EAST WEST UNIVERSITY

A comparative study regarding the effect of weight constraint in feature extraction from malware call graph

by

Mohammad Sohag Rana 2015-2-60-074
Raghib Shahriyer 2015-2-60-076
Faddilin Mahady Riniyad 2015-2-60-055

A thesis submitted in partial fulfillment for the
degree of Bachelor of Science

in the
Jesan Ahammed Ovi
Department of Computer Science and Engineering

May 2019

Declaration of Authorship

We, Mohammad Sohag Rana, Raghib Shahriyer, Faddilin Mahady Riniyad, declare that this thesis titled, 'A comparative study regarding the effect of weight constraint in feature extraction from malware call graph' and the work presented in it are our own. We confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has not previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where we have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely our own work.
- We have acknowledged all main sources of help.

Mohammad Sohag Rana

Signed: ______________________________

Date: ______________________________

Raghib Shahriyer

Signed: ______________________________

Date: ______________________________

Faddilin Mahady Riniyad
Signed: ______________________________

Date: ______________________________

“The difference between stupidity and genius is that genius has its limits..”

– Albert Einstein

Letter of Acceptance

This project entitled A comparative study regarding the effect Of weight constraint in feature extraction From malware call graph " submitted by Raghib Shahriyer (ID: 2015-2-60-076) , Md. Sohag Rana (ID: 2015-2-60-074) and Faddilin Mahady Riniyad (ID: 2015-2-60-055) to the Computer Science and Engineering Department, East West University, Dhaka-1212, Bangladesh is accepted as satisfactory for partial fulfillment of requirements for the degree of Bachelors of Science(B. Sc.) in Computer Science and Engineering.

Jesan Ahammed Ovi , Lecturer Department of Computer Science and Engineering East West University Aftabnagar,Dhaka-1212, Bangladesh	....
Dr. Ahmed Wasif Reza Associate Professor and Chairperson Department of Computer Science and Engineering East West University Aftabnagar, Dhaka-1212, Bangladesh	....

EAST WEST UNIVERSITY

Abstract

Jesan Ahammed Ovi
Department of Computer Science and Engineering

Bachelor in Science

by Mohammad Sohag Rana 2015-2-60-074
Raghib Shahriyer 2015-2-60-076
Faddilin Mahady Riniyad 2015-2-60-055

We investigate two approaches for frequent graph-based pattern mining in graph datasets. The first is an algorithm called gSpan (graph based substructure pattern mining) which discovers frequent substructures without generating candidates. The algorithm generates a new lexicographic order among graphs and maps each graphs to a unique minimum DFS code as its canonical label. Based on this order, it adopts DFS search strategy to mine frequent connected subgraphs efficiently. The other approach is algorithm called WFSM-MaxPWS which is a weighted frequent subgraph mining algorithm. Weighted frequent subgraph mining comes with an inherent challengenamely, weighted support does not hold the downward closure property, which is often used in mining algorithms for reducing the search space. The algorithm uses the MaxPWS pruning technique reducing search space significantly without changing subgraph weighting scheme while ensuring completeness. We have investigated the effect of weight constrains in feature extraction in this thesis study.

Acknowledgements

We have arrived at a point of achieving a goal in our life through various interactions and help from others. However, written words are often elusive and often not enough to express what we feel. Therefore, we would not like to make eorts to nd best words to express our thankfulness. First of all, we would like to express our deepest gratitude to the almighty for His blessings on us. Next, our special thanks go to our supervisor, Jesan Ahammed Ovi, for leading the way and giving us this opportunity. His office was always open for us whenever we needed suggestions, encouragements. We would also like to thank Md. Ashraful Islam for his unforgettable support and suggestions regarding the thesis. Without their help this accomplishment would not have been possible. Last but not the least, We would like to thank our parents for their unending support, encouragement and prayers. There are numerous other people too who have shown us their constant support and friendship in various ways in our academic life. We cannot mention them all. Thank you. . . .

Mohammad Sohag Rana

April, 2019

Raghib Shahriyer

April, 2019

Faddilin Mahady Riniyad

December, 2019

Contents

List of Figures

Dedicated to our beloved families . . .

Chapter 1

Introduction

1.1 Introduction

Malware, or malicious software, is any program or file that is harmful to a computer user. Types of malware can include computer viruses, worms, Trojan horses and spyware. These malicious programs can perform a variety of different functions such as stealing, encrypting or deleting sensitive data, altering or hijacking core computing functions and monitoring users' computer activity without their permission.

Analogous to the human immune system, the ability to recognize malware families and in particular the common components responsible for the malicious behavior of the samples within a family would allow anti-virus products to proactively detect both known samples as well as future releases of samples belonging to the same malware family. To facilitate the recognition of similar samples or commonalities among multiple samples which have been subject to modification, a high-level structure, i.e. an abstraction, of the samples is required. One such abstraction is the call graph. A call graph is a graphical representation of a binary executable in which functions are modeled as vertices, and calls between those functions as edges. So far, only a limited amount of research has been published on automated malware identification and classification through call graphs. To extract the features of malware from these call graphs we need to work on graph mining.

Graph mining is concerned with the identification of patterns within graph data of various forms. One form of graph mining is frequent sub-graph mining which aims to identify frequently occurring patterns (sub-graphs) across a collection of small graphs or within one large graph. gSpan and Weighted gSpan are two major techniques to mine graph data from graph database. gSpan is a software package of mining frequent graphs in a graph database. Given a collection of graphs and a minimum support threshold, gSpan is able to find all of the subgraphs whose frequency is above the threshold. Weighted gSpan is used in weighted graph mining. In Weighted gSpan edge weight gets the highest priority after discovery time of the vertex.

Both the algorithms mentioned above are used to mine frequent sub-graph from graph database. Both of them has some limitations. So if it is calculated that which one is better in a particular task, it will be more beneficial to mine graph from graph databases. Extracting proper features of a malware call graph is very important to detect the malwares.

1.2 Motivation

Graph mining is a time consuming task. So going through a process without knowing whether this is better or not can waste a lot of time. So choosing the appropriate algorithm is very important in graph mining. When its about malware then the right way is more significant. There should have the way of choosing appropriate process in particular aspect. gSpan and Weighted gSpan are two major algorithms for graph mining. So its needed to know which one is better in feature extraction from malware call graph. This phenomenon motivated us to study on the comparison between these two algorithms. Is the weighted gSpan necessary to extract feature from malware call graph at all ? This question actually pushed us to go through this research.

1.3 Objective

Like the human flu, it interferes with normal functioning. So to have the frequent features of malware it is require to mine the malware call graph. But mining is always dependent on the mining techniques. To have the most frequent patterns of malware we need a better approach of graph mining. So, we are looking for whether the algorithm gSpan or weighted gSpan is more suitable to extract frequent pattern from malware call graph.

Goal of this thesis is Malware or malicious software is an umbrella term that describes any malicious program or code that is harmful to systems. Hostile, intrusive, and intentionally nasty, malware seeks to invade, damage, or disable computers, computer systems, networks, tablets, and mobile devices, often by taking partial control over a devices operations to:

- Find frequent sub-graphs from malware call graph using gSpan algorithm
- Find frequent sub-graphs from malware call graph using Weighted gSpan algorithm
- Use these patterns to find out the malwares and then provide the algorithm better in performance.

1.4 Challenges

Mining is always a difficult job to do. But when the datasets are so large then it becomes more difficult as well as complicated. Malware call graph is a large dataset. There are almost thousands of graph having so many edges in each graph. So extracting frequent features from these graphs takes long time. Moreover, we did extracting features for different threshold. So, it took more time to get the features.

1.5 Our Contribution

In this research, we have tried to analyze the comparative study on weight constraint in feature extraction from malware call graph. We have studied gSpan and WeightedgSpan algorithms properly and analyzed them to find out if the weighted gSpan actually is better performing than gSpan or not.

1.6 Outline

Rest of the report is organized as follows:

- Chapter 2: Relevant background topics and graph mining related works.
- Chapter 3: Data description and our proposed framework
- Chapter 4: Experimental results.
- Chapter 5: Summary and conclusion of the thesis and future research scope.

Chapter 2

Related Works

Frequent substructure pattern mining has been an emerging data mining problem with many scientific and commercial applications. As a general data structure, labeled graph can be used to model much complicated substructure patterns among data. In this chapter we will discuss about works related to graph mining and weighted graph mining.

2.1 Background Study

To set the platform for this work, we start from introducing the concept of data mining in this chapter. After that, we will discuss about graph mining and two algorithms of our study.

2.1.1 Data Mining

Data mining is defined as the process of extracting previously unknown, comprehensible, and actionable information from large databases and using it to make crucial business decisions. It is the practice of automatically searching large stores of data to discover patterns and trends that go beyond simple analysis. It uses sophisticated mathematical algorithms to segment the data and evaluate the probability of future events. Data mining is also known as Knowledge Discovery in Data or simply interesting pattern extraction from raw data.

Figure 2.1 illustrates the phases, and the iterative nature, of a data mining project. The process flow shows that a data mining project does not stop when a particular solution is deployed. The results of data mining trigger new business questions, which in turn can be used to develop more focused models.

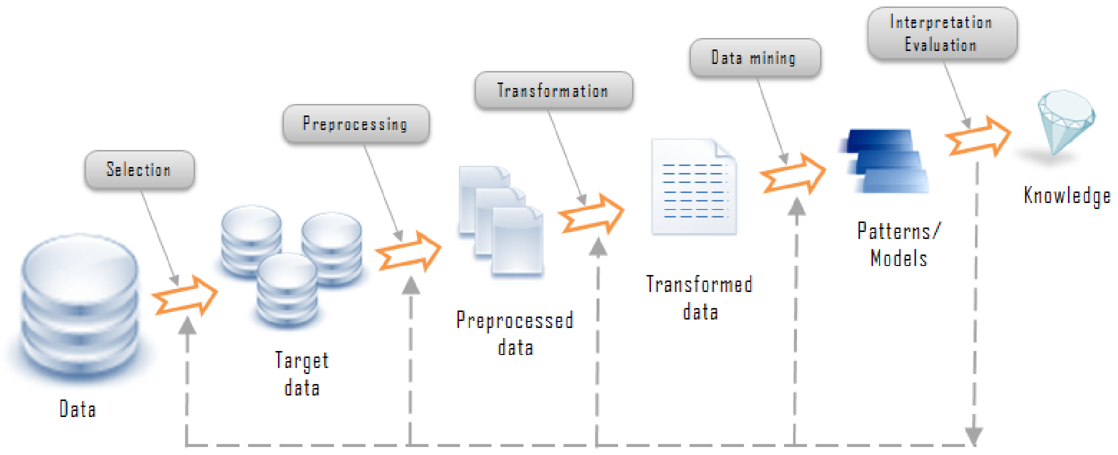

FIGURE 2.1: Data Mining Process

Let us examine the knowledge discovery process in the diagram in Figure 2.1 in details.

- Data coming from variety of sources is integrated into a single data store called target data..
- Data then is preprocessed and transformed into standard format.
- The data-mining algorithms process the data to the output in form of patterns or rules.
- Then those patterns and rules are interpreted to new or useful knowledge or information scope.

The ultimate goal of knowledge discovery and data-mining process is to find the patterns that are hidden among the huge sets of data and interpret them to useful knowledge and information. As described in process diagram above, data-mining is a central part of knowledge discovery process.

2.1.2 Graph Mining

Graphs have become increasingly important in modeling complicated structures, such as circuits, images, chemical compounds, protein structures, biological networks, social networks, the Web, workflows, and XML documents. Many graph search algorithms have been developed in chemical informatics, computer vision, video indexing, and text retrieval. With the increasing demand on the analysis of large amounts of structured data, graph mining has become an active and important theme in data mining. Graphs are everywhere. (Fig: 2.1.2)

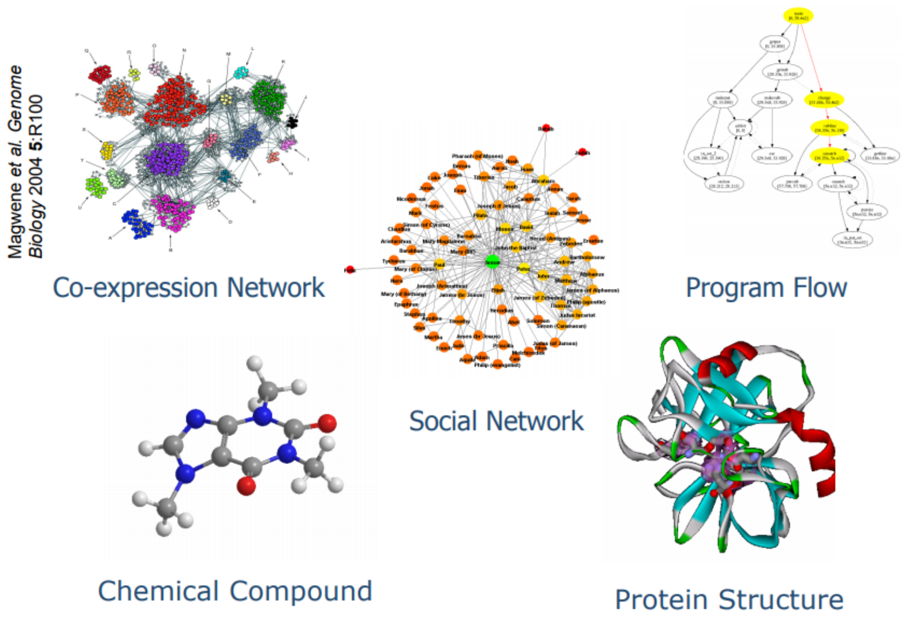

FIGURE 2.2: Graph Mining Process

Among the various kinds of graph patterns, frequent substructures are the very basic patterns that can be discovered in a collection of graphs. They are useful for characterizing graph sets, discriminating different groups of graphs, classifying and clustering graphs, building graph indices, and facilitating similarity search in graph databases.

2.1.2.1 Graph

A graph is a set of vertices and edges, having some labels. Lets illustrate this idea with an example. Consider the graph in Figure 2.2: This graph contains four vertices (depicted as yellow circles). These vertices have labels such as 10 and 11. Labels provide information about the vertices. Labels do not need to be unique. In other words, the same labels may be used to describe several vertices in the same graph. For example, if the above graph represents a chemical molecule, the labels 10 and 11 could be used for all vertices representing Oxygen and Hydrogen, respectively. Now, besides vertices, a graph also contains edges. The edges are the lines between the vertices, here represented by thick black lines. Edges may also have some labels. In this example, four labels are used, which are 20, 21, 22 and 23. These labels represents different types of relationships between vertices. Edge labels do not need to be unique either. Edges can be directed or undirected, weighted or unweighted.

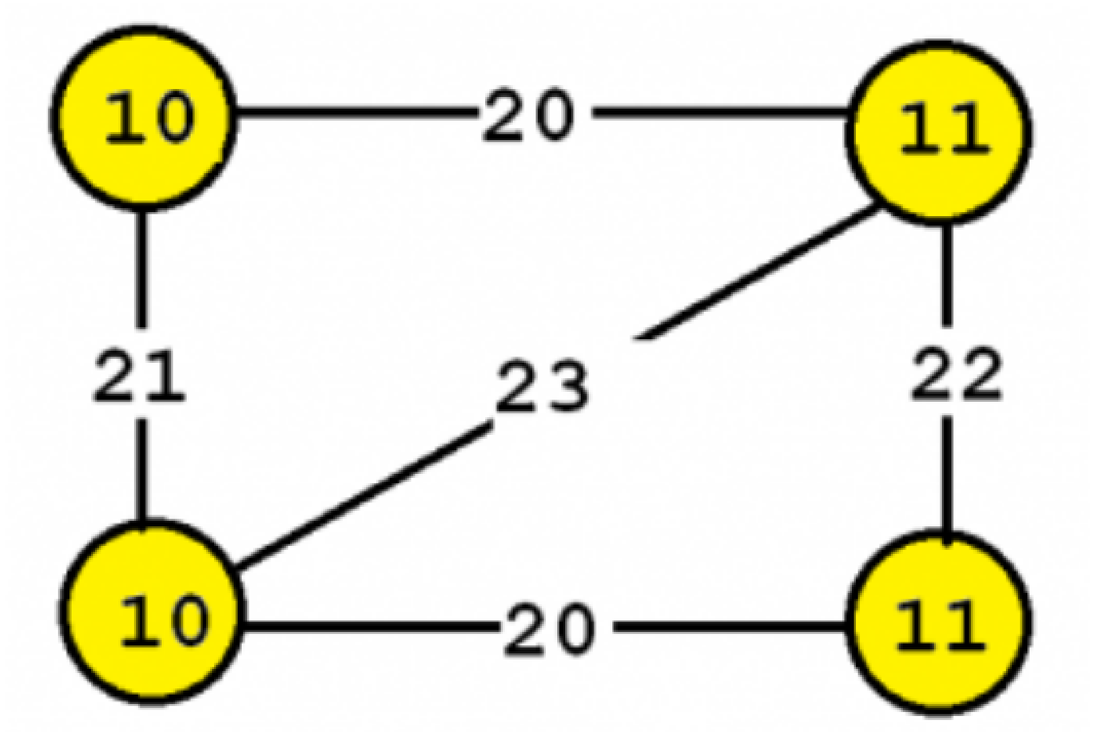

FIGURE 2.3: A Sample Graph

2.1.2.2 Subgraph

A graph G = (V' , E') is said to be a subgraph of graph G = (V, E) if $V' \subseteq V$ and $E' \subseteq E$. This definition of subgraph is a general graph theory concept where a disconnected graph can be a subgraph of a connected larger graph. But in the context of graph mining, we are not particularly interested in disconnected subgraphs. Some applications may require discovering disconnected subgraphs but that issue is out of the scope of this work. A small modification in the definition of subgraph i.e.G must be connected is perfect in this context.

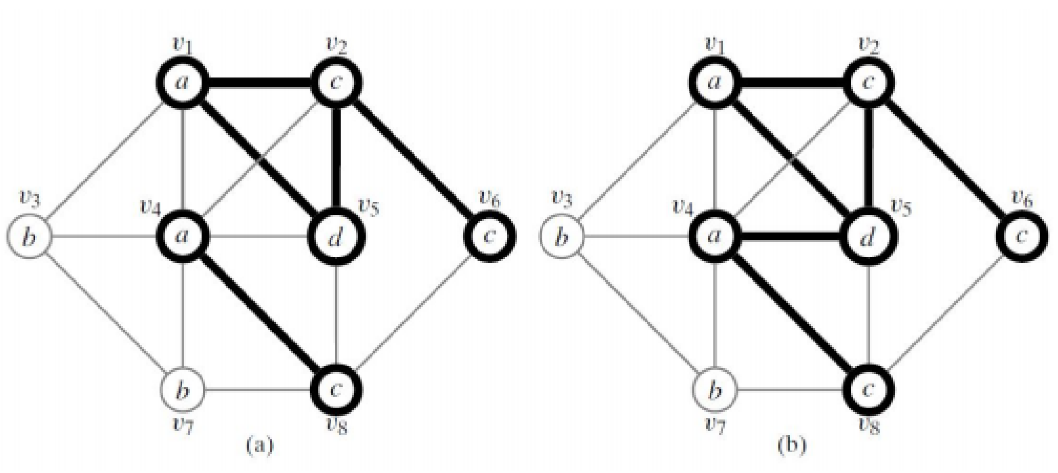

FIGURE 2.4: A subgraph (a) and connected subgraph (b).

2.1.2.3 Graph Isomorphism

A graph G' = (V' , E') is said to be isomorphic to another graph G = (V, E) if there exists a bijective function $\phi : V' \to V$ i.e. both surjective and injective such that: $(a)(u,v) \in E' \iff (\phi(u),\phi(v)) \in E(b)\forall u \in V', L(u) = L(\phi(u))(c)\forall(u,v) \in E', L(u,v) = L(\phi(u),\phi(v))$
In Figure 2.5, G1 and G2 are isomorphic to each other because: $\cdot\phi(u1) = v1 \cdot \phi(u2) = v3 \cdot \phi(u3) = v2$

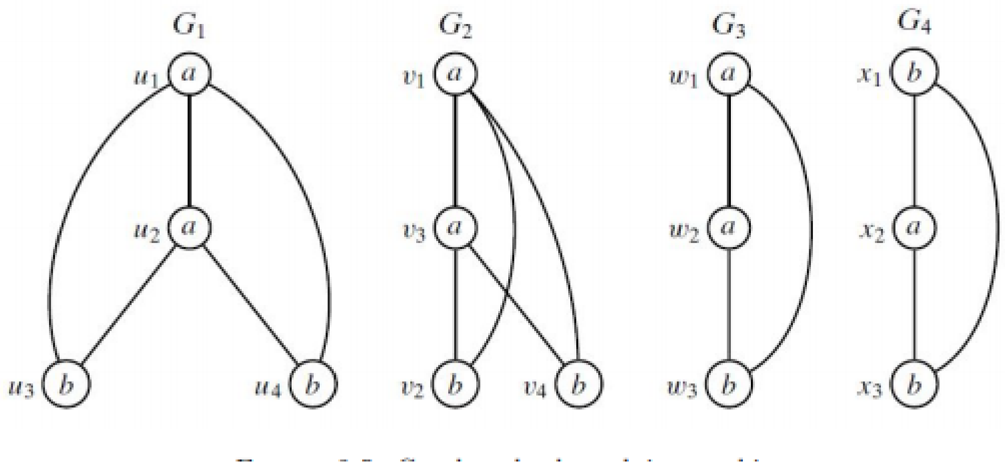

FIGURE 2.5: Graph and Subgraph Isomorphism

2.1.2.4 Subgraph Isomorphism

Though G3 is not isomorphic to G1 in Figure 2.5, we can still find a mapping between G3 and a subgraph of G1. To be more specific, consider the following vertex mapping between G1 and G2:

$$\cdot\phi(u1) = w1 \cdot \phi(u2) = w2 \cdot \phi(u3) = w3$$

and edge mapping:

$$\cdot\phi(u1, u2) = (w1, w2) \cdot \phi(u2, u3) = (w2, w3) \cdot \phi(u1, u3) = (w1, w3)$$

Thus G3 is isomorphic to G' = (V' , E'), a subgraph of G1 where V' = (u1, u2, u3) and E' = (u1, u2),(u2, u3),(u1, u3). We call this type of mapping existence subgraph isomorphism. Formally, if function is only injective but not surjective, we say mapping is a subgraph isomorphism between those two graphs. G3 also has a subgraph isomorphism relation with G2. On the other hand, G4 has subgraph isomorphism mapping with neither G1 nor G2 because of vertex label mismatch.

2.1.2.5 Frequent Subgraph

The goal of subgraph mining is to discover interesting subgraphs appearing in a set of graphs. A subgraph will be considered as interesting if it appears multiple times in a set of graphs. In other words, we want to discover subgraphs that are common to multiple graphs. These frequent subgraphs are useful, for example, to find association between chemical elements common to several chemical molecules.

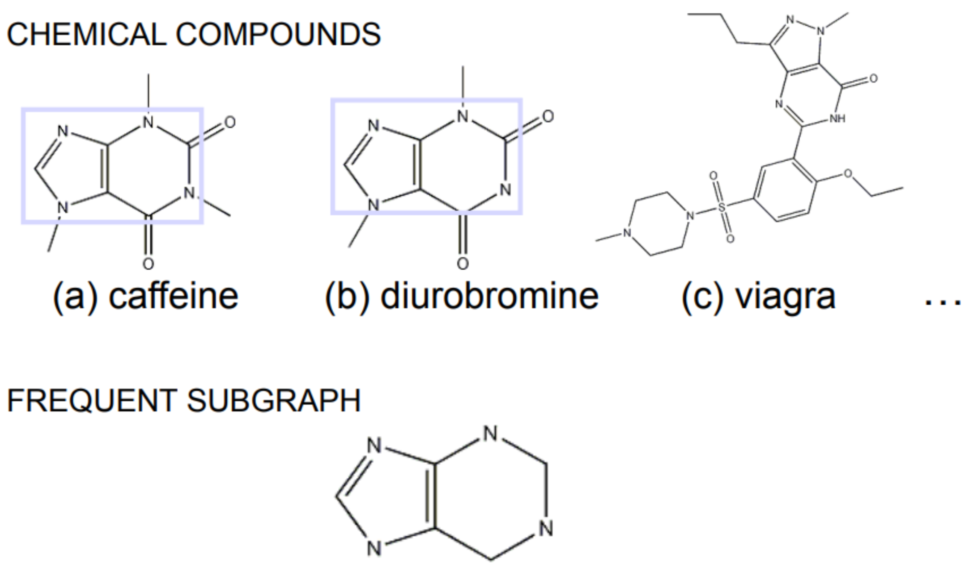

FIGURE 2.6: Frequent Subgraphs(1)

Or frequent subgraphs in program call graphs

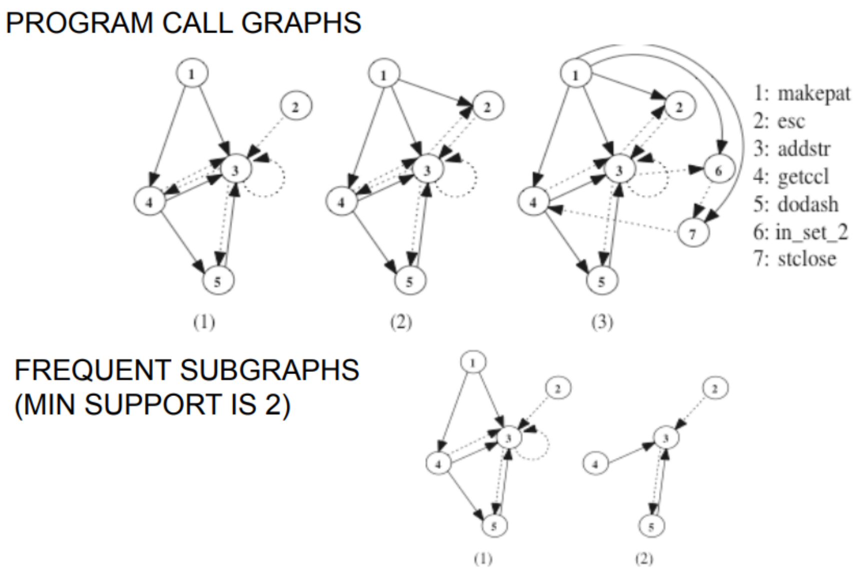

FIGURE 2.7: Frequent Subgraphs(2)

2.1.2.6 Candidate Generation

An effective strategy to enumerate subgraph patterns is the so-called rightmost path extension. Given a graph G, we perform a depth-first search (DFS) over its vertices, and create a DFS spanning tree, that is, one that covers or spans all the vertices. Edges that are included in the DFS tree are called forward edges, and all other edges are called backward edges. Backward edges create cycles in the graph. Once we have a DFS tree, define the rightmost path as the path from the root to the rightmost leaf, that is, to the leaf with the highest index in the DFS order. Consider the graph shown in Figure 2.9(a). One of the possible DFS spanning trees is shown in Figure 2.9(b) (illustrated via bold edges), obtained by starting at v1 and then choosing the vertex with the smallest index at each step. Figure 2.9 shows the same graph (ignoring the dashed edges), rearranged to emphasize the DFS tree structure. For instance, the edges (v1, v2) and (v2, v3) are examples of forward edges, whereas (v3, v1), (v4, v1), and (v6, v1) are all backward edges. The bold edges (v1, v5), (v5, v7) and (v7, v8) comprise the rightmost path.

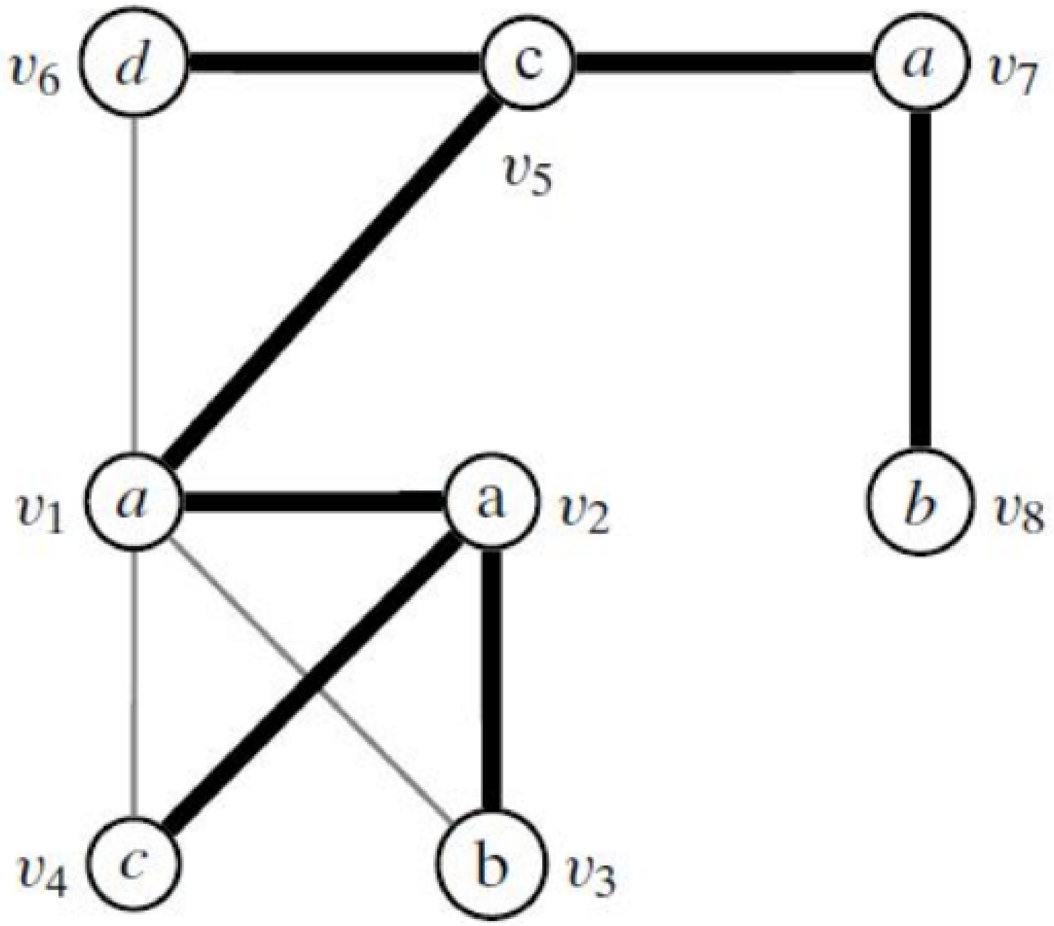

FIGURE 2.8: Rightmost path of depth-first spanning tree of a graph

For generating new candidates from a given graph G, we extend it by adding a new edge to vertices only on the rightmost path. We can either extend G by adding backward edges from the rightmost vertex to some other vertex on the rightmost path (disallowing self-loops or multi-edges), or we can extend G by adding forward edges from any of the vertices on the rightmost path. A backward extension does not add a new vertex, whereas a forward extension adds a new vertex. For systematic candidate generation a total order on the extensions has been imposed as follows: First, all backward extensions from the rightmost vertex will be extended, and then forward extensions from vertices on the rightmost path will be tried. Among the backward edge extensions, if ur is the rightmost vertex, the extension (ur, vi) is tried before (ur, vj) if $i < j$. In other words, backward extensions closer to the root are considered before those farther away from the root along the rightmost path. Among the forward edge extensions, if vx is the new vertex to be added, the extension (vi , vx) is tried before (vj , vx) if $i > j$. In other words, the vertices farther from the root (those at greater depth) are extended before those closer to the root. Also note that the new vertex will be numbered x = r + 1, as it will become the new rightmost vertex after the extension. In Figure 2.9, a complete sequencing of rightmost path extension is shown.

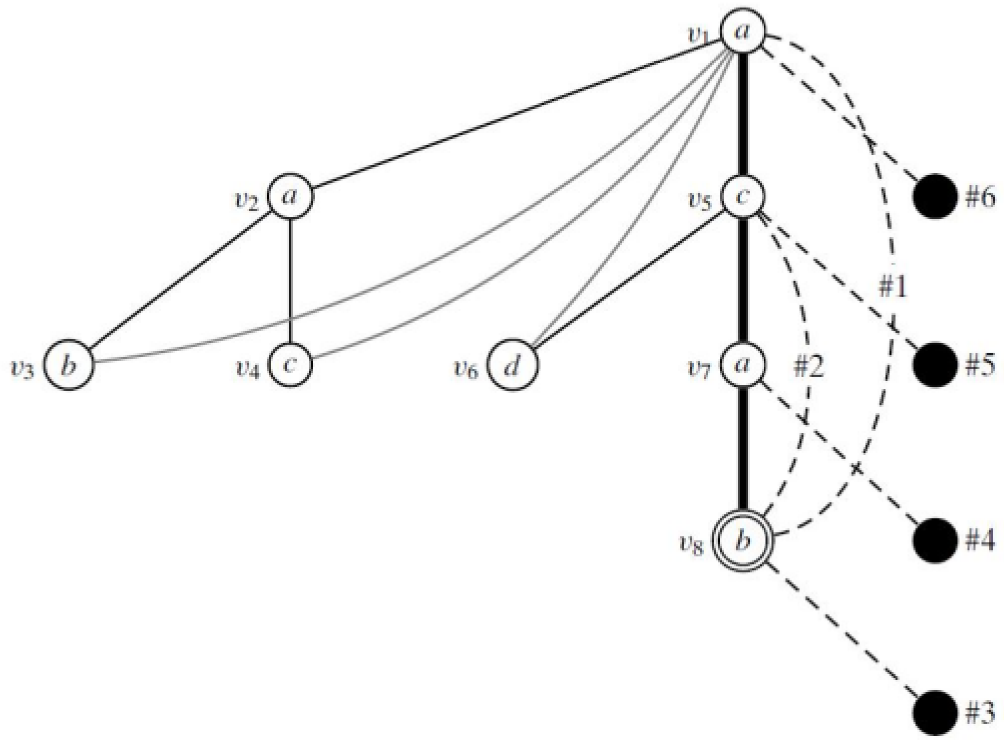

FIGURE 2.9: Rightmost path extension sequence

2.1.2.7 Canonical Code

Rightmost path extension can generate isomorphic graphs via different extensions. Only one of them need to be kept for further extension. Other isomorphic graphs can be safely pruned. Question which one to keep. If we can give ranks to each isomorphic graphs and keep the graph with least rank for further extension, this problem can be solved. Actually it was a proposal of gSpan. To give rank to a graph each edge is represented as an extended tuple of the form: $< vi, vj, L(vi), L(vj), L(vi, vj) >$ Here vi , vj are the discovery time of that corresponding nodes in DFS walk. Thus each DFS walk produces a unique DFScode following the condition that all the backward edges incident with vertex vi are listed before any of the forward edges incident with it. Let G be a graph and let TG be a DFS spanning tree for G. The DFS tree TG defines an ordering of both the nodes and edges in G. The DFS node ordering is obtained by numbering the nodes consecutively in the order they are visited in the DFS walk. We assume henceforth that for a pattern graph G the nodes are numbered according to their position in the DFS ordering, so that i ¡ j implies that vi comes before vj in the DFS walk. The DFS edge ordering is obtained by following the edges between consecutive nodes in DFS order, with the condition that all the backward edges incident with vertex vi are listed before any of the forward edges incident with it. The DFS code for a graph G, for a given DFS tree TG, denoted DFScode(G), is defined as the sequence of extended edge tuples.

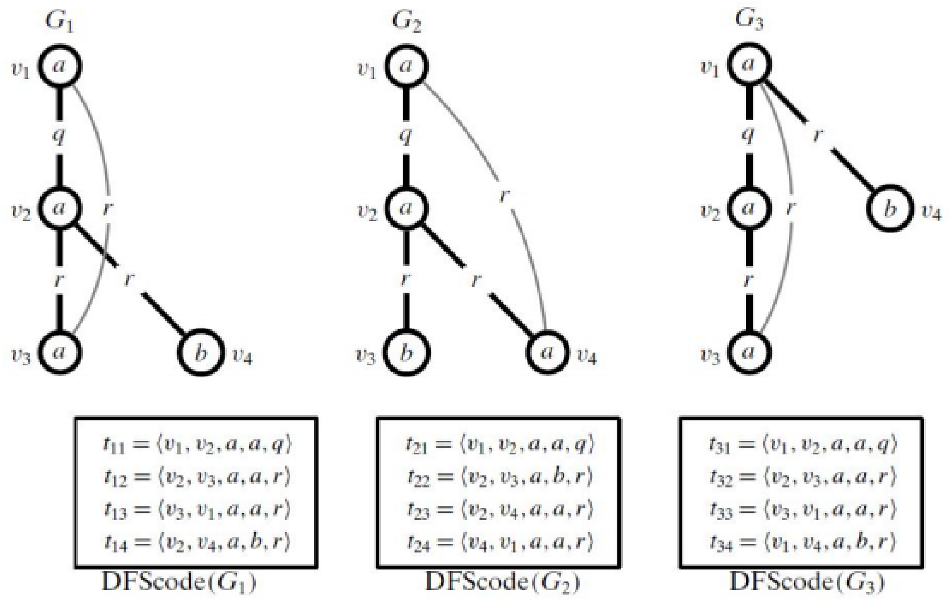
G_1
G_2
G_3
v_1
v_2
v_3
v_4
a
b
q
r
$t_{11} = \langle v_1, v_2, a, a, q \rangle$
$t_{12} = \langle v_2, v_3, a, a, r \rangle$
$t_{13} = \langle v_3, v_1, a, a, r \rangle$
$t_{14} = \langle v_2, v_4, a, b, r \rangle$
DFScode(G_1)
$t_{21} = \langle v_1, v_2, a, a, q \rangle$
$t_{22} = \langle v_2, v_3, a, b, r \rangle$
$t_{23} = \langle v_2, v_4, a, a, r \rangle$
$t_{24} = \langle v_4, v_1, a, a, r \rangle$
DFScode(G_2)
$t_{31} = \langle v_1, v_2, a, a, q \rangle$
$t_{32} = \langle v_2, v_3, a, a, r \rangle$
$t_{33} = \langle v_3, v_1, a, a, r \rangle$
$t_{34} = \langle v_1, v_4, a, b, r \rangle$
DFScode(G_3)

FIGURE 2.10: DFS subscripting

Figure 2.10 shows the DFS codes for three graphs, which are all isomorphic to each other. The graphs have node and edge labels drawn from the label sets P V = a, b and P E = q, r. The edge labels are shown centered on the edges. The bold edges comprise the DFS tree for each graph. For G1, the DFS node ordering is v1, v2, v3, v4, whereas the DFS edge ordering is (v1, v2), (v2, v3), (v3, v1), and (v2, v4). Based on the DFS edge ordering, the first tuple in the DFS code for G1 is therefore $< v1, v2, a, a, q >$. The next tuple is $< v2, v3, a, a, r >$ and so on. The DFS code for each graph is shown in the corresponding box below the graph. A subgraph with smallest DFScode is the canonical. The small relation between two DFScode is defined as: Let C1 = (t11, t12, t13, ..., t1m) and C2 = (t21, t22, t23, ..., t2m) are DFScodes for two isomorphic graph where t is extended tuple format of edge as mentioned before. $C1 < C2$ if and only if $\forall(k < i)t1k = t2k$ and $t1i < t2i$.

This gives birth to a new question: how to compare two edge tuples? Let two DFS tuples be: $t1 =< vi, vj, L(vi), L(vj), L(vi, vj) > t2 =< vx, vy, L(vx), L(vy), L(vx, vy) >$ We say $t1 < t2$ if and only if 1. (vi , vj) Here, $< y$ or (ii) j = y and $i > x$ when eij and exy both are forward edges. $(b)(i)i < x$ or (ii) i = x and $j < y$ when eij and exy both are backward edges. $(c)jx$ when eij is a forward edge and exy is a backward edge. $(d)i < y$ when eij is a backward edge and exy is a forward edge. Consider the DFS codes for the three graphs shown in Figure 2.11. Comparing G1 and G2, we find that t11 = t21, but $t12 < t22$ because $< a, a, r >< a, b, r >$. Comparing the codes for G1 and G3, we find that the first three tuples are equal for both the graphs, but $t14 < t34$ because (vi , vj) = (v2, v4).

2.2 gSpan Algorithm

The gSpan algorithm is designed to reduce the generation of duplicate graphs. It need not search previously discovered frequent graphs for duplicate detection. It does not extend any duplicate graph, yet still guarantees the discovery of the complete set of frequent graphs. Concepts behind gSpan:

- Produces a Depth-first Search (DFS) code for each edge in graphs.
- Edges are sorted according to lexicographic order of codes.
- Yan and Han (Authors of gSpan) proved that graph isomororphism can be tested for two graphs annotated with DFS codes.
- Starting with small graph patterns containing 1-edge, patterns are expanded systemically by the DFS search.
- Employ anti-monotonic property of graph frequency

ALGORITHM 11.1. Algorithm GSPAN

```
  // Initial Call: $C \leftarrow \emptyset$
  GSPAN ($C$, D, $minsup$):
1 $\mathcal{E} \leftarrow$ RIGHTMOSTPATH-EXTENSIONS($C$, D) // extensions and
     supports
2 foreach $(t, sup(t)) \in \mathcal{E}$ do
3    $C' \leftarrow C \cup t$ // extend the code with extended edge tuple $t$
4    $sup(C') \leftarrow sup(t)$ // record the support of new extension
     // recursively call GSPAN if code is frequent and
        canonical
5    if $sup(C') \geq minsup$ and ISCANONICAL ($C'$) then
6       GSPAN ($C'$, D, $minsup$)
```

gSpan[1] enumerates patterns in a depth-first manner, starting with the empty code. Given a canonical and frequent code C, gSpan first determines the set of possible edge extensions along the rightmost path (line 2). The function RIGHTMOSTPATHEX-TENSIONS returns the set of edge extensions along with their support values, $\in$. Each extended edge t in leads to a new candidateDFS code $C0 = C \cup t$, with support sup(C) = sup (t) (lines 3–4). For each new candidate code, gSpan checks whether it is frequent and canonical, and if so gSpan recursively extends C0 (lines 5–6). The algorithm stops when there are no more frequent and canonical extensions possible.

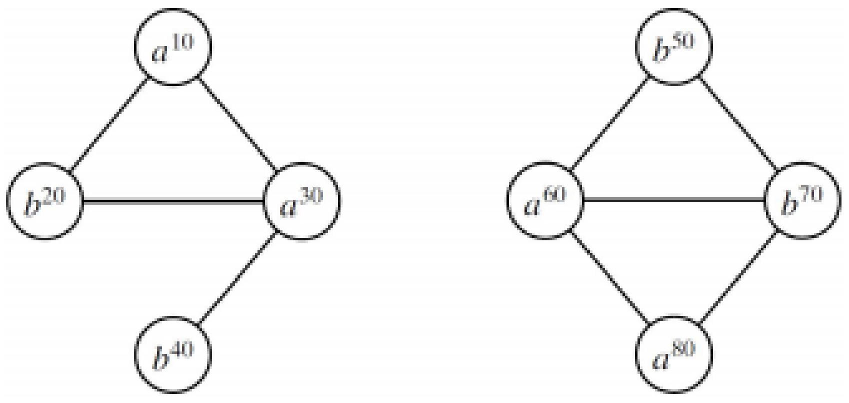

FIGURE 2.11: Database for gSpan Simulation

Example 2.1. Consider the example graph database comprising G1 and G2 shown in Figure 2.11. Let minsup = 2, that is, assume that we are interested in mining subgraphs that appear in both the graphs in the database. For each graph the node labels and node numbers are both shown, for example, the node a10 in G1 means that node 10 has label a.

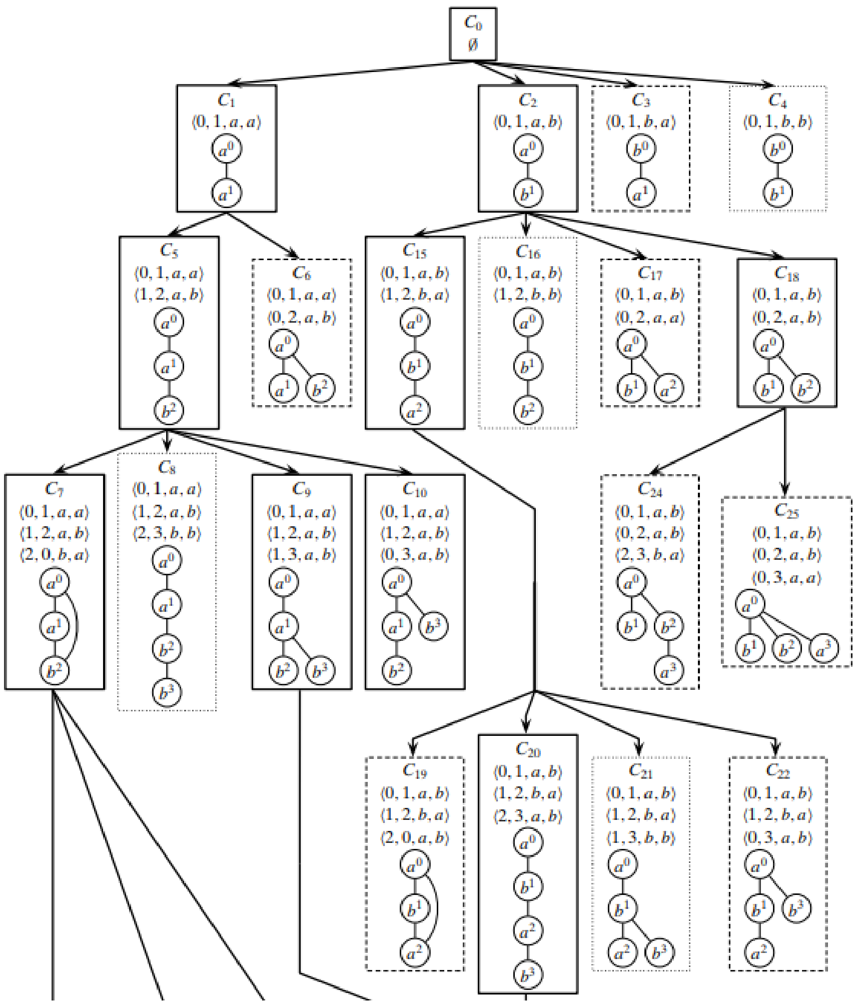
C_0 $\emptyset$
C_1 $\langle 0,1,a,a\rangle$
C_2 $\langle 0,1,a,b\rangle$
C_3 $\langle 0,1,b,a\rangle$
C_4 $\langle 0,1,b,b\rangle$
C_5 $\langle 0,1,a,a\rangle$ $\langle 1,2,a,b\rangle$
C_6 $\langle 0,1,a,a\rangle$ $\langle 0,2,a,b\rangle$
C_{15} $\langle 0,1,a,b\rangle$ $\langle 1,2,b,a\rangle$
C_{16} $\langle 0,1,a,b\rangle$ $\langle 1,2,b,b\rangle$
C_{17} $\langle 0,1,a,b\rangle$ $\langle 0,2,a,a\rangle$
C_{18} $\langle 0,1,a,b\rangle$ $\langle 0,2,a,b\rangle$
C_7 $\langle 0,1,a,a\rangle$ $\langle 1,2,a,b\rangle$ $\langle 2,0,b,a\rangle$
C_8 $\langle 0,1,a,a\rangle$ $\langle 1,2,a,b\rangle$ $\langle 2,3,b,b\rangle$
C_9 $\langle 0,1,a,a\rangle$ $\langle 1,2,a,b\rangle$ $\langle 1,3,a,b\rangle$
C_{10} $\langle 0,1,a,a\rangle$ $\langle 1,2,a,b\rangle$ $\langle 0,3,a,b\rangle$
C_{24} $\langle 0,1,a,b\rangle$ $\langle 0,2,a,b\rangle$ $\langle 2,3,b,a\rangle$
C_{25} $\langle 0,1,a,b\rangle$ $\langle 0,2,a,b\rangle$ $\langle 0,3,a,a\rangle$
C_{19} $\langle 0,1,a,b\rangle$ $\langle 1,2,b,a\rangle$ $\langle 2,0,a,b\rangle$
C_{20} $\langle 0,1,a,b\rangle$ $\langle 1,2,b,a\rangle$ $\langle 2,3,a,b\rangle$
C_{21} $\langle 0,1,a,b\rangle$ $\langle 1,2,b,a\rangle$ $\langle 1,3,b,b\rangle$
C_{22} $\langle 0,1,a,b\rangle$ $\langle 1,2,b,a\rangle$ $\langle 0,3,a,b\rangle$

FIGURE 2.12: gSpan Example

Figure 2.12 shows the candidate patterns enumerated by gSpan. For each candidate the nodes are numbered in the DFS tree order. The solid boxes show frequent subgraphs, whereas the dotted boxes show the infrequent ones. The dashed boxes represent non-canonical codes. Subgraphs that do not occur even once are not shown. The figure also shows the DFS codes and their corresponding graphs. Figure 11.8 shows the candidate patterns enumerated by gSpan. For each candidate the nodes are numbered in the DFS tree order. The solid boxes show frequent subgraphs, whereas the dotted boxes show the infrequent ones. The dashed boxes represent noncanonical codes. Subgraphs that do not occur even once are not shown. The figure also shows the DFS codes and their corresponding graphs.

The mining process begins with the empty DFS code C0 corresponding to the empty subgraph. The set of possible 1-edge extensions comprises the new set of candidates. Among these, C3 is pruned because it is not canonical (it is isomorphic to C2), whereas C4 is pruned because it is not frequent. The remaining two candidates, C1and C2, are both frequent and canonical, and are thus considered for further extension. The depth-first search considers C1before C2, with the rightmost path extensions of C1 being C5 and C6. However, C6 is not canonical; it is isomorphic to C5, which has the canonical DFS code. Further extensions of C5 are processed recursively. Once the recursion from C1 completes, gSpan moves on to C2, which will be recursively extended via rightmost edge extensions as illustrated by the subtree under C2. After processing C2, gSpan terminates because no other frequent and canonical extensions are found. In this example, C12 is a maximal frequent subgraph, that is, no super-graph of C12 is frequent.

This example also shows the importance of duplicate elimination via canonical checking. The groups of isomorphic subgraphs encountered during the execution of gSpan are as follows: $\{C2, C3\}, \{C5, C6, C17\}, \{C7, C19\}, \{C9, C25\}, \{C20, C21, C22, C24\}$, and $\{C12, C13, C14\}$. The remaining codes are pruned.

2.3 WFSM-MaxPWS Algorithm

To reduce candidate subgraph generation, we propose (i) a tight pruning condition MaxPWS for weighted frequent subgraph mining and (ii) an algorithm called WFSM-MaxPWS for Weighted Frequent Subgraph Mining by using the MaxPWS condition.

2.3.1 Canonical Ordering Subgraph

For edge-weighted graphs, we add weight-property in the extended edge-tuple representation used in gSpan. Consequently, the new extended tuple for an edge (u, v) is of the following form as a 6-tuplet:

$$\_disu, disv, L(u), L(v), L(u, v), W(u, v)\_$$

Where W(u, v) is the weight of edge (u, v). To give a rank of a DFScode of subgraph, the WFSM-MaxPWS canonical order gives the highest priority to (disu, disv) as in gSpan. The second highest priority is given to edge weight W (u, v). The higher the weight, the smaller is the tuple for the same discovery times. The third priority is given to a lexicographic comparison on node and edge label trio. To elaborate, let

$$t1 = \_vi, vj, L(vi), L(vj), L(vi, vj), W(vi, vj)\_;$$

$$t2 = \_vx, vy, L(vx), L(vy), L(vx, vy), W(vx, vy)\_.$$

Then, $t1 < t2$ if and only if,

1. $(vi, vj) < e(vx, vy)$; or

2. (vi, vj) = (vx, vy) and $W(vi, vj) > W(vx, vy)$; or

3. (vi, vj) = (vx, vy) and W(vi, vj) = (vx, vy) and

$\_L(vi), L(vj), L(vi, vj)\_ < l\_L(vx), L(vy), L(vx, vy)\_.$

Here, $< e$ is an ordering on edge, and $< l$ is an ordering on vertex and edge labels. Note that $< l$ follows lexicographic order. The edge order $(< e)$ rule is derived from the rightmost path extension sequence. The edge that extended earlier is smaller.

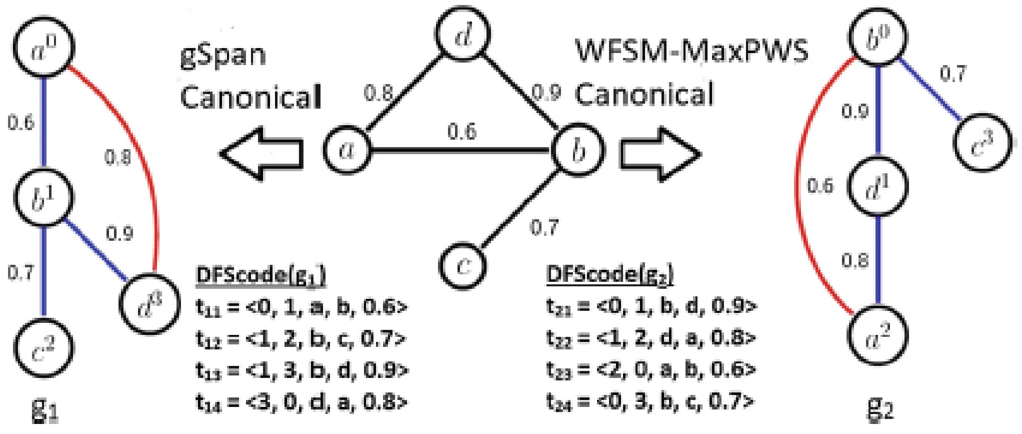
gSpan Canonical
WFSM-MaxPWS Canonical
a^0
b^1
c^2
d^3
0.6
0.8
0.9
0.7
g_1
d
a
b
c
0.8
0.9
0.6
0.7
b^0
d^1
a^2
c^3
0.7
0.9
0.6
0.8
g_2
DFScode(g_1)
t_{11} = <0, 1, a, b, 0.6>
t_{12} = <1, 2, b, c, 0.7>
t_{13} = <1, 3, b, d, 0.9>
t_{14} = <3, 0, d, a, 0.8>
DFScode(g_2)
t_{21} = <0, 1, b, d, 0.9>
t_{22} = <1, 2, d, a, 0.8>
t_{23} = <2, 0, a, b, 0.6>
t_{24} = <0, 3, b, c, 0.7>

FIGURE 2.13: Canonical code comparison with gSpan (without edge label)

Figure 2.13 shows a comparison between the canonical representations in gSpan and our WFSM-MaxPWS algorithm. For simplicity and readability, tuples are shown as 5-tuplets by omitting the edge labels L(u, v). As gSpan puts an edge ordering during the mining time of endpoint nodes and compares the (node labels-edge label)-trio in lexicographic order, tuple t11 becomes the smallest for graph in the figure. On the other hand, after performing edge ordering during the node mining time, the second importance of our WFSM-MaxPWS is put on edge weight. The higher the weight, the smaller is the tuple. As the first tuple of any DFScode has node discovery pair (0, 1), the highest weighted edge would be the smallest. So, WFSM-MaxPWS would consider tuple $\_0, 1, b, d, 0.9\_$ as the smallest. DFScode with other edges as first tuple would not be canonical.

Lemma 1. No tuple can have a weight higher than the weight of the first tuple in a canonical WFSM-MaxPWS DFScode.

2.3.2 MaxPWS Pruning Technique

We divide the entire graph database into partitions p1, p2, p3, . . . , px, . . . , pM. Partition p1 is the set of graphs in the database having the minimum number of edges; partition pM is the collection of graphs having the maximum number of edges. Graphs in the same partition have the same number of edges. Each graph in partition px contain at least one more edge than any graphs in partition px–1 and contain at least one fewer edge than graphs in partition px+1.We called these partition-collection Edge-Class (EC). For the sample graph database in Fig. 2, there are three partitions. Hence, $EC = \{p1 = \{G1\}4; p2 = \{G3, G4\}6; p3 = \{G2\}7\}$, where the subscript after each curly brackets indicates the edge count of the partition. For any subgraph of consideration, we calculate its occurrence list, which is a collection of subsets from each Edge-Class partition entry where each member graph of the collection is a superset for that subgraph.

The resulting algorithm (which uses MaxW) is called MaxPWS-gSpan algorithm, which can be considered as a variant of our WFSM-MaxPWS algorithm (which uses FTW).

Lemma 2. MaxPWS-measure is anti-monotonic.

Corollary 1. If MaxPWS(g) $< \tau$, then g has no potential weighted frequent extension. —

Corollary 2. Due to MaxPWS-measure $\leq$ MaxW-measure, MaxPWS pruning technique prunes more unnecessary patterns.

2.3.3 The WFSM-MaxPWS Algorithm

A pseudocode for WFSM-MaxPWS[2] is shown in Algorithm 1. Here, our WFSMMax-PWS algorithm takes the following four input parameters: (i) the canonical DFScode of a graph C, (ii) graph database D, (iii) weighted support threshold τ , and (iv) occurrence list OLC of C. With the initial call, $C = \phi$ and OLC = EC (edge class). WFSM-MaxPWS puts all frequent weighted subgraphs in a result set.

Algorithm 1. Algorithm *WFSM-MaxPWS*

```
1: procedure WFSM-MaxPWS($C, D, \tau, OL_C$)
2:     $OL\_vec$ = rightmost-path-extension($C, OL_C, D$)
3:     for each ($t, OL_t$) $\in OL\_vec$ do
4:         $C' = C \cup t$
5:         if $IS\_CANONICAL(C')$ = false then continue.
6:         Set $sup_{C'} = 0$ and $MaxPWS = 0$
7:         for each $q_i \in OL_t$ in reverse order (last to first) do
8:             $m_i$ = edge count of $q_i$
9:             $sup_{C'} = sup_{C'} + |q_i|$    //cardinality of $q_i$
10:            $PWS_i = \frac{|C'| \times W(C') + (m_i - |C'|) \times FTW}{m_i} \times sup_{C'}$
11:            if $MaxPWS < PWS_i$ then $MaxPWS = PWS_i$
12:        $wsup_{C'} = W(C') \times sup_{C'}$
13:        if $wsup_{C'} \geq \tau$ then
14:            $result = result \cup C'$    //Enlist $C'$ as frequent weighted subgraph
15:            WFSM-MaxPWS($C', D, \tau, OL_t$)
16:        else if $MaxPWS \geq \tau$ then WFSM-MaxPWS($C', D, \tau, OL_t$)
```

The function rightmost-path-extension in line 2 enumerates all possible extensions on rightmost path of the given DFScode and returns a vector of those extensions as new edge tuples and their corresponding occurrence list OL vec. Each entry in OL vec is then checked for canonicity (lines 4 & 5). The support of code C_ is updated while iterating through each entry in OL (line 9). Simultaneously, possible weighted support from the highest to the lowest OLM(qi) is calculated to find MaxPWS (lines 8–11). The actual weighted support wsup of the code/subgraph is calculated in line 11. If it satisfies weighted support threshold condition (line 13), then it is enlisted as a frequent weighted subgraph (line 14) and sent for further extension (line 15). Otherwise, if MaxPWS satisfies weighted support threshold condition, though it will not be enlisted as frequent weighted subgraph, then it will be sent to the WFSM-MaxPWS algorithm for further extension (line 16) as its extended graph still has a chance to be frequent weighted subgraph.

2.4 Applications of Graph Mining

- Program control flow analysis
- Mining biochemical structures
- Finding biological conserved subnetworks
- Finding functional modules
- Intrusion network analysis

- Mining communication networks
- Anomaly detection
- Mining XML structures Building blocks for graph classification, clustering, compression, comparison, correlation analysis, and indexing

Chapter 3

Out Proposed Framework

Recent years have encountered massive growth in malwares which poses a great threat to modern computers and internet security. Existing malware detection systems are confronting with unknown malware variants. Recently developed malware detection systems investigated that the diverse forms of malware exhibit similar patterns in their structure with minor variations. So, it is required to discriminate the types of features extracted for detecting malwares. So that potential of malware detection system can be detect unfamiliar program as malware.

3.1 Data Description

The aim of this dataset is to learn structural information on unknown files, based on the call graphs, in order to classify them in two categories: malware or goodware. A call graph is an abstract object produced by program analysis tools to represent the relationships between subroutines in a computer program. Each node represents a function and each edge represents a call from a function to another.

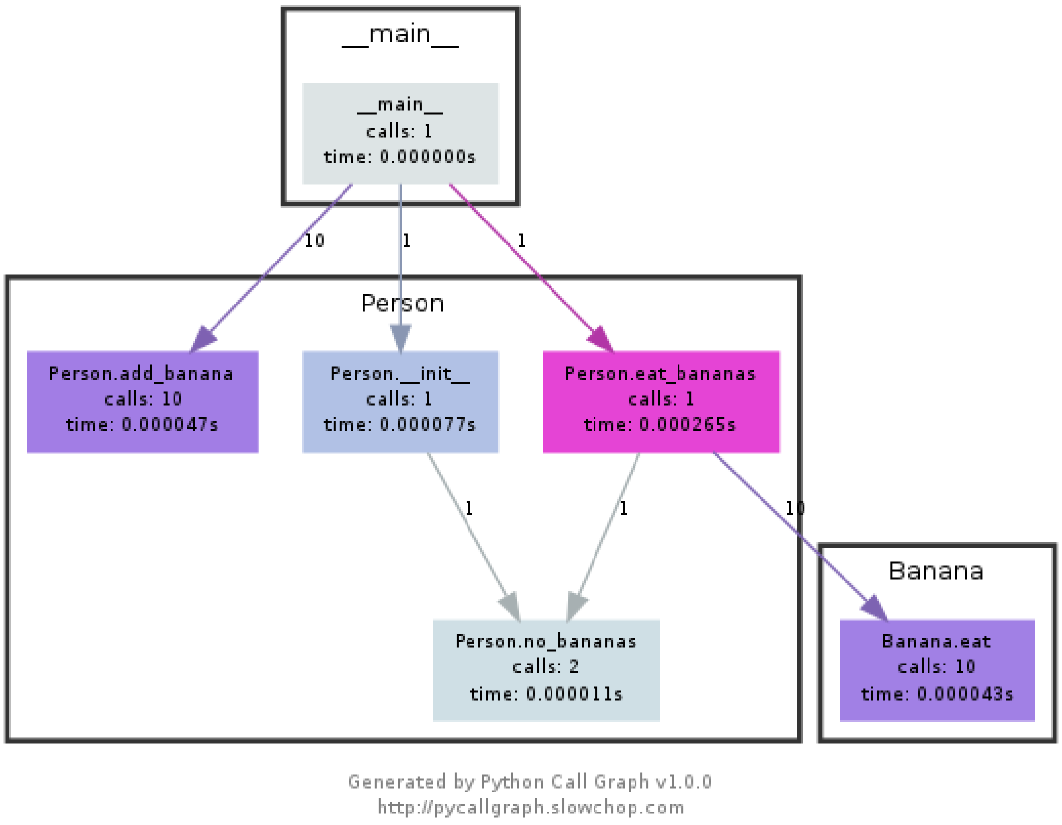

FIGURE 3.1: Call Graph

The call graphs are organized into two subsets, one for the goodwares (sane files) and a second for the malwares. They are in the form of a list, each element contains two dictionaries:

- One dictionary Node $<->$ Attributes
- One dictionary Node $<->$ Neighbors

This format allows us to access directly each node's neighbors which are commonly used in graph matching and classification.

Here are some descriptive statistics for this dataset:

Goodware:

- Number of graphs : 546
- Mean number of nodes : 648.1
- Mean degree : 3.3

- Median degree : 2.7
- Maximum degree : 10.1
- Number of isolated nodes : 130812
- Mean of isolated nodes : 239.6
- Number of self-loops : 0

Malware:

- Number of graphs : 815
- Mean number of nodes : 871.5
- Mean degree : 3.6
- Median degree : 3.7
- Maximum degree : 34.4
- Number of isolated nodes : 231990
- Mean of isolated nodes : 284.7
- Number of self-loops : 0

Dataset[3]

3.1.1 Extraction Method

The dataset has 1361 PE (portable executable), 546 are considered as sane files and 815 as malicious files. Call graphs are extracted from these binaries with the help of radare2. Call graphs contain a lot of information. In order to treat them from a statistical point of view, it was reformatted the attribute of each node, which represents a function, to summarize relevant information.

Each node were counted the number of calls to the most frequent opcode:

['mov', 'call', 'lea', 'jmp', 'push', 'add', 'xor', 'cmp', 'int3', 'nop', 'pushl', 'dec', 'sub', 'insl', 'inc','jz', 'jnz', 'je', 'jne', 'ja', 'jna', 'js', 'jns', 'jl', 'jnl', 'jg', 'jng'].

Extraction Method[4]

In computing, an opcode is the portion of a machine language instruction that specifies the operation to be performed. For example if the node's attributes are: ['mov': 3, 'cmp': 4] there are 3 mov operations, 4 cmp operations and none other from the previous list.

3.2 Our Proposed Framework

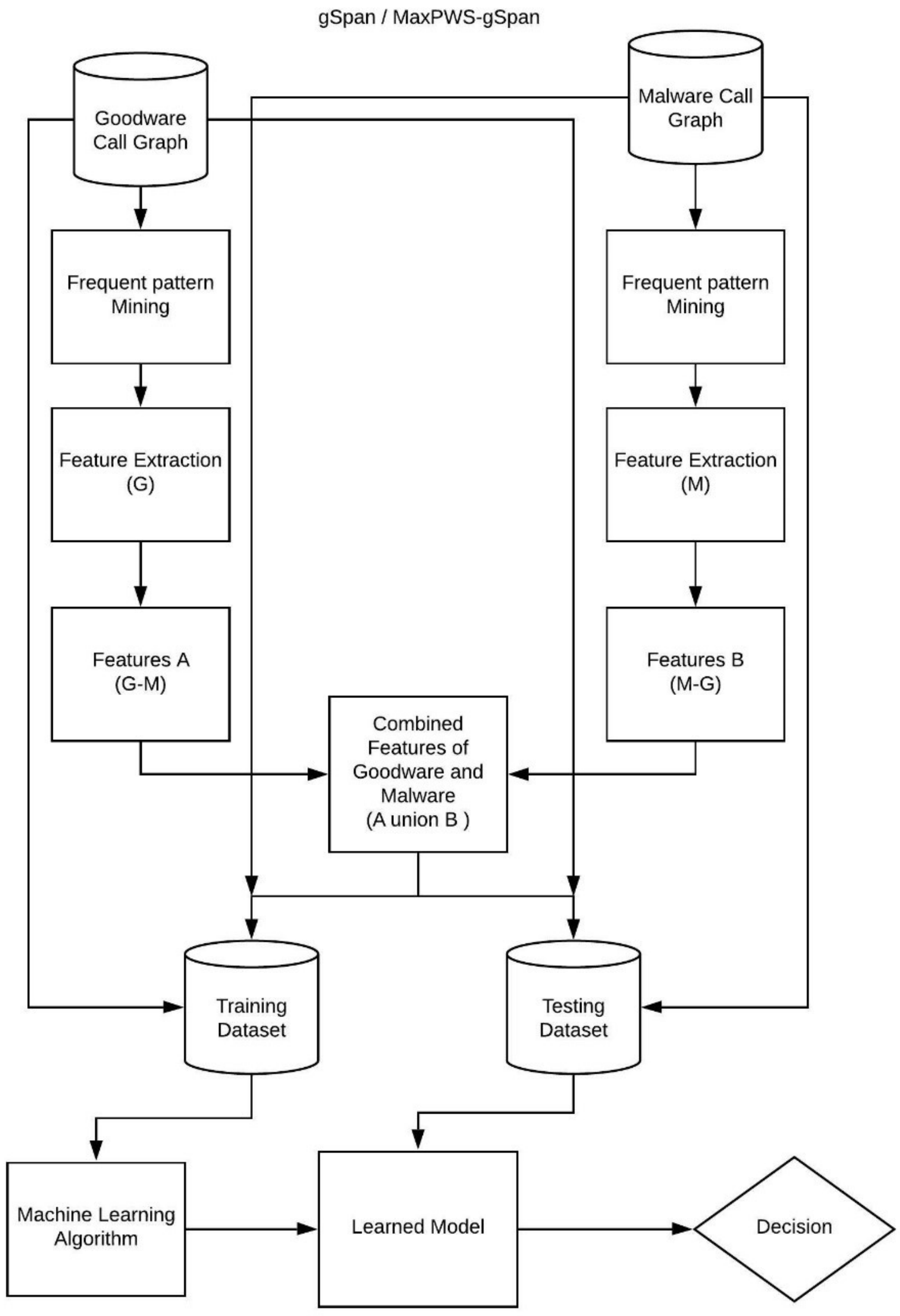

FIGURE 3.2: Flow Chart of Our Proposed Framework

Step 1:

First we took the Goodware call graph dataset and Malware call graph dataset and applied gSpan algorithm and MaxPWS algorithms for frequent pattern mining and extract features from them. Lets consider them as sets, so we find the set of features G from goodware dataset and set of features M from malware dataset.

Step 2:

Now some features may appear in both the set G and M. For decision making or detection of malware, these features will not be helpful and could misguide a decision. The features that are present in both G and M set, we have removed it from the both sets and construct two distinct set. i.e. set A=(G-M) and set B=(M-G).

Step 3:

By performing a set union we get the combined features of goodware and malware datasets. This data consists of both features from malware and goodware.

Step 4:

Now training data set and testing dataset is constructed from the combined features, goodware call graphs and malware call graphs using K-fold cross validation for better results.

Step 5:

In the training data we have applied a machine learning algorithm Nave Bayes classifier to classify the data in two classes, goodware and malware. From the testing dataset and the classifier, a model has been formed to take a decision if a software is goodware or malware.

Chapter 4

Implementation Details

This chapter studies the performance of gSpan and MaxPWS algorithm to make comparison and observe the effect of weight constraint in feature extraction. Two datasets consisting Call Graphs, Goodware Call Graphs and Malware Call Graphs were used to extract features from.

First, we applied the gSpan algorithm and MaxPWS algorithm to extract features or frequent patterns from the datasets. At the beginning we tried with 90% minimum support and gradually decreased the support percentage. We decreased the percentage by 10% each time and observed that there were no noticeable difference in the number of patterns found when the minimum support were 60% or 50% in case of goodware call graphs.

4.1 Environment Setup

Environmental setup for performance analysis is given below:

CPU: Intel Core i5-6200 2.40GHz

RAM: 8GB

OS: Windows 10 64-bit

Language: Python 3.5

IDE:Visual Studio

4.2 Finding Individual Features

gSpan Goodware:

Pattern 1
(0, 1, 'L_', 'L_', 'el')
Pattern 2
(0, 1, 'L_', 'L_pushxorcallmov', 'el')
Pattern 3
(0, 1, 'L_', 'L_pushxorcallmov', 'el')
(1, 2, 'L_pushxorcallmov', 'L_callmov', 'el')
Approximately 4500 patterns were found from goodware call graphs when gSpan was used to extract features and 109500 patterns found in MaxPWS algorithm.

MaxPWS goodware:

Pattern 109466
(0, 1, 'L_leaaddcalljmpmov', 'L_xorsubmovaddcallcmp', 'el', 29.0)
Pattern 109467
(0, 1, 'L_leaaddcalljmpmov', 'L_leaaddcallmovsub', 'el', 23.0)
Pattern 109468
(0, 1, 'L_leaaddcalljmpmov', 'L_leaaddcallmovsub', 'el', 23.0)
(0, 2, 'L_leaaddcalljmpmov', 'L_xorsubmovaddcallcmp', 'el', 26.0)
The same approach was also taken for extracting features from malware dataset and 139 features were found applying gSpan algorithm and 2364 were found applying the MaxPWS algorithm.

Some features are present in both goodware and malware and in decision making, they are of no use only making the difference between goodware and malware feeble. So the features those are common in both goodware and malware features we removed it by set subtraction and combined the pure features together for making an dataset consisting feature from both goodware and malware consisting of 4610 patterns in case of gSpan algorithm and 4003 patterns from MaxPWS algorithm.

4.3 Checking Subgraphs

After combining the individual features of Goodware and Malware we used it to detect malwares and goodwares. We used 585 Malware graph and 354 Goodware graph for unweighted features. Then we used all the features for each graph to find out which features are subgraph of goodware or malware. We took 406 Malware graph and 518 Goodware graph for weighted features. We checked subgraph for weighted features also.

4.4 Spliting Train and Test Set

We have weighted test train dataset of 924 x 4003 dimension and 939 x 4610 dimension dataset for unweighted features. We took 70% of data as train set and 30% of data as test set.

4.5 Getting The Result

After training the train set we tested our test set and found 74.11% accuracy for unweighted features and 66.54% accuracy for weighted features. Here are the results of Precision and Recall (1 for Malware and 0 for Goodware) :

Unweighted:

	Precission	Recall	F1-Score	Support
0	0.81	0.46	0.59	113
1	0.72	0.93	0.81	169
micro avg	0.74	0.74	0.74	282
macro avg	0.77	0.69	0.70	282
weighted avg	0.76	0.74	0.72	282

Accuracy ====> 74.11347517730496 %

Weighted:

	Precission	Recall	F1-Score	Support
0	0.62	0.97	0.75	146
1	0.90	0.33	0.49	132
micro avg	0.67	0.67	0.67	278
macro avg	0.76	0.65	0.62	278
weighted avg	0.75	0.67	0.63	278

Accuracy ====> 66.54676258992805 %

Chapter 5

Conclusion

In this research, we analyze two algorithms gSpan and WFSM-MaxPWS for extracting features from Call Graph. We tried to compare the result for malware detection.

5.1 Analyzing Result

After completing the research work successfully we have found some outcomes.

Precision is the percentage of predicting a class correctly.

Recall is the percentage of predicting a classified class correctly.

Our findings from this research :

- To find out the Precision for Malware weighted is better
- To find out the Precision for Goodware unweighted is better
- To find out the Recall for Malware unweighted is better
- To find out the Recall for Goodware weighted is better
- Overall accuracy is better for unweighted approach

5.2 Future Research Scopes

The main challenge regarding the study was time. The extractions of features took a long time execute and while it was executing, it never occupied more than 40% of the computers CPU. The blessing of having a multicore CPU was not utilized in our code so, minor modifications in the algorithms could have been made for reducing time consumption. Research scope for future researchers can be as follows :

- A Multithreaded system could be introduced in the algorithms for faster extraction of feature using more of the idle CPU.
- The study could be done in a distributed system like HADOOP or any other distributed model for better and faster result.
- The common features existing in both goodware and malware were cut regardless their importance in the MaxPWS algorithm. The true weighted feature wasnt determined.

- A Multithreaded system could be introduced in the algorithms for faster extraction of feature using more of the idle CPU.
- The study could be done in a distributed system like HADOOP or any other distributed model for better and faster result.
- The common features existing in both goodware and malware were cut regardless their importance in the MaxPWS algorithm. The true weighted feature wasnt determined.

Result of the study may vary much regarding if deep neural networks were used. The design of the neural network will be a challenge and it should give a better result than Nave Bayes.

Bibliography

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

[2] M. A. Islam, C. F. Ahmed, C. K. Leung, and C. S. Hoi, “Wfsm-maxpws: an efficient approach for mining weighted frequent subgraphs from edge-weighted graph databases,” in *Pacific-Asia Conference on Knowledge Discovery and Data Mining.* Springer, 2018, pp. 664–676.

[3] Neitsa, “Dataset source,” Quarkslab dataset-call-graph-blogpost-material, Feb 2017. [Online]. Available: https://github.com/quarkslab/dataset-call-graph-blogpost-material#ref

[4] S. Ranveer and S. Hiray, “Comparative analysis of feature extraction methods of malware detection,” *International Journal of Computer Applications*, vol. 120, no. 5, 2015.