Centroid Based Efficient Approach for Mining Frequent Sub-graph in a Single Large Graph

Exam Roll: Curzon Hall-591

Registration No: 2013-012-065

Session: 2013-14

A project submitted in partial fulfilment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNIVERSITY OF DHAKA

December 9, 2019

Declaration

Me, Md Mahamudur Rahaman Sajib, hereby, declare that the work presented in this project is the outcome of the investigation performed by us under the supervision of Dr. Suraiya Pervin, Professor, Department of Computer Science and Engineering, University of Dhaka. We also declare that no part of this project has been or is being submitted elsewhere for the award of any degree or diploma.

Countersigned	signature
.....	.....
(Dr. Suraiya Pervin)	(Md Mahamudur Rahaman Sajib)

Abstract

Graphs are very necessay tool to symbolize the complex relationships between objects. It will be very useful to extract various intutitve pattern from graph. In real-world interaction between different objects often influence the graph data. We will consider these kind of interaction as an edge between two nodes in graph. Human brain is intuitive in every field. Human brain tries to gather information in every step in life. By gathering information they try to think intuitively and solve problem of life. Graph is scattered version of pattern. Now problem is how we interpret graph in various fields. Correct extraction and interpretation can give us better understanding and insights. These insights that is extracted from scattered graph lead us to take better decisions. Frequent patterns mining is popular topic in data mining. Lot of researches has been done in this topic. Finding frequent pattern as graph is now a difficult problem. But it will give us better interpretation of co-relation between objects. Maximum databases are now tranactional databases. So finding frequent pattern is not enough now a days. We need to extract more information. Graph mining gives us more intuitive insights in terms of mining frequent patterns. Althogh this is an NP- complete problem to find frequent sub-graph in transactionl databases, there has been some ground breaking stategy which analyse data faster than we thnk. Our proposed algorithm solve the frequent sub-graph mining algorthm. Althoug there has been some work in the past, Our approach is to improve an existing algorithm and make it more efficient. Various experiments and real-life data-set evaluate our approach in terms of runtime complexity.

Acknowledgements

All praise is to the Almighty, who is the most gracious and most merciful. There is no power and no strength except with Him. Our deep gratitude goes to our thesis supervisor, Dr. Suraiya Pervin, Lecturer, Department of Computer Science and Engineering, University of Dhaka, for his proper guidance in our research field. She has shared her expert knowledge gathered from working in the field over an extensive period, and has been an integral support

in our thesis work by constantly keeping updates and urging us to something significant.

We want to thank our families and friends for their unwavering love and support. The opportunities that our parents have made possible for us determines the personalities we have built and the work that we produce today. Lastly, we want to thank the Department of Computer Science and Engineering, University of Dhaka, its faculty, staff, and all other individuals related to the department. The department has facilitated us throughout our undergraduate program and subsequent thesis, and has also formed the base for our future endeavours.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

Data mining is the kernel of finding co-relation between real life objects. In the real world data is scattered. By gathering the information we often discover some hidden patterns. Patterns are the base level of intuition. Human brain is intuitive in every field. Human brain tries to gather information in every step in life. By gathering information they try to think intuitively and solve problem of life. Data is scattered version of pattern. Now problem is how we interpret data in various fields. Correct extraction and interpretation can give us better understanding and insights. These insights that is extracted from scattered data lead us to take better decisions. But knowledge extraction from data is not so easy task. As a human being we often take decisions depending on our valuable insights which we gather from childhood. But in real-world calculative data we have to extract knowledge as mathematical form. This form can be hypothesis, lemma or corollary. But to find these form of knowledge we have to analyze data from different aspects.

Data mining is a process of extracting useful insights from large databases. In short we can say that it is a process of mining information or extracting patterns from large databases. When database is big we can not extract information by just observing the data by hand. We need a proper method which will judge every pros and cons of data. Now finding patterns from data is the fundamental quest of data mining aspects. Patterns lead us to co-relate between objects. Now frequent pattern which occurs often in the database will help us more in terms of co-relation between data and real-world incidents. Thus mining frequent pattern from large database is the burning question in data mining field. Mining frequent pattern

is important whose various application proceeds to various data mining algorithm including co-relation analysis(Chandaka Babi et al. (2017)), clustering(Van Dongen (2000)), classification(Buja and Lee (2001)).

There has been a lot of algorithm for finding frequent pattern in large data-set Such algorithms tends to find the frequent item-sets in the trasactional database. Such database contains pattern of items as transaction. In these transactional database, transaction is considered to be real-life events. In short we can take transaction as activity of real life events. As an example let us consider a computer shop has it's own transactional database. It contains every logging history of their customer. It contains in which date a customer buy their product, and which products are bought by him. We can consider this events a transaction. From this database we can get huge insights of a computer shop and apply this insights in terms of selling strategy. We can get various co-relation based data. For example, There is a ratio that 56% people who buy a computer also buy a a anti-virus software. So this kind of co-relation based data can help a seller to plot his strategy.

But now problem is to mining frequent item-set in the transactional databases. Apriori(Rao and Gupta (2012)) and FP-growth(Fan and Li (2003)) algorithm solves this problem in polynomial time. But in practice pattern can occur in any kind of form. For all kind of pattern, finding frequent pattern in database is not easy as for item-set as pattern. Specially for graph based pattern it becomes an NP-complete problem. Because we need to count frequency and for that reason we need to distinguish difference between two graphs. Now whether two graph is ismorphic or not is an NP-complete problem. For this reason finding frequent graph is an NP-complete problem.

1.1 Problem Definition

We improved an existing algorithm which implement a system that will enable a user to find frequent sub-graph. There is generally two types of problem. There are (i)finding frequent sub-graph in disconnected graphs, (ii)finding frequent sub-graph in single large graph. In first problem, each sub-graph is counted against each graph, that whether that sub-graph has occured or not. But in second problem we need to

count not only existence but also frequency or occurence in a single large graph. In our proposed approach we try to solve second problem.

Our model takes a graph as input and tries to find all of is sub-graph whose frequency value is greater than some predefined threshold. As this is an NP-complete problem, we can divide already existing algorithm in two category. They are iterative apriori based approach and recursive pattern growth based approach. In the first kind of approach ensures that all small sub-graph should be discovered before all large sub-graph. Eventually here small sub-graph helps to find the large sun-graph which are super-graph of those sub-graphs. But in second kind of approach, if we take a graph as node, then all of is super-graph will be descendent of that node in the recursion tree. If er discover a sub-graph, in next step we have to discover all super-graphs of it before finding other graphs. Our model follows recursive approach. There already exist some ground-breaking algorithm like MoFa, FFSM, gSpan, GASTON. Which we will be discussed later. Our proposed approach is based on gSpan algorithm. In our proposed approach we have added some decomposition technique which will solve sub-graph isomorphism test faster than gSpan.

1.2 Motivation

Graph is the most complex pattern in every aspect of real-world. Day by day these aspects of operation in graph is increasing day by day. Queries on graph is going to be more complex in nature. Maximum query based problems for research in graph are NP-complete problem. So they can not be solved in polynomial time. So we are doing some heuristic approach to solve these NP-complete problem. These approach based on the brute-force approach which contain some greedy strategy of removing unnecessary search space. Applying various data-structure to solve these kind problems will make faster procedure in terms of run-time complexity. But it depends a lot how we approach these kind of data structure. So 3 things that motivated us to use a data-structure in gSpan are given below.

1. gSpan uses dfs-code for isomorphism testing. But similar graph can have multiple dfs code. gSpan tries to find the minimum lexicographic dfs code among all-possible dfs codes and claim that identical graph will give same minimum dfs

code. dfs code basically done from the dfs tree of a sub-graph. Finding minimum dfs code is also an NP-complete problem. So here we used centroid decomposition tree based on dfs tree. Then we try to find minmum dfs code form this centroid decomposition based tree.

2. This will reduce the tree height between $log(size(n))$.

3. This approach can be done dynamically. So motivation of our work is to make faster sub-graph isomorphism test, which will faster the whole process because this is the only NP-complete step in the algorithm.

1.3 Challenges

1. Our main challenge is to show improvement in terms of runtime. There will be a lot of parmeters. We have to judge our approach against these parameters.

2. Because of large single graph, Big data is always a major concern in our algortihm. Unrelaiable data can create a huge problem. So we need to ensure reliable data-set has been generated.

3. Test against the real-life data.

4. Soring graph is another major concenerns in our approach.

5. How our proposed algorithm cope up with pruning based CSP model, that is anothe major concerns.

1.4 Organization of the Report

The project report is organized as follows.

Chapter 2, talks abount some preliminary concepts co-related to our algorithm and also show some related work which has been done in the past.

In **chapter 3,** We have proposed our approach and showed some examples how our approach is done.

In **chapter 4,** we have showed some experimental comparison between gSpan and our approach.

In **chapter 5,** we have discussed about further field of improvement of our algorithm.

Chapter 2

Preliminary Concept and Related Works

In this chapter, some preliminary concepts and necessary background of frequent sub-graph mining of a single graph will be discussed. Also some other research workers have worked on similar area on frequent sub-graph mining term and have provided different approach. These approaches will be discussed comparatively in terms of various measuring parameter of computational circumstances.

2.1 Introduction

Researchers often need to make some combined approaches which include different methods of different authors. For this reason there has been some ground-breaking results in this topic. But first of all they have taken some preliminary steps which is related to Basic data mining algorithm. After that they have tried to evolve their work with some ground-breaking optimization technique. These optimization technique optimize in terms of time complexity, memory efficiency, maintaining real time data, maintaining dynamic behaviour of graph, sparse graph, enumeration process, pruning technique in recursion.

2.2 Preliminary Concept

2.2.1 Frequent Pattern Mining

The primary goal of data mining is to extract statistically significant and useful knowledge from data. Frequent patterns are patterns which occur frequently in the database. The data of interest can take many forms: vectors, tables, texts, images, graphs and so on. Data can also be represented by various means. This data of interest corresponds to the co-relation between item-sets in database. Objectives of frequent pattern mining algorithm is to find those patterns or sub-structures which occur in transactional or non-transactional database with minimum support(min_sup) and minimum confidence. Minimum support(min_sup) value is a given minimum threshold value. A pattern or sub-structure will be frequent if it covers the minimum threshold value min_sup.

Thus, frequent pattern mining is the task of discovering all patterns P in a database D with transactions $T_1, T_2, T_3...T_n$ where P appears $min/_sup$ or more times in the transactions.

An example of transactional database is provided in Table.3.1 Here in Table.3.1

Transaction ID	Items
T1	A, B, C
T2	C, D
T3	C, D, E
T4	C, D, F
T5	A, C, D

Table 2.1: Transactional Database

$item\_set,\ T_2 = \{C,\ D\}$ it is clearly shown that occurs 4 times. And also we can find a co-relation between item C and D that item D occurs 4 times with item C.

Frequent pattern mining has been the leading concerns for data mining researchers. There has been a lot of categorical research on this topic. Initial research leading to algorithms such Apriori and Frequent Pattern Growth. These algorithm try to find the frequent patterns in the transactional database as well as find the co-relation between pairwise items.

2.2.1.1 Apriori

Apriori is the pruning based approach for finding frequent unordered item-set for different size. This algorithm states the aprior or downward closure property that if $k-$item-set is frequent then all of is subset which size is less than k is also frequent. So it tries to find the frequent item-set in increasing order of it's size. In each step it tries to find the $k-$item-set. If a set with size k is not frequent then all of its super-set will be pruned in the next step.

2.2.1.2 Frequent Pattern Growth

FP-Growth algorithm is an improved version of Apriori algorithm which reduce the number of candidate in the Apriori. FP-Growth follows a divide and conquer strategy. In every step of dividing It tends to create a trie. Which covers all the item-set with a fixed prefix of item-set. items are ordered by the frequency of occurrence. It is one of the fastest and most efficient algorithm discovered as frequent pattern mining algorithm.

2.2.2 Frequent Graph mining

Frequent Graph Mining is finding frequent pattern as graph. In a relational database transaction can be signified by graphs. Where each transaction assigns an edge as relation between two nodes between two items. Pattern of relation occur frequently in the database. Frequent graph mining algorithm tries to find the frequent graph that occurs min_sup times in the database.

2.2.2.1 Frquent Subgraph Mining

There are two kind of problem for FSM : (i) graph transactional based FSM (ii) single graph based FSM. In graph trasactional based FSM transaction is borrowed from association of rule mining.A graph g is considered to to frequent if it's occurence count is greater than some predefined threshold value. In graph transactional based FSM there are different categorical disconnected graphs in the data base. A subgraph is counted once in a single graph that if occurs more than one in the same graph it will be counted once. This signifies whether a subgraph g has occured in a

graph or not. This kind of counting is called *transaction based counting*. Fomally given a database $G = \{ G_1, G_2 G_3 ....G_n\}$. Number of occurence of sub-graph g is defined as $\sigma(g) = \{G_i \mid g : g \subseteq G_i \}$. Another approch of counting occurrence of graph is occurence based counting. Which is used most commonly in single graph based FSM. In occurence based counting we simply count the number of occurence of a sub-graph in a single connected graph. There are some advantages of *transaction based counting* over *occurence based counting*. In transaction based counting downward closure property(DCP) can be maintained to reduce the computaional complexty overhead. Which is basically associated with candidate generation in FSM. In occurence based counting an alternative frequency based measure will be employed to maintain the DCP. On the other way there can be some heuristic which will be adopted to reduce the expensive operation.

2.2.3 Primary Definitions

A graph is defined to be a set of nodes interconnecting with edges between them. Graph that will be used in FSM will be labeled graph. Every vertex will be labled with some key and every edge signifies a relation between these keys.

2.2.3.1 Labeled Graph

A labeled graph is represented by $G = \{ V, E, L_V, L_E \phi \}$, here V represents the set of vertexes, E represents the set of edges between them, L_E represents the set of labels for vertex, L_E represents the set of labels for edges. Here ϕ is a mapping function which defines mapping of $V \rightarrow L_v$ and $E \rightarrow L_E$. A path in a graph represents a sequence of vertexes $V_1 \rightarrow V_2 \rightarrow ....V_i \rightarrow V_{i+1}.... \rightarrow V_n$ where there is an edge between V_i and V_{i+1}. A graph is connected if there is a path between every vertex of the graph. A graph is complete if there will be an edge between every pairwise vertex of the graph.

2.2.3.2 Sub-Graph

two graphs $G_1 = \{V1, E1, L_{V_1}, L_{E_1}, \phi_1\}$ and $G_2 = \{V_2, E_2, L_{V_2}, L_{E_2}, \phi_2\}$ is given. graph G_1 is sub-graph of graph G_2 if every vertex of graph G_1 occur in graph G_2 with the same label and every induced edges between vertices of graph G_1 occur in graph

G_2 with the same label. Formally $V_1 \subseteq V_2$, $E_1 \subseteq E_2$, $\forall(u,v) \in E_1, \phi_1(u,v) = \phi_2(u,v)$ these conditions will be followed.

2.2.3.3 Free Tree

Free tree is a graph that is connected and acyclic.

2.2.3.3.1 Unordered Labeled Tree

Unordered Labeled tree is a free tree which is labeled and every child of a parent node is unordered. It is an acyclic graph $G = \{V, E, L_V, L_E, \phi\}$, $E \subset V \times V$, $\phi(v_i, v_j) \rightarrow L_E$. Here V is the set of vertices and E is the set of edges. There is a special node v_r which is the root. Every node in the tree has a path from the root node v_r. $depth(v_i)$ is the distance of node v_i from root v_r. For each edge $(v_i, v_j) \in E$ if $depth(v_i) < depth(v_j)$ then v_i will be the parent of v_j, otherwise v_j will be the parent of v_i. Vertexes that share the same parent are $siblings$. $deg(v_i)$ is the number of child under vertex v_i. A vertex $deg(v_i) = 0$ is called leaf (Sebastian et al. (2004)).

2.2.3.3.2 Ordered Labeled Tree

A labelled ordered tree (an ordered tree, for short) is a labelled unordered tree but with a left-to-right ordering imposed among the children of each vertex(Sebastian et al. (2004)).

2.2.3.3.3 Sub-Tree

tree $T = \{V, E, L_V, L_E, \phi\}$ contains sub-tree underneath each vertex. A sub-tree is a tree under-neath any vertex of the original tree. Basically if we cut a tree from any vertex and take that vertex as root then we will get a sub-tree of that tree(Sebastian et al. (2004)).

2.2.3.3.4 Lattice

given a database σ, A lattice is a formal form of search space associating the database for finding frequent sub-graph. Here every vertex contains a connected sub-graph. In the lower level of the model there is empty sub-graph as vertex and in the upper level of the model there is a single node which contain the full graph as the node. A node p is parent of node q in the model if q is sub-graph of p and which

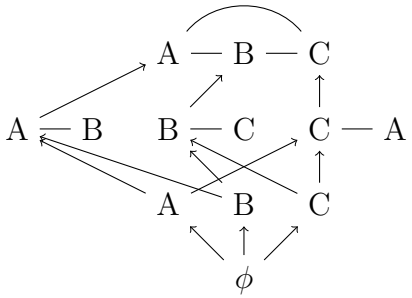


Figure 2.1: Candidate Generating Lattice

differs from exactly one edge. all subgraphs occuring the graph will also be referred in the lattice as a node. In short lattice is a graph of sub-graphs of the single graph database, which covers all the sub-graph of the database(Wörlein et al. (2005)).

2.2.3.4 Graph Isomorphism

Main core part of FSM is graph isomorphism as well as sub-graph isomorphism. In the worst scenario this problem is considered to be the NP-complete problem. But in practice we can have lots of pruning approch to reduce the search complexity. When a graph is a tree then it is called tree isomorphism. Tree isomorphism can be solved in linear $\mathcal{O}(n)$ if the tree is rooted. But for unrooted tree we need to find the centroid of the tree and make the centroid as root. Now problem is we can have two centroid so we need to check for 2 possible cetroid and check for one whether tree has been matched or not. In terms of sub-tree isomorphism faster algorithm tends to take $\mathcal{O}(k^{1.5}n)$ time complexity in worst case which is proposed by Matula and Chung(1987)(Shamir and Tsur (1999)). Then it is further improved by Shamir and Tsur(1999)(Shamir and Tsur (1999)) with $\mathcal{O}(\frac{k^{1.5}}{logk}n)$. Subgraph isomorphism is the kernel of FSM. A lot of efficient and fast approach has been developed to detect the subgraph isomorphism. Some approches are recursive and some approaches are iterative or enumerative. But main target of sub-graph ismorphism algorithm is to reduce the search complexity. SD algorithm is implemented by a recursive approach which utilizes the distance matrix to reduce the search space(Foggia et al. (2001)). On the other hand Ulmann's algorithm also tends to find a backtracking approach maintaining a carry look-ahead function to reduce the search space. Enumerative algorithms like Nauty algorithm tends to find the canonical form of graph and by matching this canonical form of graph sub-graph isomorphism is done. But

construction of canonical form is an NP-complete problem in worst case(Cordella et al. (2004)). Nauty was considered to be the fastest algorithm in terms of subgraph isomorphism problem. On the other hand VF(Cordella et al. (1999)) and VF2(Cordella et al. (2004)) use depth first search approach to detect subgraph isomorphism. But there is some rule based reliable pruning approaches which assist to reduce the search complexity.

2.2.3.5 Generic Review of FSM

There is two type of generic category to classify FSM algorithm. (i)Apriori based category (ii) Pattern Growth based category. These categories has some common feature but they are also very different in terms of procedure. Apriori based category is enumerating algorithm. In each step it generates subgraph and make a bfs tree of subgraphs Where bfs tree can be called as lattice that has been discussed in section 2.2.3.3.4. In every node of the lattice there will be a distinct subgraph. And in the kth level of the lattice there will be all subgraphs of k nodes which will call as k-graph. Now we will use these k-graphs to generate all possible $(k + 1)$-graphs. Now pruning step is if k-graph is not frequent then every super-graph of that k-graph will not be frequent so we can just prune that part from the bfs tree(Rao and Gupta (2012)). On the other hand pattern growth based approach is based on dfs startegy where for each already found subgraphs we will extend the supergraph until all frequent super-graph are found(Fan and Li (2003)).

Algorithm 2.1: Apriori Based FSM

```
1  Input : G ← a graph, δ ← min_sup;
2  Output : G_1, G_2, G_3, ...G_n a set of frequent subgraphs size <= n;
3  intialize G_1 ← all frequent subgraph of size 1;
4  i ← 2 while G_{i-1} ≠ φ do
5      G_i ← φ;
6      L_i ← candidate_gen(G_{i-1});
7      foreach graph g in L_i do
8          g.count ← 0;
9          foreach G_i ∈ G do
10             if is_isomorphic_subgraph(g, G_i) then
11                 g.count ← g.count + 1;
12             end
13         end
14         if g.count ≥ δ(G) then
15             G_i ← G_i ∪ g
16         end
17     end
18 end
```

This apriori based approach maintains the DCP. This kind of enumerative FSM contains three steps : (i)candidate generation (ii) subgraph detection and support counting (iii) pruing. So basically these kind of FSM researches based on optimizing one of these three steps. Researches often try to find fast way of sub-graph isomorphism detection to find frequent sub-graphs in an optimized way. But another group of researchers rely on the already found sub-graph isomorphism algorithm, they just try to find various pruning condition to reduce the search complexity in practice. Sometimes FSM is identical to FTM(frequent Tree Mining) when we convert a graph to it's canonical form. In our next section we will take look on how we can convert a graph to canonical for and thus can generate various code. This will help us to understand the already existing algorithm and how these work. We also took some review on some hash based data structure to generate a key from a graph's code. Although this kind of data structure is deterministic and have one way key gener-

ation process that we can generate key from graph's code not vice versa but these data-structure can be used to map graph with distinct key in $O(1)$ time on average case.

2.2.3.5.1 Canonical Cover of Graph

Canonical cover is a way of representing a graph. A very common way is to use adjacency matrix or adjacency list. But the problem of using adjacency matrix or adjacency list that same ideantical graph can have different type of representation in terms of labeling. So it can be happened that two identical graph has different kind of adjacency matrix or adjacency list. So we need to have a method by which we generate some key for a graph and we have to ensure that identical graph gives the same key. In adjacency matrix $adj[i][j]$ contains the number of edges between vertex v_i and vertex v_j. So with respect to isomorphism testing we need to have a labeling strategy. One way of getting canonical representation of a graph is to just starighten the matrix and join the matrix's rows and column composing the list of integers in minimum or maximum lexicographical order. canonical labeling's are usually considered to be compressed which is known as deterministic scheme(McKay and Piperno (2014)). There are a lot of canonical representation of graph has been proposed. We will discuss on different kind of canonical representation in our proposed approach and we will show an analytical comparison between these canonical form.

2.3 Related Work

In this section we will take a review on vairous existing FSM algorithms. Then we will try to give a comparison based analysis between different FSM. My proposed approach is related to 4 existing algorithm. They are MoFa(Huan et al. (2003)), GASTON(Nijssen and Kok (2005)). So we will try to give comparison based overview between these 4 algorithms.

2.3.1 MoFa algorithm

MoFa algorithm is based on molecular database. But this can be applicable for any kind of arbitrary graphs. Mofa store all kinds of embeddings including both vertices and edges. Then it tends to generate all kind of fragment based sub-structure. Detection of sub-graph isomorphism is done cheaply by whether sub-graph's embeddings can be formatted as the same way. MoFa uses fragment based local numbering method which reduce the number of embeddings generated from the fragmen. Mofa refines the sequence of a fragment according to node's sequence in which order they have been added in the fragment. a n-node graph is generated from the n-node graph or $(n-1)$-node graph. All extensions of subgraph that has been discovered from the same n-node graph ar ordered according to the node labels and edge labels. This local ordering helps a lot in terms of reducing the search space. Still Mofa needs to use the standard isomorphism test to prune the already discovered fragments[6].

2.3.2 gSpan

gSpan is the most efficient algorithm in modern times. This algorithm use one types of canonical representation to represent a graph. We know every graph has a lot of random spanning trees. By depth first search strategy we can discover all spanning tree. Every different dfs-traversal will provide a different spanning tree. By this dfs-traversal we get a spanning tree and some backedges. From these gSpan generates a dfs-code Which is canonical representation of the graph. But problem is two identical graph can generate different dfs-code. Because same graph can have multiple dfs-codes. gSpan take the lexicographicall smallest dfs-code from all possiblem dfs-code which is called minimum dfs code and claims that 2 identical graph will generate same minimum dfs code. Restriction of refinements is happened in gSpan in two ways.(i) Fragments can only be extended that lie on the righmost path of the dfs tree.(ii) fragment generation is guided by occurence in the appearence list. Since it is not possible by these two rules to prevent isomrphic fragment in the worst case so gSpan takes lexicographically smallest dfs-code for each fragment. gSpan stored occurence list for each fragment by some hashbased data-structure. Sub-graph isomorphism testing must be done on all graphs stored as the dfs-code key value in the hash based data-structure[13]. Full algorithm is given in 2.

Algorithm 2.2: gSpan Algorithm(Yan and Han (2002)

```
1  ) Subgarph_mining(GS, FS, g) {
2  if g ≠ min(g) then
3  |  return
4  end
5  FS ← FS ∪ {g}
6  enumerate g in each graph in GS and count g's childres
7  foreach c(child of g) do
8  |  if support(c >= min_sup) then
9  |  |  Subgraph_mining(GS, FS, c)
10 |  end
11 end
12 }
13 GraphSet_Porjection(GS, FS){
14 sort labels of the vertices and edges in GS by frequency
15 remove infrequent vertices and edges
16 relabel the remaining vertices and edges(descending)
17 S_1 ← all frequent 1 − edge graphs
18 FS ← S_1
19 foreach edge e in S_1 do
20 |  init g with e, set g.DS ← {h|h ∈ GS, e ∈ E(h)}
21 |  Subgraph_mining(GS, FS, g)
22 |  GS ← GS − e
23 |  if |GS| < min_sup then
24 |  |  break
25 |  end
26 end
27 }
```

2.3.3 FFSM

FFSM is called Fast Frequent Subgraph Mining algorithm. Here graph is represented as triangle matrix. Where in the diagonal entry of the matrices there will be node labels and $matrix[i][j]$ will represent the edge labels between node v_i and node v_j. Here canonical representation is formed from the matrix code. Matrix code is created by concatenating rows and columns of the matrix. same graph has different permutation of this matrix code. But by taking minimum lexicographical representation we ensure the isomorphism of sub-graphs. This kind of canonical representation of graph is called Canonical Adjacency Matrix(CAM)(Cordella et al. (2004)) which we have discussed in section 2.3.5.1. When FFSM joins two matrices of fragments for further refinements we can have only 2 new sub-structure as result. There is a rule in FFSM that new added edge may only be added in the last node of a CAM. After generating candidate as refinements FFSM permutes the matrix in possible formation to check whether it is the minimum lexicographic form or not. Various pruning approach has been taken in this steps to remove the unusual state. If the permutation is not the minimum lexicographic form this state can be pruned from the search space. FFSM only registers nodes which has been matched and all edges will be discarded in terms of storing. This is very helpful to speeding up the process because we only need to consider list of embeddings of new fragments which can be calculated from the registered set of nodes(Huan et al. (2003)).

2.3.4 GASTON

Full form of GASTON is Graph /Sequence/ Tree extractiON(Nijssen and Kok (2005)). GASTON stored all embeddings to generate only refinements which actually appear in the procedure and which will significantly improves the process. Main feature of GASTON is that it is an enumerative process. In each step of enumeration it tries to find the paths, and by path formulation it tries to find the free trees and sub-graphs. By enumerating paths and free trees then it proceeds to the cyclic graphs at the end of formulation. In terms of path fragment isomorphism and Free tree isomorphism GASTON uses linear time technic. But only in the last phase where back-edges frequently will be added to the free trees to enumerate cyclic graphs GASTON faces NP-complete problem to check whether that sub-graph has

already discovered or not. GASTON maintains an order of back-edges(added to the free trees in the last phase) and generate cycles which is covered by these back-edges as we know that every back-edge will be conjugated with a cycle. Gaston use two step verification to detect whether a sub-graph is already found or not. These two steps are (i) hashvalues of all free trees are pre-defined sorted (ii)Graph isomorphism test for final duplicate detection.

2.4 Summary

In this chapter, we have discussed different preliminary concepts necessary to understand the developed approach that has been discussed in Chapter 3. These preliminary concept includes graph formalism, sub-graph ismorphism, canonical cover of graph which will be Kernel of Chapter 3. On the other hand We have taken an overview on 4 recent algorithms for FSM. By combining these algorithms and exteding the feature we will try to find an efficient approach of mining frequent sub-graph.

Chapter 3

Our Proposed Approach

In this Section we will describe our proposed approaches. And we will try to show how we improve already existing algorithm for FSM. First we will state the problem and we will try to elaborate some preliminary concepts which will tile the whole algortihm. We will try to discuss all the pros and cons of those small tiles of the algorithm.

3.1 Preliminary Concepts

In this section we will describe or aspects of algorithm and problem statement and our challenges to solve the problem.

3.1.1 Problem Statement

Given a single connected large graph, $G = \{V, E, L_V, L_E, \sigma\}$. Here V represents the set of vertices, E represents set of edges $E \leftarrow V \times V$, L_V represents the set of labels(for an unlabeled graph, $cardinality(L_V) = 1$), L_E represents set of edge labels(for an unlabeled graph, $cardinality(L_E) = 1$). Now we will define a threshold value min_sup. min_sup represents the minimum support threshold value or percentile required for a sub-graph to be frequent. Now we need to find set of sets of graphs $GS = \{F_1, F_2, F_3.....F_n\}$, where F_i represents the set of graphs with exactly i nodes and those graphs occurs at least min_sup times in the graph as sub-graphs.

3.1.2 Research Challenges

Here we will divide our key challenges of research into 3 parts. We know that for FSM first we need to generate cadidate of each fragment this will be our first challenge. After that for every fragment of subgraph we need to do an isomorphism test. This is the key part of our FSM because run time of algorithm depends a lot on this phase. Third part is storing each fragment with a unique code or key. This is the bottleneck function of our algorithm. Because Every time we generate a candidate we have to store the fragment's key value. And last part is there can be a lot of frequent subgraph. To store all the sub-graph will take a lot of space and also it won't show any significant co-relation between sub-graphs and whole graphs. So we have to select the significant frequent subgraphs among them.

3.1.2.1 Problem of Candidate generation

Candidate generation process is an enumerative process. In which we are making a lattice where each node of the lattice will contain a sub-graph as candidate. Candidate generation has multiple layer. Every layer has a depth value which depends on the sub-graph size. $k-$node candidate will occur in the kth layer. Now main process is to generate $(k+1)-$-node candidate from k-node candidate. while generating the $(k+1)-$node candidate there can be some kind situation that k-node candidate is not frequent. That means we can build a stoppage system for that $k-$node candidate and prune the whole sub-tree of the lattice. Now challenge is how we implement the procedure which will maintain the Downward Closure Property(DCP).

3.1.2.2 Problem of Sub-graph Isomorphism

We know that Sub-graph isomorphism test is an NP-complete problem. To determine the graph's frequency number it is necessary to mine isomorphic sub-graphs. sub-graph isomorphism has been described in chapter 2 and we also describe the formal idea of graph isomorphism. We also describe various existing algorithm of sub-graph isomorphism. Now challenge is how we show some modification in the sub-graph isomorphism algorithm and improve the practical run time complexity.

3.1.2.3 Problem of Generating and Storing Key for Each Fragment

We are generating candidate as sub-graph. But we can not store the whole graph in memory. So we need a one to one mapping system that will have some predefined structure. That will generate a unique key for a graph. But problem depends on the reliability of key. We have to ensure that 2 identical graph will generate the same key and different graph will generate different key. we have showed some technique of generating key in chapter 2, section 2.2.3.5.1. In that section we have just recognized the idea but in next section 3.2.1 we will take a brief overview.

3.1.2.4 Problem of Finding Significant Frequent Sub-graph

In our research we will take this challenge as minor challenges. Because our key concept is to speed up the procedure. We are not focusing on finding co-relation between sub-graphs. But we will take some lower-level action to find a method of storing significant subgraphs which are frequent and all other frequent sub-graph except those will be ignored.

3.2 Algorithm Details

We have discussed all related fundamental topics that is the kernel of FSM. Now we will move forward to state the algorithm details and also some basic concept related to our proposed algorithm.

3.2.1 Related Concept For Proposed Approach

In this section we will discuss about some concept which are kernel of our proposed algorithm. Our proposed algorithm is based upon gSpan(Yan and Han (2002)). So we will consider our algortithm as pattern growth based approach. In this section we will discuss about dfs-code as canonical representation and some formal proof. We also discuss on centroid decomposition which will be used to composing the graph before finding the canonical form.

3.2.1.1 Depth First Search Tree

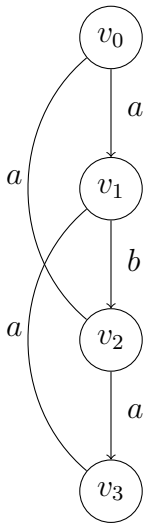
By depth first search traversal we can get the depth first search(DFS) tree. DFS traversal always find a random spanning tree of a graph. There can be different DFS tree for same graph. This depends on the start of DFS traversal and also depend on the choice of similar candidate node for recursive traversal(Tarjan (1972)). DFS tree can be found in linear time. we number's each node by their discovering time. Before traversal we set a global vairable $t \leftarrow 0$ and after traversal of each node we increment the t and assign t's value as discovery time of that node. Given a DFS tree, $T = \{V, E, d\}$ here V is the set of vertices, E is the set of edges, d is a one to one mapping between each vertex with it's discovery time. If $v_i, v_j \in T$ and $d(v_i) < d(v_j)$ then we can tell that v_i is discovered before v_j. Another thing is for each $v_i \in V$, $0 \leq d(v_i) \leq n-1$.

3.2.1.1.1 Forward Edge and Back Edge

Given a tree $G = \{V_G, E_G\}$, $E_G \leftarrow V_G \times V_G$ and also give a tree $G_T = \{V_T, E_T\}$ which is the dfs tree of graph G. So here every edge $e_g \in E_G \cap E_T$ will be the forward edge of graph and every edge $e_g \in E_G / E_T$ will be the back edge. Now every edge $(u, v) \in E_G$ will be ordered in their discovery time.

3.2.1.2 Canonical Labeling and Minimum DFS Code

One way to solve graph isomorphism is to calculate canonical labels of two groups. If two graphs show the same canonical labeling then they are considered to isomorphic. There are a lot of algorithm to compute canonical labeling which we have already discussed in chapter 2. Here in this section we will discuss abount a new canonical labeling system which has been used in gSpan algorithm(Yan and Han (2002)) which is called minimal DFS encoding. DFS code is a sequence of 4 tuples containing edge and 3 labels. A linear order of vertices show a linear order of edges. (i)$(u, v) <_T (u, w)$ if $v < w$, (ii)$(u, v) <_T (v, w)$ if$u < v$, (iii) $e_1 <_T e_2$ *and* $e_2 <_T e_3$ *implies* $e_1 <_T e_3$. So linear order of edges is the DFS codesYan and Han (2002). In figure 3.1 linear order of edges is the $\{(v_0, v_1), (v_1, v_2), (v_2, v_0), (v_2, v_3), (v_3, v_1), (v_1, v_4)\}$. Now dfs code is extended to 4-tuples instead of 2 endpoints of verices(Yan and Han (2002)). A tuple is $t_i = (u_i, v_i, l_{V_{u_i}}, l_{E_{(u_i, v_i)}}, l_{V_{v_i}})$. Here in figure 3.1 suppose ver-



$\{(v_0, v_1), (v_1, v_2), (v_2, v_0), (v_2, v_3), (v_3, v_1), (v_1, v_4)\}$

Figure 3.1: DFS code

tex label is given, $L_V = \{X, Y, X, Z\}$ then Table 3.1 will be the exact order of dfs code. Now problem is that same graph can have multiple dfs tree as well as

Edge Order	Edge Tuple
0	(0, 1, X, a, Y)
1	(1, 2, Y, b, X)
2	(2, 0, X, a, X)
3	(2, 3, X, c, Z)
4	(3, 1, Z, b, Y)
5	(1, 4, Y, d, Z)

Table 3.1: DFS code

multiple dfs code. So for having same code for same graph we need to get the minimum lexicographic dfs code to distingush the graph. Finding minimum dfs code among all possible dfs code is also an NP-complete problem. Now we can tell that if $min(G) = min(H)$, then $G = H$.Yan and Han (2002). Now we will discuss about property of dfs code's parent child relationship. 2 dfs codes are give $\alpha = (a_0, a_1, a_2...a_m)$ and $\beta = (a_0, a_1, a_2...a_m, b)$, then we can consider that α is β's parent and β is α's child. DFS code tree is a tree that is kind of lattice but instead of graph there is dfs code for corresponding graph in node. Here siblings are consistent with DFS lexicographic order. Here property of dfs code tree is (i) node's in nth level of tree contain the dfs code of n-size graph, (ii)it contains the minimum lexicographic dfs code for every sub-graph which are frequent(Yan and Han (2002)).

Algorithm 3.1: Construction of a DFS-code(Yan and Han (2002))

Step 1 :
initialize $i = 0$, *compare* a_0 *with* e;
Step 2 :
if $a_0 > e$ **then**
    select e *as the first element for the DFS code, goto step* 3
if $(a_0, ...a_{i-1})$ *is invalid or* $(a_0, ...a_{i-1}) \leq (a_0, ...a_{i-1}, e)$ **then**
    $i \leftarrow i + 1$
    if $i < n$ **then**
        goto step 2
else
    $(a_0, ...a_{i-1}, e)$ *forms the* $i + 1$ *elements for the DFS code. goto step* 3
Step 3 :
build the remaining elements from $i + 1$ *to* $n + 1$

3.2.1.3 Centroid Decomposition

Centroid decomposition is a concept of decomposing trees. Centroid of a tree is a node if we delete this node from the tree then tree will be sub divided into partial trees whose size will not be greater than the half of size of the actual tree. More Formally given a tree T. Centroid is the node $v_c \in T$. If we pick the node out of the tree, then tree will be sub divided into m trees $\{T_1, T_2, T_3...T_m\}$. Now this is the property of the centroid that $size(T_1), size(T_2)...size(T_m) \leq size(T)$. In Figure 3.2 v_2 is centroid. Because centroid v_2 sub divides the tree into 3 trees that is T_1, T_2, T_3. If we see in Figure 3.3 that $size(T_1) = 1$, $size(T_2) = 2$, $size(T_3) = 1$. So $size(T_1),\ size(T_2),\ size(T_3) \leq size(T)/2$. Now we will move forward to centroid decomposition. Centroid decomposition is a divide and conquer approach. In each step of centroid decomposition, it tries to find the centroid of the tree and then pick the centroid from the tree and sub-divided the trees into more trees. Now for those tree same procedure will go on. After that we will add an virtual edge between those new created tree's centroid and actual tree's centroid. This procedure will go on untill we get a tree, whose size is 0. Formally if v_c is the centroid of a give

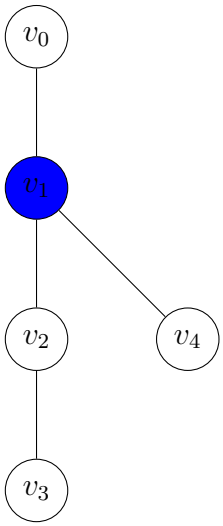


Figure 3.2: Sample Tree for Finding Centroid

Algorithm 3.2: Agorithm For Finding Centroid of The Tree

```
1  FindCentroid(T, v){
2  initilize, max ← 0, bigChild ← null, isCentroid ← true
3  foreach child c of v do
4      T_c ← subtree under child c
5      if size(T_c) > mx then
6          mx ← size(T_c)
7          bigChild ← c
8      if size(T_c) > size(T) then
9          isCentroid ← false
10 if isCentroid then
11     return v
12 else
13     FindCentroid(T_bigChild, bigChild)
14 }
```

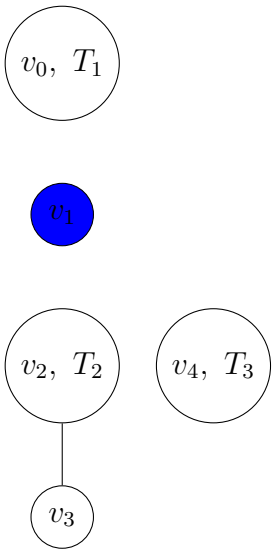


Figure 3.3: Sample Tree for Finding Centroid

tree T, then if we pick the centroid v_c from the tree, then that tree will be divided int $T_1, T_2, T_3..T_m$ new trees. We will do the same procedure for $T_1, T_2, T_3...T_m$ and add edge between v_c and centroids of $T_1, T_2, T_3...T_m$. Every tree can have at most 2 centroids and if we get 2 centroids then there will be an edge between them.

Algorithm 3.3: Centroid Decomposition

```
1 CentroidDecomposition(T, parent_centroid){;
2 v_c ← FindCentroid(T, random_node);
3 add edge between v_c and parent_centroid;
4 foreach child c of v_c do
5 |   T_c ← subtree under child c;
6 |   CentroidDecomposition(T_c, v_c);
7 };
```

3.2.1.4 Hash Based Data Structure

Hashing is a technique to generate integer key from objects by using some mathematical function. This is mapping between an object to an integer key. This mapping is one directional. A good hash function tends to reduce number of collision. That means it reduce the probability that two different object generate same hash value. In our proposed approach we create a hash function which will map dfs-code to some hash-key value. In our proposed approach we have taken a dfs code as string and then we generate hash-key value by polynomial hashing. For example suppose a string is

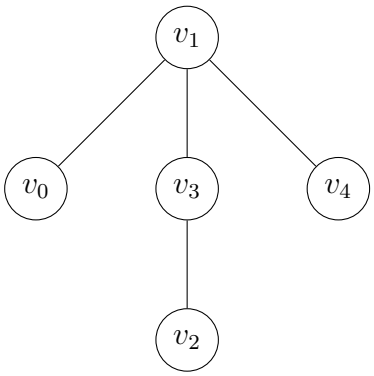


Figure 3.4: Centroid Decomposition of Tree in Figure 3.2

given $S =$ "$abcda$". Suppose our hash $base = 7$ and $bucket\_size = 10^9 + 7$(which is a prime). Then key will be $key = (a \times 7^0 + b \times 7^1 + c \times 7^2 + d \times 7^3 + a \times 7^4) mod 10^9 + 7$.

3.2.2 Algorithm Description

We divide our proposed aproach in 3 parts. (i)Sub-graph Isomorphism, (ii)Storing dfs-code, (ii)pruning.

3.2.2.1 Subgraph Isomorphism part

Our proposed algorithm is gSpan(Yan and Han (2002)) based. Kernel of gSpan is isomorphishm testing through dfs-code as canonical representation of graph. We have already discussed that same graph will generate same lexicographically smallest dfs-code. But the issue is finding lexicographically smallest dfs-code is an NP-complete problem. So gSpan provides a non-deterministic solution. This solution is given in algorithm 3.1. My approach is to first find the dfs tree and then we will do centroid decomposition of that tree and after that we will generate dfs-code on that centroid decomposition. Now here problem is Centroid decomposition is an algorithm which is only able to work for trees. But we will introduce a method from which we can use it to decompose graph also. We have discussed that a tree can have at most 2 centroids. Problem is easy for 1-centroid based tree. But for 2-centroid based tree as there will be an edge between them we will merge the 2 nodes. Thus we can make one centroid. Now we can ensure that identical tree will have same centroid decomposition tree. Here in figure 3.5 leftmost two trees are identical. So they have the same centroid decomposition tree at the rightmost

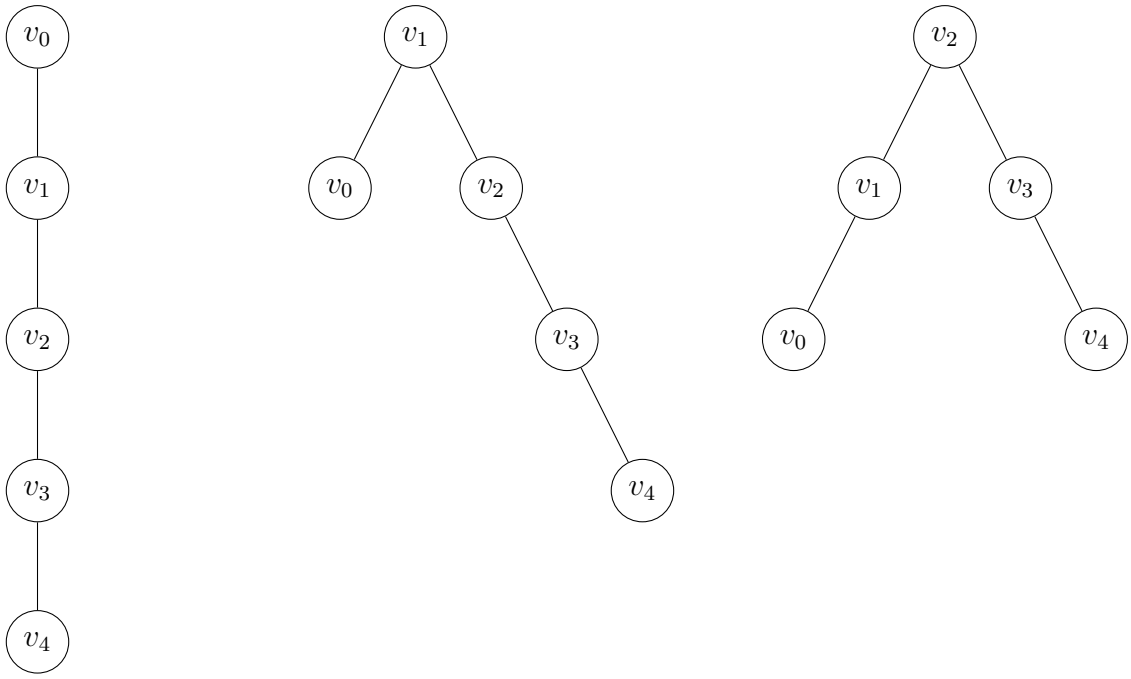


Figure 3.5: Sample Tree for Finding Centroid

side of the figure 3.5. So now the problem is how we decompose a graph in terms of centroid decomposition. We will follow gSpan in this term. We will find a dfs tree and then decompose the tree and then all the back-edges in the dfs tree will be added as usual in the centroid decomposition tree. Once decomposition done we will generate the dfs-code. Main benefit of this decomposition is to get a dfs tree which has at most $log(size(graph))$ levels. Since centroid decomposition is a divide and conquer approach it always divide the tree into at least two sub-tree whose size will be at most half of the size of actual tree. So we can ensure that level of centroid decomposition graph will be at most $log(size(graph))$ levels. Now idea is to find the dfs code on this graph as gSpan has stated. So we have to find the dfs tree for which dfs code of centroid decomposition graph is minimal. In centroid decomposition we can add a node dynamically and update the centroid decomposition graph. So using dfs tree and centroid decomposition tree dont show much more differnce in this case. But in practical scenerio, and if we see algorithm 3.1 where dfs code generation algorithm is stated, we will see that tree's level acts a key role here. If the level is smaller then we can take fast decision that whether we can reach the minmal dfs code for the given state. If not then we will reduce the search space and search for another combination from where we can get minimal dfs code. This

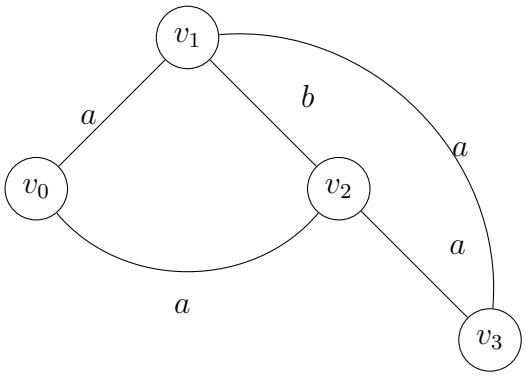


Figure 3.6: Centroid Decomposed Graph of Figure 3.1

solution improve's run time complexity of subgraph isomorphism test.

3.2.2.2 Storing Dfs Code

In each level of dfs code tree, we will get minimum dfs code's in the tree. So we need to generate a integer value for corresponding dfs code. After that we will map that integer key with the number of occurence of that sub-graph. This is the bottle neck of the algorithm. So we have done this $O(1)$ average time complexity by using polynomial hashing. Here to map the hash key with frequency we have used unordered map which also operate in $O(1)$ average time. To do polynomial hash we just concatenate the minimal dfs code into string and after that we have done polynomial hashing as described in section 3.2.1.4.

3.2.2.3 Pruning

We have divided our pruning approach into 2 parts.They are, (i) $s \neq min(s)$ pruning and (ii) partial counting pruning.

3.2.2.3.1 $s \neq min(s)$ Pruning

$if(s \neq min(s))$ is the first line of gSpan's subgraph mining function. This ensures pruning of all DFS codes which are not minimum. It significantly reduces the search space because for this pruning all the descendants of that dfs code will be ignored. Formally pruning is done in 4 steps. If we look at sub-graph mining algorithm in algorithm 3.4 we will understand clearly the steps of pruning here. At the start state of sub-graph mining recursion g will contain only one edge e_0. So any potential child of g won't have any child which contain's smaller edge that e_0 as edge's are selected

in the ascending order of their minimum dfs code. For all newly added back edge $(v_x, v_y), x > y$ should not be smaller than any edge that is connected to v_y. After all these pre-pruning session is done post pruning is applied to the remaining unpruned nodes(Yan and Han (2002)).

Algorithm 3.4: Sub-graph mining algorithm(Yan and Han (2002))

```
1 Subgarph_mining(GS, FS, g) {;
2 if g ≠ min(g) then
3     return;
4 FS ← FS ∪ {g};
5 enumerate g in each graph in GS and count g's childres;
6 foreach c(child of g) do
7     if support(c >= min_sup) then
8         Subgraph_mining(GS, FS, c);
9 };
```

3.2.2.3.2 DCP pruning

In this section pruning is done for maintaing the Downward closure property. If g is frequent in the algorithm 3.4 then we will look forward to it's child. Otherwise we will prune g. Furthure extensions will not be needed. Because g will be the sub-graph of it's child. So if g is not frequent all of it's child will not be frequent.

3.2.3 Complexity Analysis

As our approach is similar to gSpan. So complexity of gSpan will play a vital role here. Sub-graph isomorphism is NP-complete problem. So runtime of gSpan is exponential. But this can be measured by number or sub-graph isomorphism tests. For this reason runtime is bounded by $O(kFS + rF)$(Yan and Han (2002)), where k is the maximum number or sub-graph isomorphishm exists, F is the number of frequent sub-graph, S is the size of the graph, r is the maximum number of already existing codes in the map that proceed from other minmum dfs code. So by multiplying $k \times F \times s$ we enclose the number of isomorphism test in $O(kFs)$(Yan and Han (2002)). rF encloses the maximum number of $s \neq min(s)$ operation. But in our proposed approach this will be $O(rFlog(size(graph)))$. But in practice for using centroid decomposed tree to generate dfs code r will reduce a lot.

3.2.4 Summary

In this section, We have differentiate our approach from gSpan algorithm in terms of sub-graph isomorphism. In gSpan dfs code is generated from a random spanning tree, but here we are generating dfs code from centroid decomposition tree. This will restrict the tree's height between $log(size(graph))$. For this reason we can reduce the search space frequently.

Chapter 4

Experimental Results and Analysis

In this chapter, we will take a review on overall performance between our proposed approach and gSpan. We know that our problem is an NP-complete problem, so we will try to give some comparative data in terms of runtime.

4.1 Intoduction

In recent days maximum research problems are NP-complete problems. Measurin algorithm's complexity against an NP-complete problem is also an NP-complete problem. So we can not just through a complexity analysis out of nowhere. So to evaluate the algorithm's efficiency performance evaluation is necessary. Though my proposed approach show worst performance in terms of theoretical complexity in worst case but on average case my proposed approach show better performance than gSpan. Here we have used chemical compound based dataset. Same dataset hasbeen used in (Yan and Han (2002)). Performance measured taken into account are run time in seconds.

4.2 Experimental Environment

For our experiment, we used C++ language as our primary language and JAVA language as our secondary language. We have used JAVA for the sake of reformatting of data. Whole algorithm is written in C++ language. We have wrote all the necessary codes and tested agains the dataset. We also wrote a bruteforce algorithm to

verify our algorithm. Bruteforce algorithm can judge our prposed approach against the small dataset and show that our approach works fine. Our programs are run in a personal computing machine which has following configuration (i)Porcessor: Intel Core-I3, 3.5 GHz. (ii) RAM: 4 GB DDR3. (iii)Available Storage: 250GB SSD. (iv)Operating System: Windows 10.

4.3 Dataset

A valid data set is needed to test FSM algorithm. We compare our proposed approach with gSapn against synthetic datasets. Syntehtic datasets are generated using similar procedure described in (Sato et al. (2011)). The synthesis data can be measured in terms of 5 parameters. $|D|$ = total number of graph generated, $|T|$ = the average size of graph in terms of edge, I = average size of frequnt subgraph, $|N|$ = number of labels.

4.4 Comparison Between GSpan and Our Approach in Terms of Runtime

In figure 4.1 we have showed some runtime difference against sum of $|D|$. here we have seen that our proposed approch has worked better for large data. In figure 4.2 we have compared our approach against the number of labels. In figure 4.3 we have compared against the average size of subgraphs. We have seen that against every parameter our algorithm works fast for large graphs.

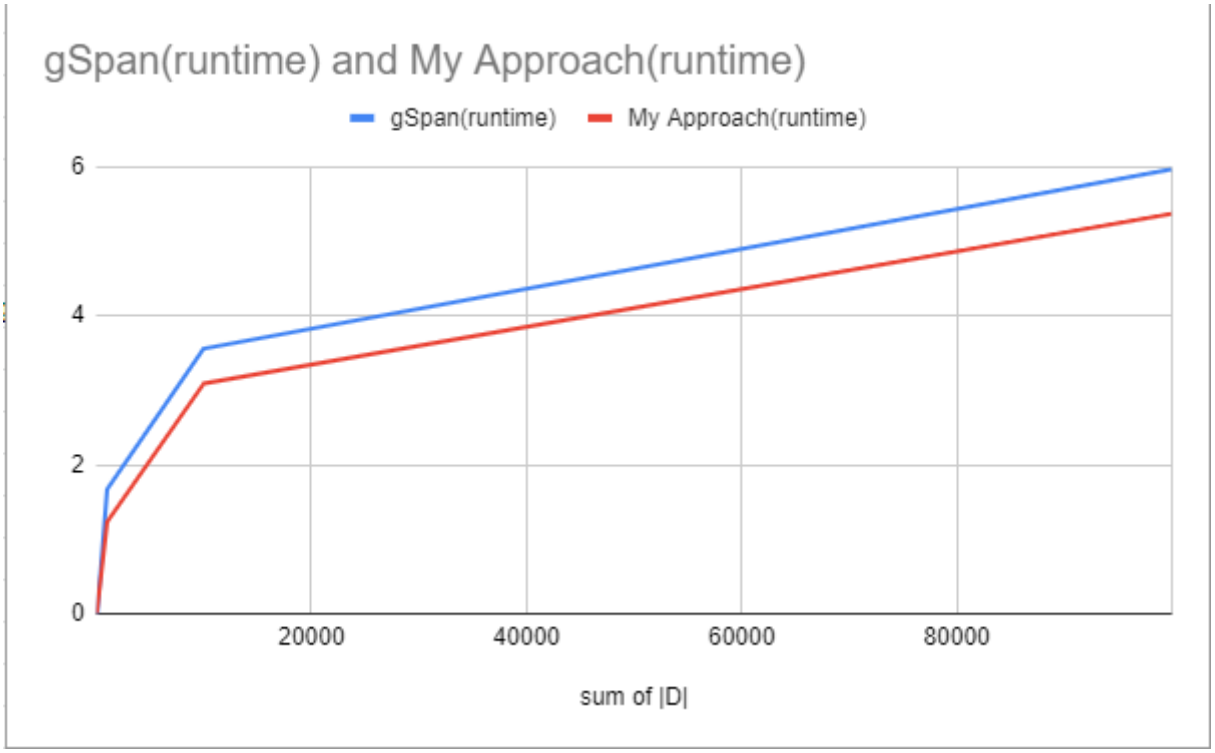


Figure 4.1: Runtime Comparison Against Sum of $|D|$

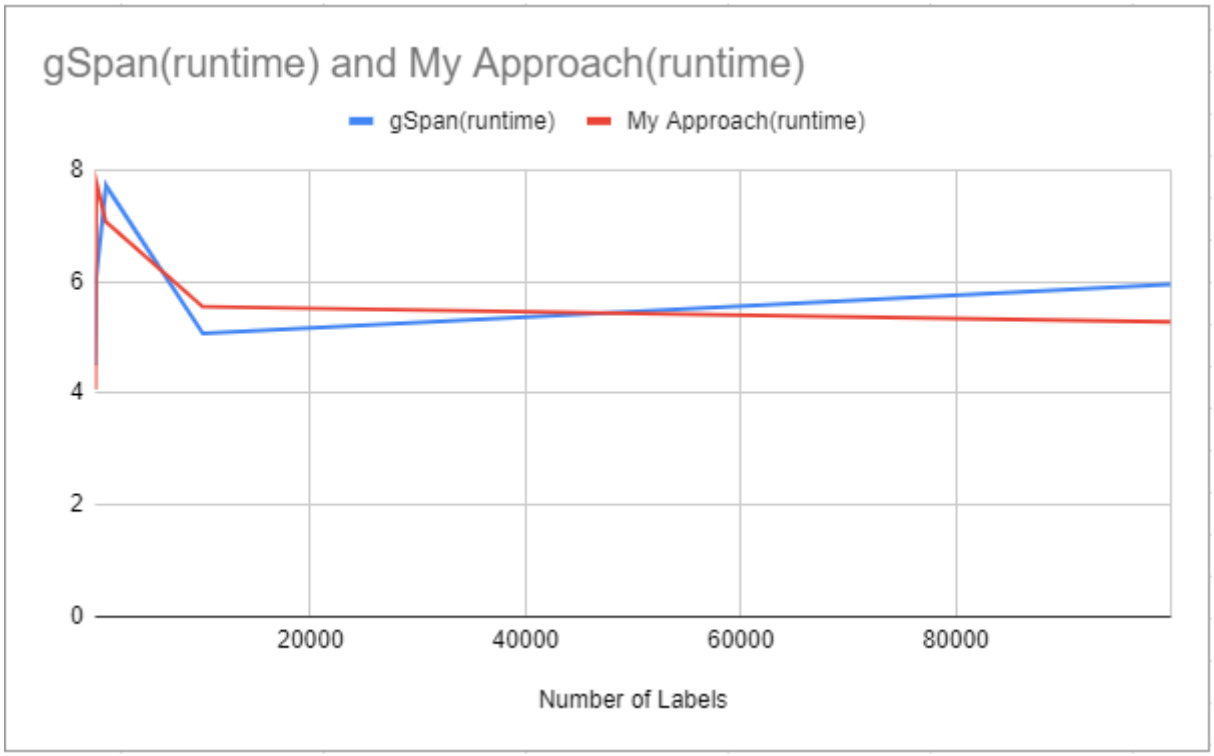


Figure 4.2: Runtime Comparison Against Number of labels

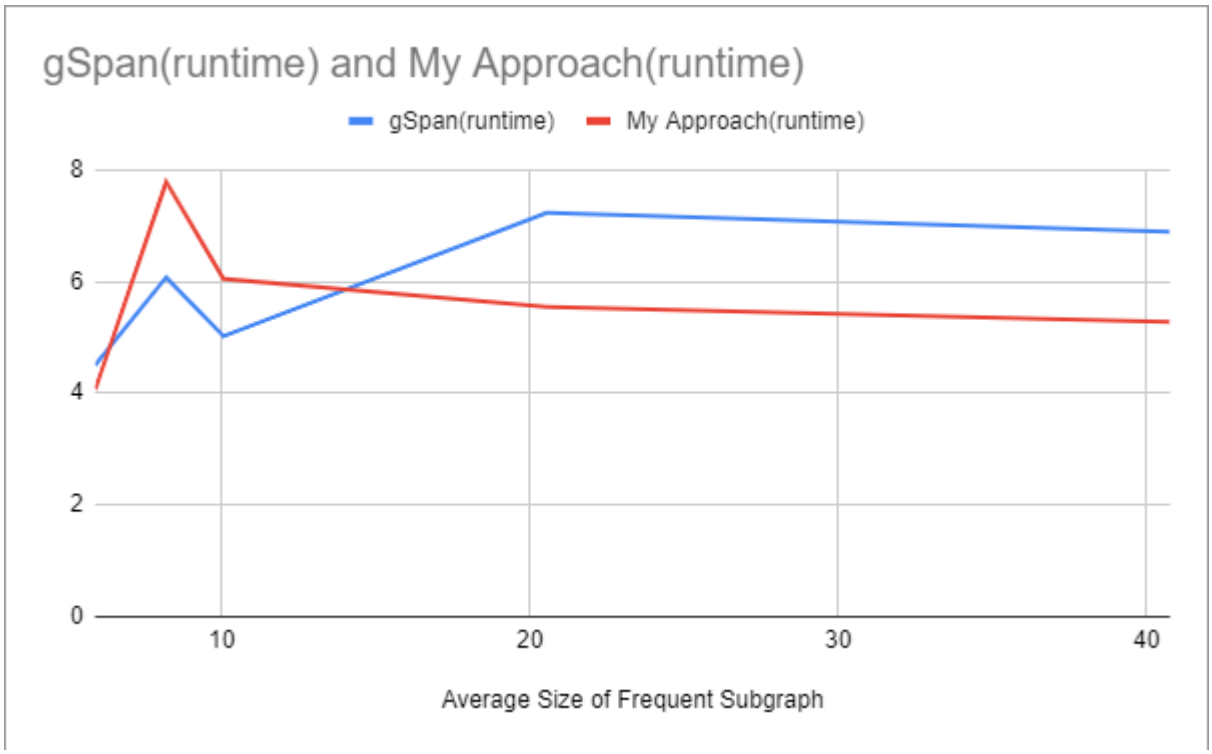


Figure 4.3: Runtime Comparison Against average size of graphs

Chapter 5

Graphs are an important tool than can be used to model complex different types of entities in the various real-world applications. FSM can leads us to discover new information. Thus we can find co-relation between different graph. By this algorithm we can analyse the frequenty occuring graph in social networking website. Our project can also do some ground breaking result in terms of sub-structure finding in checmical substances. In chapter 4 we have showed how fast our algorithm against chemical based database.

5.1 Furthure Research Scopes

1. Our proposed appraoch only try to solve the NP-complete part faster than gSpan and we Also have showed some improvement in terms of storing graph. In furture we wil try to extend our project in terms of findin weighted frequent sub-graph.

2. In our project we have find all the frequent sub-graph. which won't show any kind of co-relation. In futue we will extend our project to find significant and co-relative sub-graph mining.

3. In our project we have used some pruning based approach, which can be extended as a large feature of our project in future.

Bibliography

Andreas Buja and Yung-Seop Lee. Data mining criteria for tree-based regression and classification. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 27–36. ACM, 2001.

Dr Chandaka Babi, Mandapati Venkateswara Rao, and Vedula Venkateswara Rao. Study of association rule mining for discovery of frequent item sets on big data sets. *International Journal of Applied Engineering Research*, 12(22):12169–12175, 2017.

Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. Performance evaluation of the vf graph matching algorithm. In *Proceedings 10th International Conference on Image Analysis and Processing*, pages 1172–1177. IEEE, 1999.

Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

Ming Fan and Chuan Li. Mining frequent patterns in an fp-tree without conditional fp-tree generation. *Journal of computer research and development*, 40(8):1216–1222, 2003.

Pasquale Foggia, Carlo Sansone, and Mario Vento. A performance comparison of five algorithms for graph isomorphism. In *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199, 2001.

Jun Huan, Wei Wang, and Jan Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Third IEEE International Conference on Data Mining*, pages 549–552. IEEE, 2003.

Brendan D McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of Symbolic Computation*, 60:94–112, 2014.

Siegfried Nijssen and Joost N Kok. The gaston tool for frequent subgraph mining. *Electronic Notes in Theoretical Computer Science*, 127(1):77–87, 2005.

Sanjeev Rao and Priyanka Gupta. Implementing improved algorithm over apriori data mining association rule algorithm 1. 2012.

Yuna Sato, Kouji Kuramochi, Takahiro Suzuki, Atsuo Nakazaki, and Susumu Kobayashi. The second generation synthesis of (+)-pseudodeflectusin. *Tetrahedron letters*, 52(5):626–629, 2011.

Thomas B Sebastian, Philip N Klein, and Benjamin B Kimia. Recognition of shapes by editing their shock graphs. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (5):550–571, 2004.

Ron Shamir and Dekel Tsur. Faster subtree isomorphism. *Journal of Algorithms*, 33(2):267–280, 1999.

Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

Stijn Marinus Van Dongen. *Graph clustering by flow simulation*. PhD thesis, 2000.

Marc Wörlein, Thorsten Meinl, Ingrid Fischer, and Michael Philippsen. A quantitative comparison of the subgraph miners mofa, gspan, ffsm, and gaston. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 392–403. Springer, 2005.

Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, pages 721–724. IEEE, 2002.