed bellow:

Generation of the first frequent edges: This step is crucial in order to keep the minimum lexicographic order of the generated *DFS codes* reviewed

in section 3.3.2.1, the algorithm checks every single pair of connected vertices, their labels and their support, if the support in this first iteration is not reached, a pre-pruning process takes place. The pruning takes place when a specific edge do not have the necessary support in the graph, in consequence all its subgraphs will not reach the support neither. Therefore, the search space can be reduced in the first iteration.
The original implementation reorders the edges in lexicographic order to set a starting point for the subgraph mining step. However, for the current problem domain, this reorder is not possible because if this step is done, it loses track of the origin of the subgraph.

Subgraph mining: The subgraph mining step starts its growing from the set of first frequent edges described before. The growing starts by evaluating first if there is a backward edge from the right most path and if this condition does not hold, it starts to look for the forward vertices from the right most path approach shown in sub-section 3.3.2.1.5. Before the process of adding edges takes place, the algorithm checks if the *DFS code* of the analyzed subgraph is the minimum possible, if it is not, the process discard this *DFS code* and continuous to the next one. A further explanation on the implementation of each of this is presented bellow:

1. **Transactions Ids:** The original gSpan approach keeps a notion of the projection of every subgraph with a transaction ID to keep a record of which graphs are supporting the specific subgraph. However, since the actual problem is seeking to find frequent subgraphs within a single graph, this approach is changed by keeping the IDs of the vertices instead of the graph ID. The supporting vertex is the first vertex of the first edge of the subgraph. This approach is of great use when measuring the total and group support of a subgraph.

2. **The non trivial minimum DFS condition:** As stated before, before growing the edges of a given *DFS code*, the algorithm checks if this code is the minimum possible, this is done to improve the performance by avoiding the checking of unnecessary *DFS codes* and also to avoid duplication. gSpan calculates the *minimum DFS code* by transforming the *DFS code* into a graph and generating a *minimum DFS code* by growing the graph from its origin taking into consideration only the minimum edges at each iteration; for instance we can have this trivial example of a *DFS code*: {1C-2C, 2C-3S, 2C-4C}, this *DFS code* obeys the rules of the lexicographic order presented in 3.3.2.1. However, it is not the minimum possible; the *minimum DFS code* process first determines which is the minimum first edge and starts the construction from it, this is {1C-2C}, in the second iteration it has two edges {2C-3S} and {2C-4C}, here it takes the minimum of the two, resulting in {2C-4C} and then it keeps constructing the *DFS code*. After all iterations, the resulting code will be {1C-2C, 2C-3C, 2C-4S} which is different from the one presented before. Therefore, the

algorithm skips this code and all its descendants. Figure 4.8 shows an example of the minimum DFS code generation.

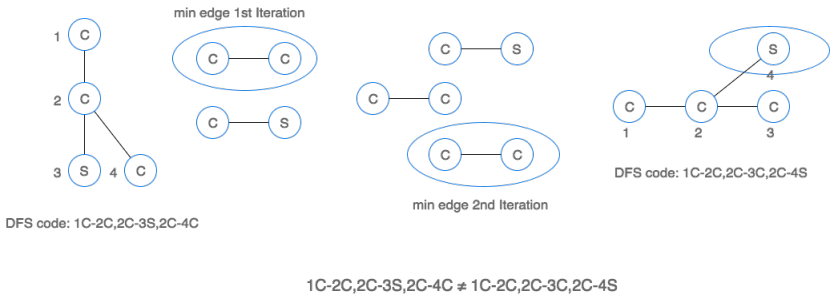


Figure 4.8: Minimum DFS code generation

3. **Backward vertices:** Given the right most path, the backward vertices are tested from the last introduced vertex; eg: given a *DFS code* {1-2, 2-3, 3-4} the backward procedure tries to add one edge at the time, first from the edge {3-4} (vertex 4) to the edge {1-2} (vertex 1), if that edge does not exist, it analyzes if there is one edge from {3-4} (vertex 4) to {2-3} (vertex 2). By looking at the example the reader may have realized that the algorithm add edges from the last vertex in the right most path to the origin vertices on the upper edges on the right most path. However, this only works for an undirected configuration, on a directed configuration the principle is the same but there are more considerations to take into account, this is further described in sub-section 4.3.2.

4. **Forward vertices:** The forward vertices are generated from the right most path by analyzing the neighbors of such vertices, this operation generates one additional vertex and one additional edge, the difference with a backward edge is that the latter only generates one edge connecting existing vertices. When the algorithm finds the neighbors of one particular vertex, it orders them in lexicographic order as stated in sub-section 3.3.2.1. However, this order can not be fully accomplish given specific properties of single graphs.

Results Generation: The advantage of use an approach which discovers all the vertices which support a certain pattern is that the group and total support of a subgraph can be easily checked on the fly and stop the search when a particular subgraph does not have the necessary support.

4.3.2 gSpan for directed graphs

Adapting gSpan to work on a single graph with a directed configuration is not a trivial process, apart from the fact that is a single graph dataset, it also has

to take into account important directed graph properties such as: directionality of the edges, and the admission of self edges into the frequent subgraphs. Given the directionality of the edges, the construction of the right most path is greatly affected. Even though the base core of gSpan can be kept, new concepts have to be introduced to the algorithm.
Three main adaptations have been made to the algorithm to make it work in a directed configuration, one is related to the generation of the 1-edge subgraphs, the second is related to the growth approach of the right most extension and the third is about the minimum lexicographic order.

1-edge frequent graphs: When analyzing 1-edge subgraph on an undirected configuration the algorithm simply checks for all the vertices and their corresponding neighbors given that directionality does not matter here. However, in the case of directed graph, this is not as simple given that a origin vertex can have outgoing vertices and incoming ones. Keeping in mind the rule of the thumb than once an origin always an origin the way of forming the edge changes from outgoing and incoming edges.
The outgoing vertices originated from the vertex under analysis is represented with the ID 1 starting the edge and the subsequent ID 2 following it. As an example 1C-2S. On the other hand, when there is an incoming edge to the vertex under analysis, the pattern is formed first by the neighbor with ID 2 to the vertex under analysis with ID 1, as an example, the pattern 2S-1C which clearly shows the directionality. Figure 4.9, shows and outgoing edge (a) and an incoming one (b).

Figure 4.9: Right Most Path for Directed Graphs

Additionally to the directionality of the edge, here it is also important to note that a self edge can be added if the vertex has a connection to itself. As an example a self edge in the first iteration is represented by 1C-1C.

Pattern Growth approach: The rules which are applied for the right most path of the original approach can not rule a direct graph because of the additional complexity that a directed graph has. In order to use the gSpan core which is this Right Most Extension approach, there are new rules to create and also to reorder the previous ones against the new ones. Figure 4.10 illustrates the rules.
The following are the rules in their corresponding order.

1. **Self Edge:** This is the first rule to apply before growing the subgraph in any other direction. This rule is applied first because it helps to check the self edges without affecting the behavior of the rest of rules. Figure 4.10 (a) shows a self edge in dotted line.

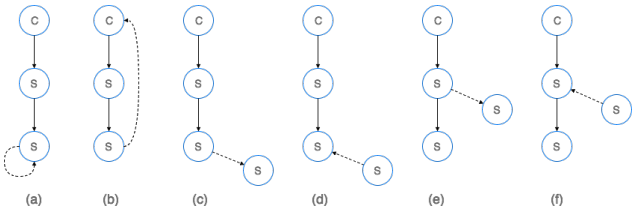


Figure 4.10: Right Most Path for Directed Graphs

2. **Backward Edge:** This rule is applied in the same way that it is stated in sub-section 3.3.2.1.5; it checks the presence of an edge going from the bottom vertex of the right most path to the upper ones in the right most path. Figure 4.10 (b), shows an outgoing backward edge in dotted line. By seeing at Figure 4.10 (b), one can realize that a backward edge can take two directions, having an outgoing edge or an incoming edge. Therefore, the implementation do keep track of the directionality of such edge.

3. **Forward Outgoing Edge:** After the backward vertex the edges with an outgoing directionality are taken into account, this is done the same as the approach discussed in sub-section 3.3.2.1.5. Figure 4.10 (c) and (d) shows the forward outgoing edges in dotted line.

4. **Forward Incoming Edge:** A forward incoming edge in the right most path growth path is a new rule, given that the algorithm takes into account also the edges which are coming to the vertex under analysis; These vertices are the last ones to be analyzed and for it, the right most path approach is used. Figure 4.10 (e) and (f) shows the forward outgoing edges in dotted line.

Figure 4.11, shows four simple graphs which right most path is all the edges of them, exemplifying the directionality problem by showing the way to represent each of them.

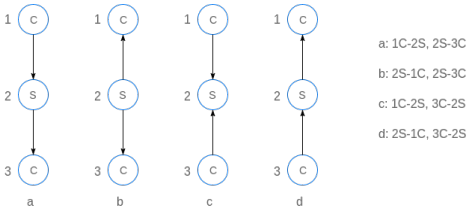


Figure 4.11: Directed Graphs Challenges

Figure 4.12 shows an example of an growth edge from figure (d), this illustrates all the possibilities from the right most path with self, incoming and outgoing edges from the vertices on the right most path of the subgraph.

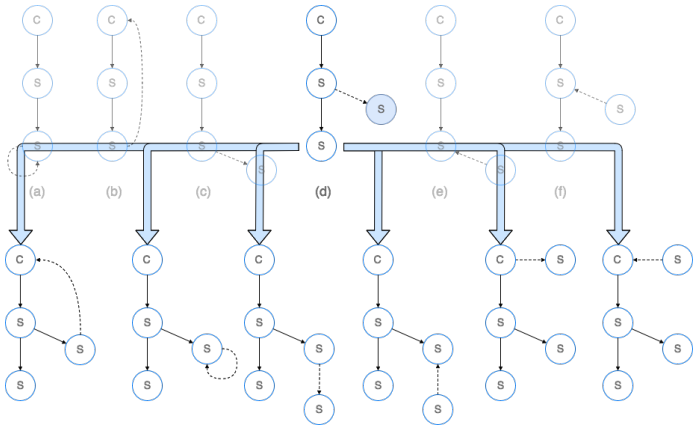


Figure 4.12: Right Most Path example for a Directed graph

Lexicographic Order: The lexicographic order is the most affected property of gSpan in a directed configuration, the growth of edges can not be done keeping a proper lexicographic order because of the directionality of the edges, for instance an edge {*1S-2C*} do not keep a lexicographic order and since we are exploring a directed graph this can not be changed. This requires to sort the possible edges to grow before the actual growing. This pseudo order is important when determining the *minimum DFS code* of a subgraph; for a directed graph this code will start with the lowest edge and is constructed over the rules of the right most path, when a vertex has more than one option to grow then this heuristic will be applied.

The *minimum DFS code* variant: The calculation of the canonical label through the *minimum DFS code* follows the same rules than in an undirected configuration with the variation that the rules to construct the *minimum DFS code* are those described on the Pattern Growth Approach of this sub-section. The only rule which changes is when there are more than one option in a vertex of the right most path which is not the right most vertex. Here, the incoming and outgoing edges are calculated and the ones with a lower vertex in the right most path are picked for the next step.

4.3.3 gSpan for a multiple labels

The generation for a multiple label configuration in this implementation of gSpan is very straightforward since than in every step where a forward edge is created in the right most path, instead of generating one single pattern, multiple patterns are created with all the possible labels that the new node can have including an empty label. These new patterns have the same projection in the graph, however, this creates a big overhead on the memory of the system because a great amount of objects are generated in every step.

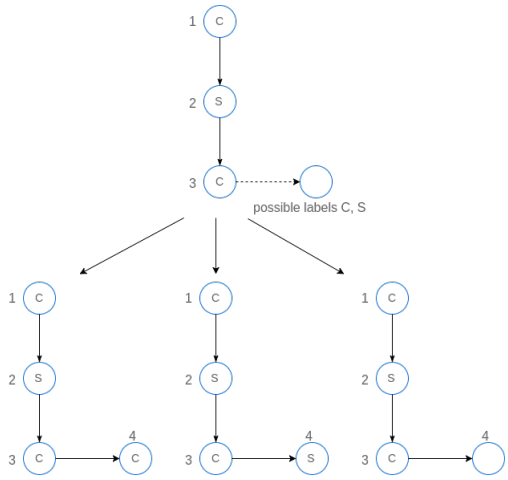


Figure 4.13: Multiple label subgraph generation in gSpan

Figure 4.13, shows an example of the generation of multiple-label subgraphs in gSpan, this approach is applied for directed and undirected graphs.

Chapter 5

Experiments, Performance and Dynamical Selection

This chapter shows the performance and results of the algorithms over different datasets; for this work different biological datasets were tested in a directed and undirected setup with the different implementations of the algorithms. Based on this experiments and their results, the performance is measured among different settings for every dataset. Section 5.1, executes the algorithms over the different datasets, next, Section 5.2, measures the performance in terms of time efficiency and memory usage, finally a method to suggest the best algorithm for a given graph is shown in Section 5.3.
The algorithms presented in Chapter 3 and developed and adjusted to the current problem in Chapter 4, were developed in the programming language Java; to have a uniform approach to test and experiment all the algorithms are tested on the same platform and with the same equipment, a Mac Pro i5 with 8GB of Random Access Memory (RAM).

5.1 Experiments

In order to control the development and to check for possible differences among the algorithms, a number of tests are reproduced with different biological datasets and the parameters shown at Table 5.1. A detailed analysis of the found differences are detailed in the sub-sections 5.1.2, 5.1.6, 5.1.4, 5.1.5.

5.1.1 Experiments Settings

Significant subgraph mining problem is a complex problem because the algorithms take as input multiple parameters which can vary depending on the purpose of the graph miner and the graph itself, this results in a high number of combinations of parameters for testing each algorithm. Therefore, the parameters which are taken into account for the measurements are the following:

1. Directionality of the graph: Directed and undirected graphs are treated separately according to the dataset. The available datasets have one undirected and two directed graphs.

2. Multiplicity of the labels: The algorithms are tested for single and multiple labels depending on the datasets available. The available datasets for the experiments have single label, multiple labels and no labels. A detail of this can be seen in subsections 5.1.4, 5.1.5 and 5.1.6.

3. Number of maximum vertices: The number of maximum vertices is one of the parameters which makes the mining more exhaustive for the CPU and memory usage because a larger number of vertices, results in a larger number of subgraphs to be tested. Therefore, the algorithms are tested with three, four and five maximum vertices. This values were chosen because with a larger number of maximum vertices, the algorithms would run for too long, making the gathering of results not feasible with the equipment used for experimentation.

4. Support threshold: The support threshold tells the minimum number of times the subgraph has to be present to be considered frequent. In consequence, the smaller the support threshold is, the more demanding the processes are, since they have to consider a larger amount of subgraphs. The traditional approach to set the support threshold is to calculate it based on an estimation of the number of subgraphs to be tested in relation with the P-value and the probability that a given support would be significant over the mentioned relation $P - value/Subgraphsstimation$. However, for an experimental setting, this approach does not help with the measurement of the performance of the algorithms on different settings. Therefore, the algorithms are run with fixed supports of two, five, ten, and twenty five, to calculate their performance with small and large supports, this is done given that with the traditional approach is not possible to define the behavior of the algorithms over different supports which is one of the main contributions of this work.

The parameters showed above along with measurements of time efficiency and memory, of every studied algorithm in a specific set of settings, will be used to determine a dynamical selection of the best approach for a particular set of parameters, this is explained in more detail in section 5.3.

Table 5.1, shows the list of possible combinations of parameters for the algorithms' execution. The support and maximum number or vertices were selected according to empirical runs which are able to finish in less than 10 hours among the different algorithms.

5.1.2 Example dataset

An example dataset of a simple small graph is showed in Figure 5.1, this example is specifically created to verify the differences of results among the different

Table 5.1: Parameters for efficiency measurement and dynamical approach

Type of Graph	Labels' multiplicity	Support	Max Vertices
Undirected	Single	2 - 5 - 10 - 25	3 - 4 - 5
	Multi-label	2 - 5 - 10 - 25	3 - 4 - 5
Directed	Single	2 - 5 - 10 - 25	3 - 4 - 5
	Multi-label	2 - 5 - 10 - 25	3 - 4 - 5

implementations of the algorithms. The base approach, the algorithms take different approaches for exploring the search space and discovering significant subgraph and even though there should not be differences on their results, some differences can be found due to the different strategies to identify canonical representations. Figure 5.1, shows a graph with twenty vertices with their corresponding labels in their center, five interesting vertices marked with light blue in their background, and 32 edges connecting the vertices indistinctly. This example was designed to model the behavior of the algorithms with a fully connected graph with a larger number of edges than vertices.

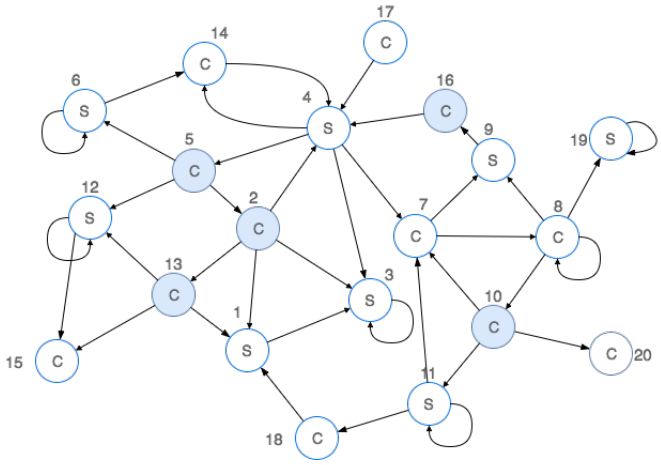


Figure 5.1: Toy Example

Different configurations were used for the runs, this configurations have a support of two with a number of maximum vertices going from two to five with single and multiple labels and for directed and undirected graphs. Table 5.2, shows the results of applying the different algorithms to the example dataset. The first observation over the results of the table is that when the number of maximum vertices is larger, the algorithms begin to behave differently, for instance with a $Freq_{max}$ of four in the single label configuration we can observe that the base algorithm finds 42 frequent subgraphs while gSpan and FSG find 39 and 40 respectively. To detect the reason of these differences, the results

Table 5.2: Algorithms results in an undirected configuration - Example Dataset

	Single label							
	Frequent subgraphs				Significant subgraphs			
$Freq_{max}$	2	3	4	5	2	3	4	5
Base	2	10	42	158	0	0	0	0
gSpan	2	10	39	131	0	0	0	0
FSG	2	10	40	145	0	0	0	0
	Multiple labels							
Base	6	44	372	2.848	0	0	0	0
gSpan	6	44	322	2.106	0	0	0	0
FSG	6	44	312	2.154	0	0	0	0

Table 5.3: Algorithms results in an directed configuration - Example Dataset

	Single label							
	Frequent subgraphs				Significant subgraphs			
$Freq_{max}$	2	3	4	5	2	3	4	5
Base	4	23	77	190	1	1	0	0
gSpan	4	23	77	187	1	1	0	0
FSG	4	23	80	187	1	1	0	0
	Multiple labels							
Base	14	150	1.100	6.340	2	0	0	0
gSpan	14	150	1.100	6.212	2	0	0	0
FSG	14	146	1090	5.963	2	0	0	0

were manually inspected finding that the base algorithm and FSG found 3 and 1 duplicate graphs respectively. Therefore, we can draw the conclusion that the canonical labeling approach of these two algorithms is not completely efficient for an undirected configuration.
A second conclusion that we can draw after analyzing the results on a undirected, multiple label configuration is that gSpan has the best canonical labeling method; the results for this configuration have also proven to not contain any duplicate graph for the gSpan result, while the base approach and FSG again contain duplicates.

The same analysis done for undirected graphs was done as well for directed graphs by analyzing Table 5.3. The findings show that again gSpan has the best canonical labeling method for representing graphs. There were duplicate subgraphs found in the base algorithm and also in FSG for a single and multiple label configuration while gSpan did not calculate any duplicate graph.

gSpan has proven to be the algorithm which uses the best possible canonical labeling approach, avoiding duplicates and finding more accurate results than the other algorithms in every tested configuration of mining parameters.

5.1.3 The Canonical Comparator Tool

The studied algorithms use a different approach to determine graph isomorphism by calculating a canonical label for each frequent subgraph in order to avoid duplicates, saving time and improving the efficiency in the execution. However, after the run of the algorithms over the example dataset described in sub-section 5.1.2, the number of resulting frequent subgraphs differed from one algorithm to another. Therefore, a tool to check the duplicate graphs is necessary to check if such canonical methods are actually working efficiently.
Furthermore, the use of a tool is needed given that the amount of frequent subgraphs resulting after the run of every algorithm is large, finding thousands of subgraphs with some combination of parameters over the experimental datasets, making a manual comparison not feasible. Thus, an automatic canonical comparison technique based on the best possible approach to find canonical labels must be used to compare the algorithms' results.
The selection of the best approach for the construction of the canonical comparator tool is not a trivial process because there is no proof that one algorithm has a better canonical labeling technique than the others. The selection for this tool is purely based on an manual analysis of the results of the Example Dataset shown on section 5.1.2. As a result of this analysis, the algorithm which shows the best canonical calculation for directed and undirected graphs is gSpan; this conclusion was achieved through manual analysis, gSpan did not calculate any duplicate subgraph while the base algorithm and the FSG algorithm did. Tables 5.2 and 5.3, show the number of frequent and significant subgraphs after running the algorithms on the example dataset. The column frequent subgraph clearly shows that gSpan in most cases calculated less subgraphs than the other algorithms and by using the Motif Comparator Tool, it was checked that the base and FSG algorithms calculated duplicate subgraphs in some cases, generating more results. However, in some cases, FSG generated less results than the other algorithms, this cases were analyzed by the tool and turned out that FSG calculated less subgraphs.

5.1.4 Yeast dataset

The Yeast dataset is a graph representing the transcription regulatory network from baker's yeast [3]. This dataset is directed graph, its labels are derived from the gene ontology assignments available in Unitprot-GO for yeast and some vertices have multiple gene ontology annotations [3]. Therefore, this is a multiple-label graph. This graph is formed with 6.402 genes which are the vertices and 48.080 edges connecting those vertices. 900 vertices are selected as interesting because it is said in [3] that these genes are identified as being

Table 5.4: Algorithms run over Yeast dataset

Maximum vertices = 3								
	Frequent subgraphs				Significant subgraphs			
Support	2	5	10	25	2	5	10	25
Base	45.083	27.404	18.096	8.693	4.646	4.960	5.137	4.015
gSpan	34.520	22.218	15.179	8.027	3.604	3.766	3.906	3.356
FSG	19.412	13.135	9.447	5.269	2.339	2.409	2.490	2.238
Maximum vertices = 4								
Base	*TtL*	*TtL*	*TtL*	*TtL*	*TtL*	*TtL*	*TtL*	*TtL*
gSpan	*OoM*	*OoM*	*OoM*	*OoM*	*OoM*	*OoM*	*OoM*	*OoM*
FSG	*OoM*	*OoM*	*OoM*	*OoM*	*OoM*	*OoM*	*OoM*	*OoM*

OoM: Out of Memory Exception
TtL: Time too long, the algorithm keeps running for more than 10 hours

retained in duplicate over time.
This graph is considered more densely connected than the PDB dataset shown in sub-section 5.1.6. given that the vertices have a bigger average degree. To define the density of the graph simple statistics values were analyzed.

- The minimum amount of edges connecting a vertex is 1.
- The maximum amount of edges connecting a vertex is 527.
- The average degree of outgoing edges is 28,23 with a standard deviation of 42,90.

For this dataset, the algorithms are run with a maximum vertices of 3 and 4, given that for a number of vertices of 4, gSpan and FSG showed to have an out of memory exception as showed in table 5.4. Table 5.4, shows the result of running the algorithms over this dataset.

By analyzing the results from the executions with the Motif comparator tool, the following observations can be made:

- By running the Motif comparator tool in a duplicate detection mode, there is a vast number of duplicate subgraphs in the Base algorithm, as example, with a support of 2 and a maximum number of vertices of 3, the base algorithm presents 1.041 duplicates; this indicates that the gSpan algorithm calculates more accurate results than the Base algorithm. However, when increasing the number of vertices, gSpan and FSG algorithms present a memory error.
- By running the Motif comparator tool in a comparison mode, it was identified that the FSG algorithm calculates incomplete results. Therefore, this algorithm is not accurate for dense, directed graphs.

Table 5.5: Algorithms run over Bact dataset with maximum 3 vertices

Maximum vertices = 3								
	Frequent subgraphs				Significant subgraphs			
Support	2	5	10	25	2	5	10	25
Base	37	22	8		1	3	1	
gSpan	30	21	8		1	4	1	
FSG	31	22	8		1	1	1	
Maximum vertices = 4								
Base	460	166	28		5	6	0	
gSpan	*OoM*	*OoM*	*OoM*		*OoM*	*OoM*	*OoM*	
FSG	78	58	27		0	0	0	
Maximum vertices = 5								
Base	6340	1541	97		0	3	0	
gSpan	*OoM*	*OoM*	*OoM*		*OoM*	*OoM*	*OoM*	
FSG	611	370	98		0	0	0	
OoM: Out of Memory Exception								

5.1.5 Bact dataset

The Bact dataset is a directed graph with no labels featuring the combined transcription network from seven bacterial organisms [3]. The dataset contains contains seven independently connected graphs where the selected vertices were defined as the 10 homologs of the PhoR present in the 7 species, which are involved in the regulation of the phosphate homeostasis which is of critical importance for both the energy levels of the organism and the creation of new molecular compounds, such as DNA and RNA, in living cells [3]. The dataset contains 28.609 vertices and 62.675 edges connecting those vertices. This graph is considered densely connected given that the vertices have a big average of edges connecting the vertices. To define the density of the graph simple statistics values were analyzed.

- The minimum amount of edges connecting a vertex is 1.
- The maximum amount of edges connecting a vertex is 2.183.
- The average degree of outgoing edges is 264,18 with a standard deviation of 359,04.

Table 5.5, show the result of running the algorithms over this dataset.

By analyzing the results from the executions with the Motif comparator tool, the following observations can be made:

- Given the few significant subgraphs resulting by the run of the algorithms, for this graphs was not possible to run the Motif Comparator Tool. How-

ever, the number of frequent graphs between the algorithms still presents differences. The output of the algorithms was changed momentarily to frequent graphs instead of significant subgraphs. By analyzing the results, there are duplicate subgraphs in the base algorithm and gSpan presents the most accurate results.

- This is a very densely connected graph and gSpan has proven not being able to calculate the subgraphs because of memory errors.
- FSG presents less results than the Base algorithm with a number of maximum vertices of 4 and 5. As in the last dataset showed in sub-section 5.1.4, FSG has proven to not find accurate results in a directed graph.

5.1.6 PDB dataset

The PDB dataset is a protein graph where the vertices are the amino-acids that make up the protein and the edges denote the residues whose C_α are within 4 angstrom [3]. The selected vertices are those which are reported to bind a manganese Mn-ion.
The dataset is an undirected graph which contains 23.810 vertices with 60.730 edges connecting such vertices. The graph has 20 possible labels and is run in a single label configuration because each vertex is an amino-acid. The selected interesting vertices are 208 which are reported to bind a manganese (Mn) ion as stated before. This graph is considered to not be densely connected given that the vertices do not have a big average of connecting edges independently. To define the density of the graph simple statistics values were analyzed.

- The minimum amount of edges connecting a vertex is 1.
- The maximum amount of edges connecting a vertex is 13.
- The average degree of the vertices is 5,10 with a standard deviation of 2,56.

Given these statistics values, the graph can be said to be not densely connected because the average degree is low. Table 5.6, shows the results of the execution of the algorithm for these dataset.

Table 5.6, shows that for this specific kind of graph, gSpan gets the best results in every configuration, after running the Motif Comparator Tool over the results of the significant subgraph results, the following observations can be made:

- By running the tool in a duplicate selection mode over the results of the base algorithm, there is a vast number of duplicates as significant subgraphs; For instance, by running the tool over the results of the Base algorithm with a configuration of support of 2 and maximum vertices of 4, there are 260 duplicate graphs; the Base algorithm canonical labeling find duplicates such as *1GLU-2ILE,2ILE-3THR,2ILE-4SER,1GLU-3THR* and *1GLU-2ILE,2ILE-4SER,2ILE-3THR,1GLU-3THR* which are the same but they are represented in a different manner.

Table 5.6: Algorithms run over PDB dataset with maximum 3 vertices

Maximum vertices = 3								
	Frequent subgraphs				Significant subgraphs			
Support	2	5	10	25	2	5	10	25
Base	1.435	802	404	57	687	685	403	57
gSpan	1.435	802	404	57	687	685	403	57
FSG	1.435	802	404	57	687	685	403	57
Maximum vertices = 4								
Base	26.299	8.542	2.163	77	7.619	7.351	2.161	77
gSpan	24.846	8.410	2.163	77	7.360	7.219	2.161	77
FSG	24.673	8.420	2.164	77	7.349	7.229	2.162	77
Maximum vertices = 5								
Base	396.968	61.749	7.259	78	56.947	54.021	7.255	78
gSpan	348.982	59.195	7.141	78	53.642	52.167	7.137	78
FSG	TtL	TtL	TtL	TtL	TtL	TtL	TtL	TtL

OoM: Out of Memory Exception
TtL: Time too long, the algorithm keeps running for more than 10 hours

- In a configuration with a support of 2 and a maximum vertices of 4, there were found 260 duplicates by using the Base Algorithm; with the same support and a number of maximum vertices of 5, 4.293 duplicates were found. This shows that the larger the number of vertices, the labeling process of the Base algorithm is more likely to get larger amount of duplicates.
- By running the tool in a duplicate selection mode over the results of the FSG algorithm, it was found that there are duplicates, with a support of 2 and maximum vertices of 4, there are 49 duplicates. However, by running the tool in a comparison mode with the gSpan results, there is a mismatch between them. The FSG algorithm, calculates less no duplicate subgraphs than gSpan.

5.2 Performance

The experimental performance of the algorithms is measured by their memory usage and time efficiency.

5.2.1 Time Efficiency

This sub-section measures the time in seconds to execute the algorithms given the specific set of parameters given in table 5.1. This sub-section shows how the different algorithms perform given the three available datasets and to analyze

Table 5.7: Measures of time over all the configurations in the datasets in seconds

		Yeast Dataset			
	Support	2	5	10	25
$Freq_{max} = 3$	Base	743	712	711	667
	gSpan	13.327	11.027	8.751	7.525
	FSG	10.051	13.381	15.111	7.311
$Freq_{max} = 4$	Base	TtL	TtL	TtL	TtL
	gSpan	NT	NT	NT	NT
	FSG	NT	NT	NT	NT
		Bact Dataset			
$Freq_{max} = 3$	Base	$3,3$	$1,8$	$1,4$	
	gSpan	$33,3$	$31,2$	$27,2$	
	FSG	183	174	174	
$Freq_{max} = 4$	Base	$32,39$	$16,19$	$2,83$	
	gSpan	NT	NT	NT	
	FSG	215	233	246	
$Freq_{max} = 5$	Base	3.451	1.591	$45,8$	
	gSpan	NT	NT	NT	
	FSG	13.830	10.372	7.138	
		PDB Dataset			
$Freq_{max} = 3$	Base	$4,7$	$3,6$	$2,8$	$1,4$
	gSpan	$2,1$	$1,9$	$1,8$	$1,2$
	FSG	$13,4$	$10,4$	$8,3$	$4,2$
$Freq_{max} = 4$	Base	$27,4$	$16,6$	$8,4$	$2,6$
	gSpan	$9,8$	$7,6$	$4,3$	$1,6$
	FSG	2.188	701	290	44
$Freq_{max} = 5$	Base	476	252	37	$3,5$
	gSpan	199	155	$11,4$	$1,9$
	FSG	TtL	TtL	TtL	TtL

NT: No time saved due to Out of Memory Exception
TtL: Time too long, the algorithm keeps running for more than 10 hours

the differences in the times and explains why an algorithm can be better for an specific dataset and set of parameters than others. Table 5.7, show the results of the executions. In Table 5.7, it is seen that for the different datasets, the algorithms have different time efficiency through the different configurations of support threshold and maximum number of vertices. There is a general trend in all the algorithms that can be seen, which shows that the smaller the support and the larger the number of maximum vertices, the more time it takes to calculate all the frequent subgraphs. According to these parameters, the algorithms calculate more and larger subgraphs respectively. However, it is necessary to analyze the algorithms independently according to the dataset because they

present different scenarios of testing. However, the times for FSG algorithm are present in table 5.7, it is excluded from the directed graphs analysis given that subsections 5.1.4, 5.1.5, and 5.1.6, show that the results of the algorithm are incomplete.

- Yeast Dataset: The Yeast dataset is a densely connected, direct, multiple-label graph. For this kind of graph, the base algorithm has shown the best time efficiency for any support and maximum number of vertices. On the other hand, gSpan has shown to up to 18 times slower than the base algorithm with a support of two and a maximum vertices of 3. For a bigger number of vertices, there was no time recorded given an Out of memory exception happened in gSpan, due to the overload of memory during the execution of the algorithm. This memory issue is described in sub-section 5.2.2. It is concluded that the base algorithm would perform the best for directed and densely connected graphs.

- Bact Dataset: The Bact dataset is a densely connected, direct graph with no labels. As in the case of the Yeast dataset, the base algorithm presents the best results while gSpan shows larger times with a maximum number of vertices of three, and also gives out of memory exceptions when the number of maximum vertices is larger. It is concluded that for this kind of graph, the base algorithm also presents the best results.

- PDB Dataset: The PDB dataset is a undirected, single labeled graph with no dense structure. For this kind of graphs, gSpan has shown to perform better than any of the other algorithms, reducing the time of execution at least in half if it is compared with the base algorithm and much more by comparing it with the FSG algorithm which shows much more larger execution times. It is concluded that for this kind of graph, gSpan shows the best performance without any error in the execution.

5.2.2 Memory Usage

The algorithms are written in Java and therefore a specific method to measure their memory usage must be used in order to get the value most approximated to reality. Java divides the the memory of a running program into Stack threads memory and Heap memory; the first one is used for short live objects used during the execution of a thread, whenever an object is invoked a block containing references to the object and its primitive types is created; the latter is used to allocate memory to the objects instantiated by the program, the heap memory contains larger objects and static objects referenced during the entire execution of a program. The size of the heap has a limit and whenever this limit is reached the Java Garbage Collector is called to find and clean death objects and objects with no references from the heap memory.

Figure 5.2 shows the Java Memory Model which is composed by Thread Stack Memory (Green Zones) and the Heap Memory (Blue Zone). The focus of

the memory measurements for this work is on the Heap memory.

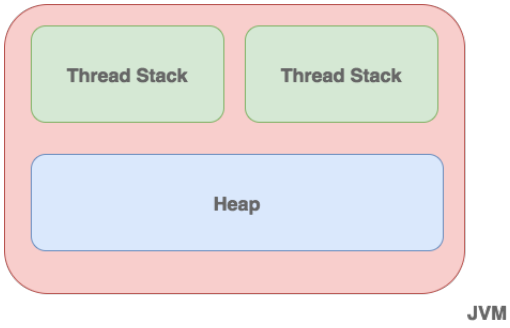


Figure 5.2: Java Memory Model [23]

Figure 5.3, shows the way the heap memory is addressed by the Java Memory Model which is composed by *Young Generation* and an *Old Generation* memory. The first one is composed by the Eden and survivor memory and contains short lived objects; when the Eden is filled the Garbage Collector comes into play and cleans the space for new objects carrying the live objects to the survivor memory. The second one is the home of long lived objects, this type is larger than the young memory and when it is filled, the Garbage Collector cleans it, operation which takes a long time and stops the other execution threads until the operation is concluded.

Figure 5.3: Java Memory Model [24]

As stated before, the heap size is changing through the execution of a program and its measurement do not result to be an exact estimate of the real memory allocation of a running algorithm. However, in Java, the measurement of the used part of the heap provides a good method to have an approximation of the real memory used by an algorithm. Therefore, this sub-section focuses in extracting statistics about the heap size used of a running algorithm given a set of parameters. The measurements are done with the algorithms running and standing alone in the operating system, this means that there are no other programs of any kind running at the same time. Also, the heap size is changed from the original size to a value of 8.192 Mb. This java parameter is changed to give the algorithms enough space to run and to call the Garbage Collector on the Old Memory the less possible, so the running time of the algorithm is more

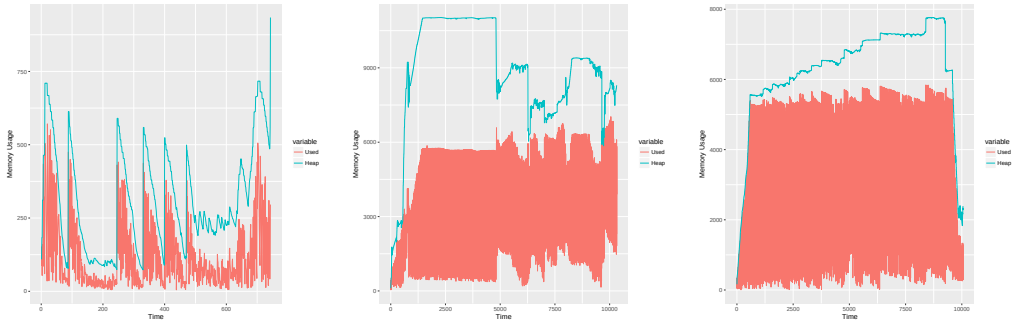
Table 5.8: Measures of memory usage over all the configurations in the datasets in MB

		Yeast Dataset			
	Support	2	5	10	25
$Freq_{max} = 3$	Base	117	118	127	127
	gSpan	3.371	3.666	3.425	3.151
	FSG	2.767	2.622	2.888	2.789
$Freq_{max} = 4$	Base	TtL	TtL	TtL	TtL
	gSpan	OoM	OoM	OoM	OoM
	FSG	OoM	OoM	OoM	OoM
		Bact Dataset			
$Freq_{max} = 3$	Base	108	63	98	
	gSpan	342	355	344	
	FSG	455	521	476	
$Freq_{max} = 4$	Base	105	115	109	
	gSpan	OoM	OoM	OoM	
	FSG	572	756	613	
$Freq_{max} = 5$	Base	571	249	140	
	gSpan	OoM	OoM	OoM	
	FSG	1519	1056	855	
		PDB Dataset			
$Freq_{max} = 3$	Base	$84,75$	$75,66$	78	72
	gSpan	64	50	70	82
	FSG	$119,23$	$121,5$	$123,13$	$94,5$
$Freq_{max} = 4$	Base	$154,51$	$160,44$	$104,88$	86
	gSpan	$144,3$	$129,14$	$82,25$	88
	FSG	$770,02$	$698,18$	$601,21$	355
$Freq_{max} = 5$	Base	$755,87$	$613,89$	$183,73$	$89,66$
	gSpan	$698,22$	$597,58$	$132,72$	86
	FSG	TtL	TtL	TtL	TtL

OoM: Out of Memory Exception
TtL: Time too long, the algorithm keeps running for more than 10 hours

accurate and tries to avoid out of memory exceptions too.
Table 5.8, shows the results of the amount of memory usage in Mb of the algorithms over the different datasets. Since all the datasets show different properties and a different configuration setting, they are analyzed separately.

Memory Usage Yeast dataset: The Yeast dataset is a dense and large graph which contains a high number of interesting vertices. Therefore, the algorithms find a high number of interesting subgraphs, this can be noted on Table 5.4. This high number of interesting vertices results subsequently in a



(a) Base Algorithm (b) gSpan Algorithm (c) FSG Algorithm

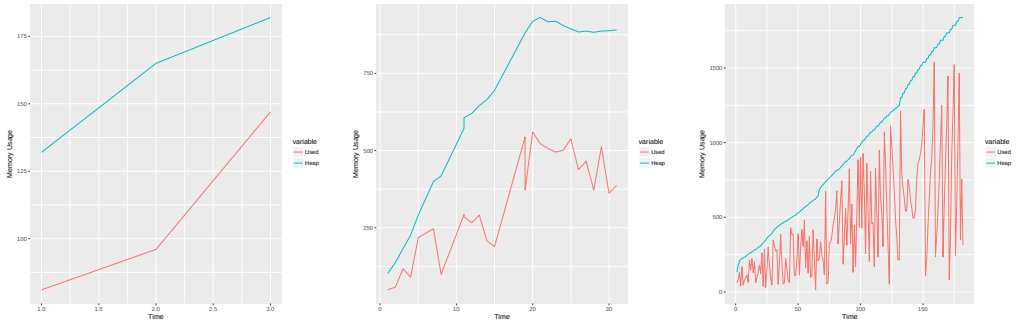
Figure 5.4: Memory Usage Yeast dataset, Maximum vertices = 3, Support = 2

high number of frequent subgraphs which has a notorious effect on the memory usage at execution time. Table 5.8, shows that the only algorithm which keeps the memory usage in low levels is the Base algorithm, this algorithm only keeps the frequent subgraphs on memory and there are not long objects that are kept during the entire run of the algorithm. On the other hand, gSpan keeps several long objects on memory, it keeps all the projections of a given subgraph on memory until there are not more possible subgraphs to find. This makes gSpan performs badly in graphs which are very densely connected given that the amount of projections greatly increases.

Figure 5.4, shows the used memory and heap size of the different algorithms running on the Yeast Dataset with a number of maximum vertices of 3 and a support of 2. It can be seen on the figure and also on table 5.8, that the Base algorithm performs the best while gSpan the worst. FSG also keeps a high number of objects on memory. However, FSG is not further taken into account for the analysis given its large execution times shown in sub-section 5.2.1 and its bad results shown in sub-section 5.4.

Memory Usage Bact dataset: The Bact dataset presents a similar scenario than the dataset presented before, The Bact dataset is a densely connected graph with a few interesting vertices. Although, the number of interesting vertices is not large, the degree of connections of the vertices makes gSpan performs poorly in this algorithm, it throws memory exceptions when the number of vertices is greater than 3. As in the dataset before, the base algorithm shows the best memory usage given that it does not keep a large amount of objects on memory.

Figure 5.5 and Table 5.8, show that the Base algorithm keeps low levels of memory usage while gSpan and FSG increases the memory usage rapidly. As in the last case, FSG is not included for further analysis given of poor results shown in Table 5.5 and long times shown in Table 5.2.1.



(a) Base Algorithm (b) gSpan Algorithm (c) FSG Algorithm

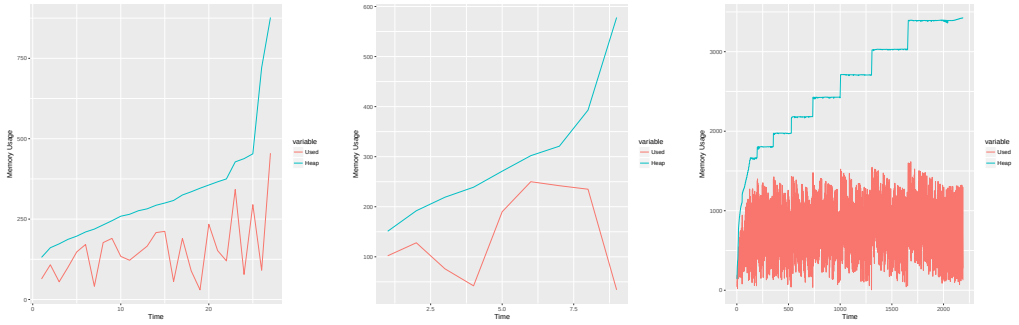
Figure 5.5: Memory Usage Bact dataset, Maximum vertices = 3, Support = 2

Memory Usage PDB dataset: The PDB presents an undirected not densely connected graph. This case shows a big difference than the other two datasets, in the terms of time efficiency shown in sub-section 5.2.1 and as well in memory. On the contrary than with densely connected graphs, gSpan presents the best memory efficiency, keeping the lowest levels of memory among the analyzed algorithms. This is because when the graph has a low degree on its vertices, the algorithm keeps a small number of projections in every iteration; this makes the algorithm perform better with this dataset. The base algorithm keeps a low memory too because it does not save too many objects into memory on the candidate generation, matching process and canonical calculation. The FSG algorithm has the lowest time and memory performance, the candidate generation step of this algorithm is heavily affecting memory because it generates a large amount of short lived objects at every iteration and as well it keeps a long list of objects from previous iterations which influence the memory usage as well.

Figure 5.6 and Table 5.8, show that gSpan performs the best for any configuration of support and maximum number of vertices. The base algorithm also has a good memory performance for this case.

5.3 Dynamical Selection

One of the goals of this work is to find an optimal strategy to pick automatically the best algorithm given a graph and a set of parameters. However, through the experiments some observations can be made, but the amount of datasets at hand are not enough to make this automatic choice.
Given that the available datasets are one undirected subgraph and two directed graphs, additional executions of the algorithm were done, trying the datasets



(a) Base Algorithm (b) gSpan Algorithm (c) FSG Algorithm

Figure 5.6: Memory Usage Pdb dataset, Maximum vertices = 4, Support = 2

with the other configuration as well. The algorithms were run with a directed and undirected configuration with the different support thresholds and a maximum number of vertices of 3; this was done in order to clarify if the differences on time efficiency are because of the configuration or mainly based on the time of graph.

Table 5.9: Measures of time over other configurations in the datasets in seconds

Yeast Dataset with Undirected Configuration				
Support	2	5	10	25
Base	99	97	95	89
gSpan	2.660	2.333	1.690	1.712
Bact Dataset with Undirected Configuration				
Base	1, 15	1, 17	1, 20	
gSpan	16, 21	16, 58	16, 60	
PDB Dataset with Directed Configuration				
Base	3, 01	2, 78	2, 20	1, 46
gSpan	2, 05	1, 93	1, 81	1, 55

Table 5.9, shows the Yeast and Bact datasets run with an undirected configuration and the PDB dataset run with a directed one. As it can be seen from the table, the algorithms keep the same trend; this is, the Base Algorithm works better for the Bact and Yeast datasets, while gSpan works better for the PDB dataset. In other words, the gSpan algorithm shows to be more efficient when mining graphs with not densely connected vertices in a directed and undirected configuration, while the Base algorithm works better for more densely connected graphs.

Even though it is establish the kind of graphs that work better for the used algorithms, it is still not possible with the available data to define until which point of density gSpan is better than the Base algorithm. In order to draw a line of when to use one algorithm or the other, there is the need to perform more experiments with a bigger amount of graphs with different densities. Therefore, with the performed experiments of this work, there is only the possibility to make a recommendation when to use one algorithm or the other in terms of time efficiency. The recommendation is to use gSpan if the average degree is 5 or lower and to use the base algorithm when the average is 20 and greater. However, there is a blind zone from 6 to 19, where it is not possible to make a recommendation of which algorithm to use.

Chapter 6

Conclusions

Graph mining is a complex task due to the structure and challenges that graphs present. The graph mining gets even more complex with the introduction of several parameters that a mining routine can have; parameters such as: the directionality of the graph, the average degree of the graph's nodes, multiple labels, different levels of support and maximum number of vertices make the construction of an efficient general approach a real test for the graph miner.

Most of the researched algorithms focus their attention only on undirected graphs where the directionality is not important. However, this does not reflect the reality of the graph datasets, given that, there are more and more directed graphs datasets emerging everyday.

The studied algorithms focus on finding frequent subgraphs in a dataset which contains several graphs. However, the current domain explores frequent subgraphs starting from a set of interesting nodes in a single graph. Therefore, there was necessary to adapt the selected algorithms to work in this domain by losing some of their algorithms' characteristics and adding some new ones. The results of these adaptations were compared against the Base algorithm, and found out accurate results in gSpan but not as good in FSG.

There are two well known approaches for walking the search space in a graph, these are: Breadth First Search and Depth First Search. They both show advantages and disadvantages when mining a graph. In consequence, there are different approaches to find frequent subgraph patterns which use one or the other. For this work, one algorithm on each category was selected: FSG for a BFS approach and gSpan for a DFS approach. Literature cites these two as being some of the best available algorithms in each approach.

A-priori algorithms such as FSG, have proven to not be an efficient approach for the significant subgraph mining problem, this is because they have bad time performance and memory efficiency, mainly due to very costly candidate gener-

ation routines. The candidate generation in FSG has proven to consume a large amount of computational resources such as memory and CPU.

FSG proved to present very inaccurate results when dealing with directed graphs, one of the possible reasons is in the canonical labeling technique does not behave properly with directed graphs and also, the candidate generation skips some subgraphs given the omission of common cores or confusion in getting the directionality of the new edges.

gSpan resulted to be a very efficient algorithm for not densely connected graphs, giving the best results on time and memory efficiency. However, gSpan performance is greatly decreased for densely connected graphs, the main reason for this is that gSpan needs large amounts of memory to keep all the information it needs to compute the frequent subgraphs. In scenarios with very large amounts of memory, gSpan should work fine for densely connected graphs too. However, this has not been proven given the resources available for this work.

The execution of the algorithms over some datasets produce thousands and slight differences on the number of results. Therefore, an automatic way to compare this results had to be implemented, to do so, the best canonical labeling technique had to be found for directed and undirected graphs. Through manual inspection on a small dataset, it was determined that gSpan produces the most accurate results on its canonical technique for directed and undirected graphs. Therefore, the Motif Comparator tool was implemented based on the gSpan labeling technique.

After running the Motif Comparator, it was discovered that the Base and FSG algorithms produce large amount of duplicates. This allowed to define the gSpan canonical technique as the best against the Vertex invariant method used by FSG and the method used by the Base algorithm.

gSpan requires a large amount of memory when dealing with graphs with a density and it is very efficient with less dense graphs. On the other hand, the Base algorithm which uses a technique of candidate generation with DFS without keeping projections in memory proven to be efficient in most scenarios keeping a low memory usage level, however, having large amounts of memory will not necessarily improve the performance of this algorithm.

The construction of a dynamical method to choose an algorithm automatically for a given dataset was not feasible in this work. There was needed more datasets in order to draw a line between when the Base algorithm or the gSpan algorithm perform best according to the density of the graph. However, it is not discarded that the other parameters would play a role in choosing the best possible algorithm for a graph. More experiments are needed to make this automatic selection possible.

Chapter 7

Future Work

Different canonical approaches have been showed in this work, with different results and efficiency in the calculations. However, there is no work in the comparison of these approaches in terms of time efficiency and memory usage. Comparisons of the presented canonical labeling techniques with others would be of great help to improve data mining algorithms.

This work have introduced exact techniques to mine data from a graph. However, novel techniques with non-exact graphs on Big graph mining are coming out. The research of these technique and their heuristics would provide more insight to improve the studied mining algorithms.

Different biological datasets have been used for experimentation in this work. However, these datasets were not enough to provide a dynamical way to categorize graphs and choose the best algorithm for them. Therefore, the use of more datasets or the creation of synthetic datasets to make more experiments would be valuable to be able to make clear and efficient decisions on the use of a particular algorithm for a given graph.

Common graph mining techniques can be divided in some states or phases that they all follow in a specific way, an example can ve candidate generation, canonical calculation and search method; ant they all can perform different according to the graph and the configuration the graph miner will use. Therefore, an approach able to combine techniques from different algorithms would be important in order to get the best efficiency on the executions.

Bibliography

[1] D. J. Cook and L. B. Holder, *Mining graph data*. John Wiley & Sons, 2006.

[2] L. Han, Z. Wu, and Q. Zhao, "Revealing the molecular mechanism of colorectal cancer by establishing lgals3-related protein-protein interaction network and identifying signaling pathways," *International journal of molecular medicine*, vol. 33, no. 3, pp. 581–588, 2014.

[3] P. Meysman, Y. Saeys, E. Sabaghian, W. Bittremieux, Y. van de Peer, B. Goethals, and K. Laukens, "Mining the enriched subgraphs for specific vertices in a biological graph," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 2016.

[4] S. Even, *Graph algorithms*. Cambridge University Press, 2011.

[5] C. Jiang, F. Coenen, and M. Zito, "A survey of frequent subgraph mining algorithms," *The Knowledge Engineering Review*, vol. 28, no. 01, pp. 75–105, 2013.

[6] L. Kris, "Network biology," Antwerp University Lecture, 2017.

[7] V. Krishna, N. R. Suri, and G. Athithan, "A comparative survey of algorithms for frequent subgraph discovery," *Current Science*, pp. 190–198, 2011.

[8] D. Ajwani, U. Meyer, and V. Osipov, "Breadth first search on massive graphs." in *The Shortest Path Problem*, 2006, pp. 291–308.

[9] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[10] A. Inokuchi, T. Washio, and H. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data," *Principles of Data Mining and Knowledge Discovery*, pp. 13–23, 2000.

[11] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining, technical report," University of Illinois at Urbana Champaing, Tech. Rep., 2002.

[12] M. Kuramochi and G. Karypis, "Frequent subgraph discovery," in *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*. IEEE, 2001, pp. 313–320.

[13] S. Nijssen and J. N. Kok, "A quickstart in frequent structure mining can make a difference," in *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2004, pp. 647–652.

[14] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*. IEEE, 2002, pp. 721–724.

[15] M. Krzywinski and N. Altman, "Points of significance: Significance, p values and t-tests," *Nature methods*, vol. 10, no. 11, pp. 1041–1042, 2013.

[16] S. Parthasarathy, S. Tatikonda, and D. Ucar, "A survey of graph mining techniques for biological datasets," in *Managing and mining graph data*. Springer, 2010, pp. 547–580.

[17] T. Ramraj and R. Prabhakar, "Frequent subgraph mining algorithms–a survey," *Procedia Computer Science*, vol. 47, pp. 197–204, 2015.

[18] M. Wörlein, T. Meinl, I. Fischer, and M. Philippsen, "A quantitative comparison of the subgraph miners mofa, gspan, ffsm, and gaston," in *PKDD*, vol. 5. Springer, 2005, pp. 392–403.

[19] M. Kuramochi and G. Karypis, "Frequent subgraph discovery tr 01-128," University of Minnesota, Tech. Rep., 2001.

[20] X. Liu and D. Klein, "The graph isomorphism problem," *Journal of Computational Chemistry*, vol. 12, no. 10, pp. 1243–1251, 1991.

[21] A. T. Balaban and T.-S. Balaban, "New vertex invariants and topological indices of chemical graphs based on information on distances," *Journal of Mathematical Chemistry*, vol. 8, no. 1, pp. 383–397, 1991.

[22] J. Isom. Graph pattern mining (gspan) - introduction. [Online]. Available: http://simpledatamining.blogspot.be/2015/03/graph-pattern-mining-gSpan-introduction.html

[23] J. Jenkov. Java memory model. [Online]. Available: http://tutorials.jenkov.com/java-concurrency/java-memory-model.html

[24] ——. Memory management in java. [Online]. Available: http://www.journaldev.com/2856/java-jvm-memory-model-memory-management-in-java