ANALYZING SOCIAL MEDIA DATA BY FREQUENT PATTERN MINING METHODS

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Engineering

by
Büşra GÜVENOĞLU

July 2018
İZMİR

We approve the thesis of **Büşra GÜVENOĞLU**

Examining Committee Members:

Assoc. Prof. Dr. Orhan DAĞDEVİREN
International Computer Institute, Ege University

Assoc. Prof. Dr. Belgin ERGENÇ BOSTANOĞLU
Department of Computer Engineering, İzmir Institute of Technology

Asst. Prof. Dr. Serap ŞAHİN
Department of Computer Engineering, İzmir Institute of Technology

2 July 2018

Assoc. Prof. Dr. Belgin ERGENÇ BOSTANOĞLU
Supervisor, Department of Computer Engineering
İzmir Institute of Technology

Assoc. Prof. Dr. Yusuf Murat ERTEN
Head of the Department of
Computer Engineering

Prof. Dr. Aysun SOFUOĞLU
Dean of the Graduate School of
Engineering and Sciences

ACKNOWLEDGMENTS

First, I would like to express my sincere gratitude to my advisor Assoc. Prof. Dr. Belgin Ergenç Bostanoğlu for her guidance during the research and writing of my thesis. I would like to thank my advisor for her endless patience, sensitivity, sincerity, and for her continuous support of me with immense knowledge and experience.

Finally, I would like to express my deep gratitude to my brother for endless support and believing in me and also thank to my family for continuous encouragement, love and support throughout my life.

ABSTRACT

ANALYZING SOCIAL MEDIA DATA BY FREQUENT PATTERN MINING METHODS

Data mining is a popular research area that has been studied by many researchers and focuses on finding unforeseen and important information in large dataset. Social media data is one of the most popular and large heterogeneous data collected from social networking sites, microblogs, photo or video sharing sites. Social media represents the entities and their relations. One of the popular data structures used to represent large heterogeneous data in the field of data mining is graphs. The nodes of a graph represent entities and the edges of a graph represent the relations between the entities. So, graph mining is one of the most popular subdivisions of data mining. A frequent pattern is referred to as pattern that is more frequently encountered than the user-defined threshold in a dataset. Frequent patterns in a dataset can give important information about dataset. Using this information, data can be classified or clustered. Frequent patterns can provide different perspective on social media data with respect to sociology, consumer behaviour, marketing, communities.

In this thesis, popular frequent pattern mining algorithms have been examined and it has been observed that most algorithms are not suitable for large datasets. Since data in today's world, especially social networks, has very large data, the existing pattern mining algorithms are not suitable for this data. The aim of this thesis is to implement an existing frequent pattern mining algorithm in parallel manner and to find frequent patterns in a social media data.

ÖZET

SOSYAL MEDYA VERİSİNİN SIK KÜMELER MADENCİLİĞİ YÖNTEMLERİ KULLANARAK ÇÖZÜMLENMESİ

Veri madenciliği, birçok araştırmacı tarafından incelenen ve büyük veri setinde öngörülemeyen ve önemli bilgileri bulma üzerine odaklanan popüler bir araştırma alanıdır. Sosyal medya verileri, sosyal ağ siteleri, mikrobloglar, fotoğraf veya video paylaşım sitelerinden toplanan en popüler ve büyük heterojen verilerden biridir. Sosyal medya, varlıkları ve onların ilişkilerini temsil eder. Veri madenciliği alanındaki büyük heterojen verileri temsil etmek için kullanılan popüler veri yapılarından biri graftır. Bir grafın düğümleri varlıkları, kenarları ise varlıklar arasındaki ilişkileri temsil eder. Dolayısıyla, graf madenciliği, veri madenciliğinin en popüler alt bölümlerinden biridir. Bir sık örüntü, bir veri kümesinde kullanıcı tanımlı eşiğe göre daha sık rastlanan örüntü olarak adlandırılır. Veri kümesindeki sık örüntüler veri kümesi hakkında önemli bilgiler verebilir. Bu bilgiyi kullanarak, veriler sınıflandırılabilir veya kümelenebilir. Sık örüntüler sosyoloji, tüketici davranışı, pazarlama, topluluklar açısından sosyal medya verilerine farklı bir bakış açısı sağlayabilir.

Bu tez kapsamında popüler sık örüntü madenciliği algoritmaları incelenmiştir ve çoğu algoritmanın büyük veri setleri için uygun olmadığı gözlenmiştir. Günümüz dünyasındaki veriler, özellikle sosyal ağlar çok büyük verilere sahip olduğundan, var olan sık örüntü madenciliği algoritmaları bu veri setleri için uygun değildir. Bu tezin amacı, mevcut bir sık örüntü madenciliği algoritmasını paralel bir şekilde uygulamak ve bir sosyal medya verisinde sık örüntüleri bulmaktır.

TABLE OF CONTENTS

LIST OF FIGURES viii

LIST OF TABLES ix

CHAPTER 1. INTRODUCTION 1
1.1. Thesis' Aim and Objectives 4
1.2. Organization of Thesis 4

CHAPTER 2. BACKGROUND 5
2.1. Basic Graph Terminology 5
2.1.1. Graph 5
2.1.2. Graph Embedding, Edge-disjoint Embedding, Overlap Graph . 6
2.1.3. Subgraph 8
2.1.4. Induced Subgraph 9
2.1.5. Subgraph Isomorphism 9
2.1.6. Frequent Subgraph 10
2.2. Apache Spark Framework 10
2.3. Fork/Join Framework in Java 11
2.4. Social Media Data 12

CHAPTER 3. RELATED WORK 14
3.1. Frequent Subgraph Mining (FSM) Process 14
3.1.1. Candidate Generation 14
3.1.2. Frequency Evolution 16
3.2. Categorization of FSM Algorithms 18
3.2.1. Algorithmic Approach 18
3.2.2. Algorithmic Design 19
3.2.3. Graph Representation 21
3.2.4. Input Type 22
3.2.5. Graph Type 23
3.2.6. Nature of Graph 23

3.2.7. Result Type 24
3.3. Popular FSM Algorithms 24
3.3.1. AGM Algorithm 24
3.3.2. FSG Algorithm 25
3.3.3. FFSM Algorithm 27
3.3.4. MOFA Algorithm 27
3.3.5. gSpan Algorithm 28
3.3.6. CloseGraph Algorithm 29
3.3.7. SIGRAM Algorithm 29
3.3.8. GERM Algorithm 31
3.3.9. Stream FSM Algorithm 32
3.3.10. Mining Interesting Patterns and Rules in a Time Evolving Graph 32
3.3.11. SUBDUE Algorithm 33
3.3.12. SEuS Algorithm 34
3.3.13. FSM-H Algorithm 34
3.3.14. gSpan-H Algorithm 35
3.3.15. p-MOFA and p-gSpan Algorithm 35

CHAPTER 4. PARALLEL FSM ALGORITHM IMPLEMENTATIONS 37
4.1. Spark based Parallel gSpan algorithm Implementation 38
4.2. Multi-thread based Parallel gSpan Algorithm Implementation 41

CHAPTER 5. EXPERIMENTAL RESULTS 44
5.1. The Results for Chemical Compound Dataset 45
5.2. The Results for Friendster Dataset 47
5.3. Discussion on Results 51

CHAPTER 6. CONCLUSION 53

REFERENCES 55

LIST OF FIGURES

Figure **Page**

Figure 2.1. A graph (G) example. 5
Figure 2.2. An example of graph (G) embedding (G_i) example. 6
Figure 2.3. A directed graph (H) example. 7
Figure 2.4. A weighted graph (G_x) example. 8
Figure 2.5. A simple graph (G_s) example. 8
Figure 2.6. Non-induced (G_m) and induced graph (G_n) examples. 9
Figure 2.7. Isomorphic graphs example. 10
Figure 2.8. Social media data example. 13
Figure 3.1. Apriori based growth. 19
Figure 3.2. Pattern-growth based growth. 20
Figure 5.1. The number of frequent subgraphs for chemical dataset. 45
Figure 5.2. Run times of multi-threaded implementation for chemical compound dataset. 46
Figure 5.3. Run times of Spark-based implementation on different environment for chemical dataset. 46
Figure 5.4. The number of frequent subgraphs for friendster dataset. 48
Figure 5.5. Run times of multi threaded implementation for friendster dataset. 48
Figure 5.6. Run times of Spark-based implementation for friendster dataset. 49
Figure 5.7. The number of frequent subgraphs for different dataset sizes. 50
Figure 5.8. Run times of multi thread implementation for different dataset sizes. ... 50
Figure 5.9. Run times of multi thread implementation for diferent thread counts. ... 51
Figure 5.10. The number of frequent subgraphs for different supports and dataset sizes. 51

LIST OF TABLES

Table		Page
Table 3.1.	Candidate generation approaches of FSM algorithms.	15
Table 3.2.	Frequency calculation approaches of FSM algorithms.	17
Table 3.3.	Categorization of FSM algorithms.	26
Table 5.1.	Result set for chemical compound dataset.	47
Table 5.2.	Result set for friendster dataset.	49

CHAPTER 1

INTRODUCTION

Data mining is the process of automatically extracting previously unknown, meaningful and useful knowledge from large datasets (Han et al., 2011). In today's world, the data grows day by day and it is necessary to find accurate and interesting information from this large volume of data. For this reason, data mining has become an important area for researchers.

Social media provides a rich source of information about people and their opinion, feelings, human relationships, any kind of information and news. Using social networking sites like Facebook, people can share personal or public information with relatives, family members, colleagues and many other friends (Barbier and Liu, 2011). This shared information constitutes social media data. Social media data is huge, for example, Facebook has about 2 billion users, and this data constantly changing. It is quite noisy because it contains lots of trivial data.

The data processed in the data mining can be obtained from many sources where different data types can be used. Examples of these different data types are text data, sound data, image data, graph data, etc. Since graphs better represent the complex and arbitrary relations among data attributes, they are used to represent data in many application areas, such as users (nodes) and their relationship (edges) in social networks, atoms (nodes) and bonds (edges) between them in chemical structures, proteins (nodes) and protein interactions (edges) in biological network, computer (nodes) and links between them in computer networks (Chakrabarti and Faloutsos, 2006). Graph mining is a data mining subdivision where the data is represented as graph (Rehman et al., 2012). Since the graphs are used to represent the social media data, within the scope of this thesis, graphs are used as patterns.

One of the important data mining tasks is the problem of finding frequent subgraphs in a graph. The aim of frequent subgraph mining (FSM) is to find all subgraphs whose occurrences are at least equal to the number of user-defined threshold (Aggarwal and Wang, 2010). In many domains it is necessary to find these common structures, because these repetitive structures can provide a better understanding of the data or give a

different perspective about data. These frequent subgraphs are used in determining the similarities between the graphs, clustering (Zimek et al., 2014), graph indexing Yan et al. (2004) and classification (Acosta-Mendoza et al., 2012).

The FSM algorithm consists of two important phases: candidate generation and frequency calculation. Candidates are generated using breath first strategy or depth first strategy. One of the most important factors affecting the performance of the algorithm when generating candidates is the generations of the same candidates more than once. Since the dataset grows, number of candidates generated grow. Duplicated and redundant candidates should be avoided during candidate generation for an efficient algorithm. The second phase in the frequent mining algorithm is to calculate the frequency of the generated candidates and to determine which are frequent among them. To calculate the frequency of a subgraph, it is necessary to find the number of graphs that are isomorphic to this subgraph in a dataset. The subgraph isomorphism testing is fundamental problem of these algorithms since this problem is NP-complete (Garey and Johnson, 2002). The cost of finding isomorphic graphs increases exponentially as the size of the graph dataset increases.

The solutions proposed by different FSM algorithms can be divided into different categories, depending on the algorithmic approach, algorithmic design, graph representation, input type, graph property, nature of graph and result type.

The algorithms examined in this area generally use two different algorithmic approaches. These approaches are apriori-based and pattern-growth approaches. Apriori-based algorithms (Inokuchi et al. (2000); Kuramochi and Karypis (2004); Huan et al. (2003)) generate candidates based on breadth first strategy and apply subgraph isomorphism testing to calculate frequencies of candidates. Especially when the dataset is large, these algorithms suffer from generating too many candidates. Pattern-growth based algorithms (Borgelt and Berthold (2002); Huan et al. (2003); Yan and Han (2002); Yan and Han (2003)) generate candidates based on depth first strategy. The pattern growth approach avoids the cost of generating and testing many candidate subgraphs. These candidates are generated by extending frequent subgraphs starting from minimal frequent subgraphs by adding one edge at every step if they are still frequent. However, these algorithms might suffer from the generation of the same candidates.

Many algorithms that solve the frequent subgraph mining problem give good results on small datasets but not suitable for working on large datasets. However, the real datasets contain very large data and the algorithms should be able to work effectively on

large datasets. As the data size grows, the data cannot fit in the memory on a single machine, or it may not be an efficient and practical method to work on a single machine with this large amount of data. For this reason, parallel algorithms have been developed to find frequent subgraphs in large datasets (Meinl et al. (2006); Bhuiyan and Al Hasan (2015)).

The most popular methods used to represent graphs are the adjacency matrix and adjacency list. Graphs should be represented uniquely to facilitate subgraph isomorphism testing. Since there may be more than one adjacency matrix representing the same graph, new methods are proposed such as canonical adjacency matrix (CAM) (Huan et al., 2003) and mininimum DFS code (Yan and Han, 2002).

There are also two different problem statements according to the input type: The dataset used in the algorithm may also be a single large graph or small graphs (set of graphs) these are called transactions. In this case, the frequency calculation of a subgraph in the single large graph dataset is different from the transactional dataset. While calculating the frequencies of candidates in a transactional dataset, the number of transactions that contain this subgraph is calculated. However, since there is no transaction in a single large graph dataset, different methods was proposed (Holder et al. (1994); Kuramochi and Karypis (2005); Ghazizadeh and Chawathe (2002)).

Graphs used in FSM algorithms may have different properties from each other, for example, graphs may be directed or undirected, multiple edges between nodes may or may not be allowed. Most of the algorithms work with connected graphs. According to properties of graphs, the solutions suggested for FSM are adapted.

Apart from these, graphs can be static (time invariant) or dynamic (time varying). Static graphs do not change over time and can be stored in a database. For this reason, the existing FSM algorithms can be applied to these graphs. Dynamic graphs are constantly changing graphs and cannot be stored in a database. For example, streaming data comes to a network only once and it is not possible to keep this data in the database. For this reason, existing FSM algorithms are not suitable for these data types.

While most FSM algorithms find all subgraphs in a dataset, some algorithms find a more meaningful superset (such as closed frequent subgraphs (Yan and Han, 2003)) of these frequent subgraphs to reduce search space and facilitate working on large datasets.

Within the scope of this thesis, first the basic two steps of the FSM process is analyzed. The methods used in these two steps and the algorithms using these methods are examined. Then a categorization of the FSM algorithms according to the above-mentioned properties is presented. There are detailed surveys that compare FSM algorithms ac-

cording to their different attributes (Lakshmi and Meyyappan (2012); Jiang et al. (2013); Muttipati and Padmaja (2015)). However, they do not capture recent algorithms focusing on new requirements as dynamicity or volume of data. In real systems, data can change continuously over time. For example, in a social or telecommunication network, data only goes through once and it is very difficult to keep such data in the database. Such data is called stream data and existing FSM algorithms are not suitable for working on stream data. Since most FSM algorithms are also not suitable for large datasets, the algorithms proposed in this area recently are parallel algorithms. Generally, the existing frequent subgraph algorithms have been modified to work in parallel. Within the scope of this thesis, dynamic and parallel algorithms are also examined.

1.1. Thesis' Aim and Objectives

The aim of this thesis is to mine a snapshot of social media data at a time with a parallel FSM algorithm. In this thesis, two different parallel implementations of an existing FSM algorithm have been performed. First algorithm is a multi-threaded Fork/Join Framework based parallel algorithm. The second algorithm is a parallel algorithm based on the Apache Spark Framework that can run both locally and on a cluster. These two algorithms are tested on a social media data and the results are compared according to different parameters.

1.2. Organization of Thesis

Chapter 1 gives an introduction for thesis topic. In the chapter 2 basic concepts that should be known about the problem of frequent subgraph mining, Apache Spark Framework and social media data are introduced. Chapter 3 gives related work of FSM. Section 3.1 presents the process of FSM and section 3.2 presents the categorization of FSM algorithms according to the different perspective and section 3.3 presents the popular FSM algorithms. In the chapter 4, two parallel implementation of FSM algorithms are addressed. In the chapter 5, the results of two parallel implementation according to the different parameters are presented. Chapter 6 gives the conclusion of this thesis.

CHAPTER 2

BACKGROUND

In this section, the basic concepts and terminology related to graph, Apache Spark Framework, Fork/Join Framework and social media data are introduced.

2.1. Basic Graph Terminology

This section introduces key terms and their definitions that should be known about graph theory within the scope of this thesis.

2.1.1. Graph

A graph $G = (V, E)$ consists a finite set of nodes (or vertices) and set of edges that connects these nodes each other. Figure 2.1 is a graph example. V represents a set of nodes of the graph G and is usually expressed as $V(G)$ or V. E represents a set of edges of the graph G and is usually expressed as $E(G)$ or E.

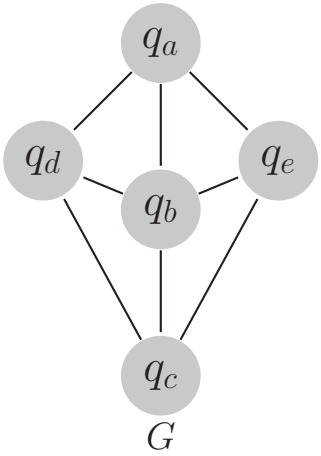

Figure 2.1. A graph (G) example.

Edges and vertices lists that contain all edges and nodes of graph can be represented as follows:

$$G = (V, E)$$
$$V = \{\ q_a, q_b, q_c, q_d, q_e\ \}$$
$$E = \{\ \{\ q_a, q_b\ \},\{\ q_a, q_d\ \},\{\ q_a, q_e\ \},$$
$$\{\ q_b, q_c\ \},\{\ q_b, q_d\ \},\{\ q_b, q_e\ \},\{\ q_c, q_d\ \}\ \},\{\ q_c, q_e\ \}\ \}$$

2.1.2. Graph Embedding, Edge-disjoint Embedding, Overlap Graph

Let $G = (V, E)$ be a graph. A graph dataset may contain this graph or its particular drawing. They are called *embedding* of graph $G = (V, E)$. If two embeddings of any graph have no common edge, these embeddings are called *edge-disjoint embeddings*. In order to create *overlap graph* of these embeddings, a vertex is created for each non-identical embeddings of a subgraph and edges are also created between vertices that represent edge-disjoint embeddings. Figure 2.2 shows an example of graph embedding.

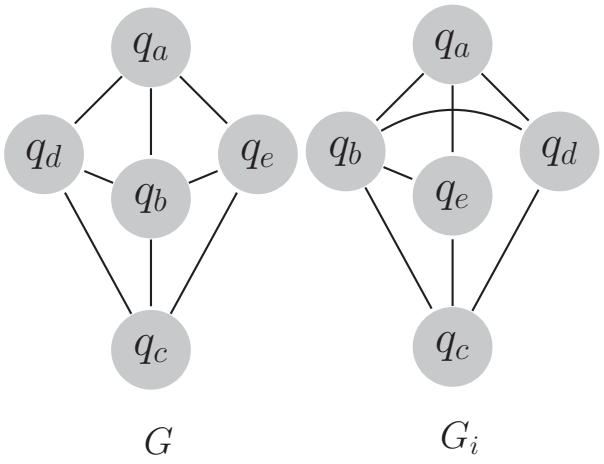

Figure 2.2. An example of graph (G) embedding (G_i) example.

In a graph, the edges can be traversed both direction between the nodes. This graph is called *undirected graph*. In an undirected graph, the edge (x, y) is identical to edge (y, x). Figure 2.1 is also an example of undirected and labelled graph.

If edges of a graph have direction, that is, this graph can be traversed one direction between nodes and this graph is called *directed graph*. In a directed, graph the edge (x, y) is not identical to edge (y, x). Figure 2.3 is an example of directed and labelled graph.

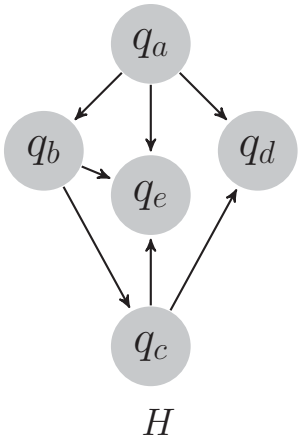

Figure 2.3. A directed graph (H) example.

Edges and vertices lists that contain all edges and nodes of directed graph can be represented as follows:

$$G = (V, E)$$
$$V = \{\ q_a, q_b, q_c, q_d, q_e\ \}$$
$$E = \{\ \{\ q_a, q_b\ \},\{\ q_a, q_d\ \},\{\ q_a, q_e\ \},$$
$$\{\ q_b, q_c\ \},\{\ q_b, q_e\ \},\{\ q_c, q_d\ \},\{\ q_c, q_e\ \}\ \}$$

In a graph, edges have weight that represents cost of traversing, such graph is called *weighted graph*. These weights are represented as numerical labels of the edges. Figure 2.4 is an example of weighted graph.

If there is a path between every pair of vertices of a graph, this is called *connected graph*, otherwise *unconnected graph*. Figure 2.1 is also connected graph examples. Figure 2.3 is also example of unconnected graph.

If a vertex in a graph is connected to itself with an edge, this is called a *loop*. In a *multi-edge graph*, there are more than one edge between any two vertices. If a graph is undirected and unweighted and there is no multi-edge and loop, this graph is called

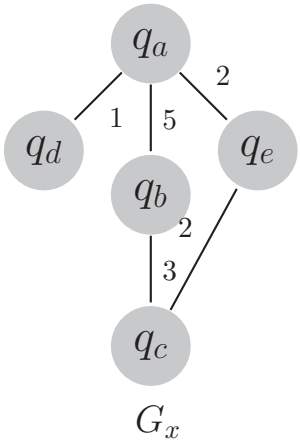

Figure 2.4. A weighted graph (G_x) example.

simple graph. Figure 2.5 is an example of simple graph. In the scope of this thesis, simple, connected, labelled graphs are used.

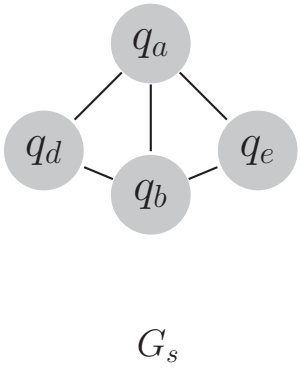

Figure 2.5. A simple graph (G_s) example.

2.1.3. Subgraph

A graph G_s is the subgraph of graph G, if only if nodes and edges of a graph G_s (V_s, E_s) are subset of nodes and edges ($V_s \subseteq V$ and $E_s \subseteq E$) of graph G (V, E). The graph G_s in Figure 2.5 is also a subgraph of graph G in Figure 2.1. Edges and vertices

lists of subgraph is represented as follows:

$$G_s = (V, E)$$
$$V_s = \{\ q_a, q_b, q_d, q_e\ \}$$
$$E_s = \{\ \{\ q_a, q_b\ \},\{\ q_a, q_d\ \},\ \{\ q_a, q_e\ \},\{\ q_b, q_d\ \},\{\ q_b, q_e\ \}\ \}$$

2.1.4. Induced Subgraph

Let, graph G_n is subgraph of G. The vertices of subgraph G_n are a subset of the vertices of graph G. If all edges of these vertices in the graph G are also exist in the subgraph G_n this subgraph G_n is called *induced subgraph* of graph G. The Fig. 2.6 gives both induced and non-induced graph example.

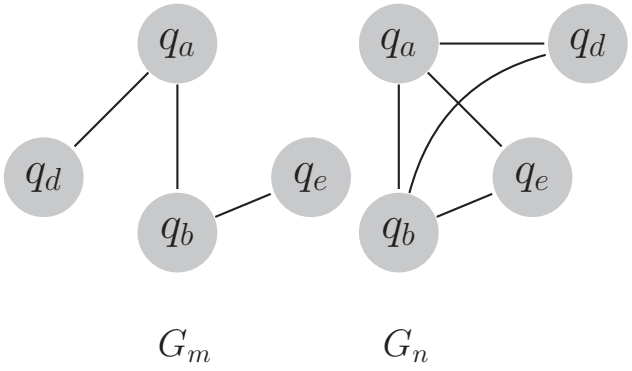

Figure 2.6. Non-induced (G_m) and induced graph (G_n) examples.

2.1.5. Subgraph Isomorphism

Let G = (V,E) and $G^{'}$ = ($V^{'}$, $E^{'}$) be graphs. Graph G and graph $G^{'}$ are topologically identical to each other, that is, if there is a mapping between the vertices and edges of two graphs. Thus, these two graphs G and $G^{'}$ are called isomorphic graphs. The subgraph isomorphism problem tries to find out whether a subgraph is included by another graph in a graph dataset. The detection of isomorphic subgraphs is a NP-complete

problem (Garey and Johnson, 2002). The cost of finding isomorphic graphics grows exponentially as the size of the dataset increases. There are many subgraph isomorphism detection techniques (Schmidt and Druffel (1976); Ullmann (1976); McKay et al. (1981); Cordella et al. (1998)). The figure 2.7 is an example of isomorphic graphs.

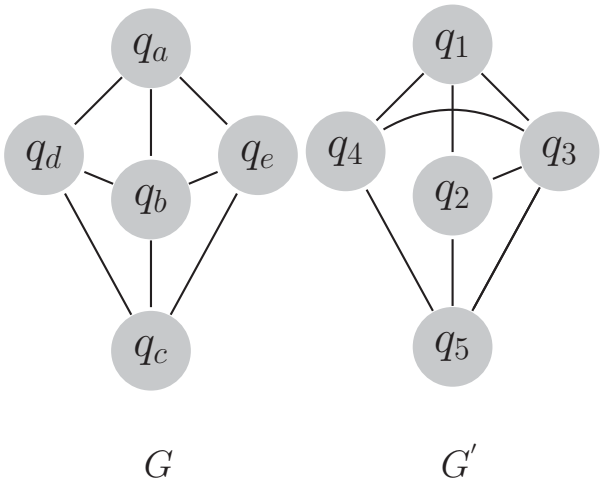

Figure 2.7. Isomorphic graphs example.

2.1.6. Frequent Subgraph

The *support* σ ($0 < \sigma < 1$) of a subgraph is the ratio of the number of graphs that are isomorphic to this subgraph to total number of graphs. If the support of a subgraph satisfies the user-defined minimum support threshold, this subgraph is called *frequent subgraph*. If a graph is frequent, all its subsets must be frequent. This is called *downward closure property*.

2.2. Apache Spark Framework

Apache Spark Framework (Spark, 2016) is a data processing framework that work on large amount of distributed data. Apache Spark is based on Hadoop MapReduce (Hadoop (2009); Dean and Ghemawat (2008)). MapReduce is a programming model that

allows a large amount of data to be processed in parallel. Hadoop is a framework based on MapReduce programming model. It processes large amount of data that distributes across clusters of computers. A MapReduce job distributes the input data into different chunks. Map-Reduce consists of two steps: Map and Reduce. During the map phase, the data in the chunks is transformed into a set of tuples (key/value pairs). The result of map phase is written to Hadoop Distributed File System (HDFS) (Shvachko et al., 2010). During the reduce phase, the result of map phase is read from HDFS and all intermediate values associated with the same key are combined. The result of reduce phase is also written the HDFS.

Apache Spark extends the MapReduce programming model and uses the HDFS for storage. Map Reduce is not suitable for algorithms and calculations that require multiple passes. At each step there are a map and a reduce phase and the results of the map and reduce phases for each step are written in HDFS. The result of map phase is the input of reduce phase, the result of reduce phase is the input of next step. Each output is written the HDFS and each input is read from HDFS. Disk access is very costly and greatly slows down the calculation. This is a disadvantage of Hadoop MapReduce. One of the most important features of Spark and difference from Hadoop MapReduce is that it calculates and stores data in memory. If the data cannot fit in-memory, then it stores the data in the disk. Spark is faster than Hadoop both in memory and disk, because it reduces the number of reading from disk and writing to disk by storing the intermediate data in-memory.

Spark provides four libraries: Spark Streaming, Spark SQL, Spark Mlib, Spark GraphX. Spark Streaming library provides processing of real-time streaming data. Spark SQL provides structured data processing and SQL-like queries execution on Spark data. Spark Mlib provides scalable and distributed machine learning library that consist of some learning algorithms such as classification, regression, dimensionality reduction. Spark GraphX is framework that allows parallel graph computation.

Spark provides different data types: *RDD*, *DataFrame* and *GraphFrame*. *RDD* is a data structure of Spark which can store any data type. *RDDs* can be distributed over the different partitions. A *DataFrame* stores the distributed data collection in named columns as in a database table in the relational database. *GraphFrame* is a data structure that using to represent the graph and the its vertices and edges represented by using *DataFrame*.

There are different Spark deployment ways: A Spark application can be deployed on distributed cluster managers like YARN and Mesos or on a simple standalone server.

2.3. Fork/Join Framework in Java

This framework (Lea, 2000) relies on the divide and conquer approach to speed up parallel processing. It separates the job to be done into smaller tasks recursively and tries to complete these tasks using all available processor cores. This framework has a thread pool that contain all alive threads. Since thread creation is very costly, available threads are used in this pool instead of creating a new thread for each sub task. This framework is also responsible for managing the threads in the thread pool and assigning jobs to be done to threads in the pool. It uses the work stealing algorithm to provide load balancing. Each thread has its own deque, which is a list variable that can be accessed from both the head and the tail. According to this algorithm, each thread takes its task from the head of its deque. If its deque is empty, it takes task from the tail of the deque of another thread.

2.4. Social Media Data

A social media data can be represented as graph. Figure 2.8 is an example of social media data. The nodes represent the entities and the edges represents the relationship between them. For example, for a Facebook data, there may be a *friends* relation between two users. These users are represented by nodes and *friends* relation is represented by an edge. Social media data has three important characteristics.

- Social media data contains a large amount of data.
- This data is constantly changing and it is very difficult to track and keep these changes.
- It is inevitable that such a huge dataset contains lots of unnecessary or trivial data.

The analysis of social media data can provide different perspectives for many areas. By analyzing the social media data of a community, significant conclusions for sociology can be discovered about that community's characteristics. In the field of marketing, a recommendation system can be constructed by analyzing people's interests and behavior in social media. Or analysis of the feelings, opinion and behaviors of people in

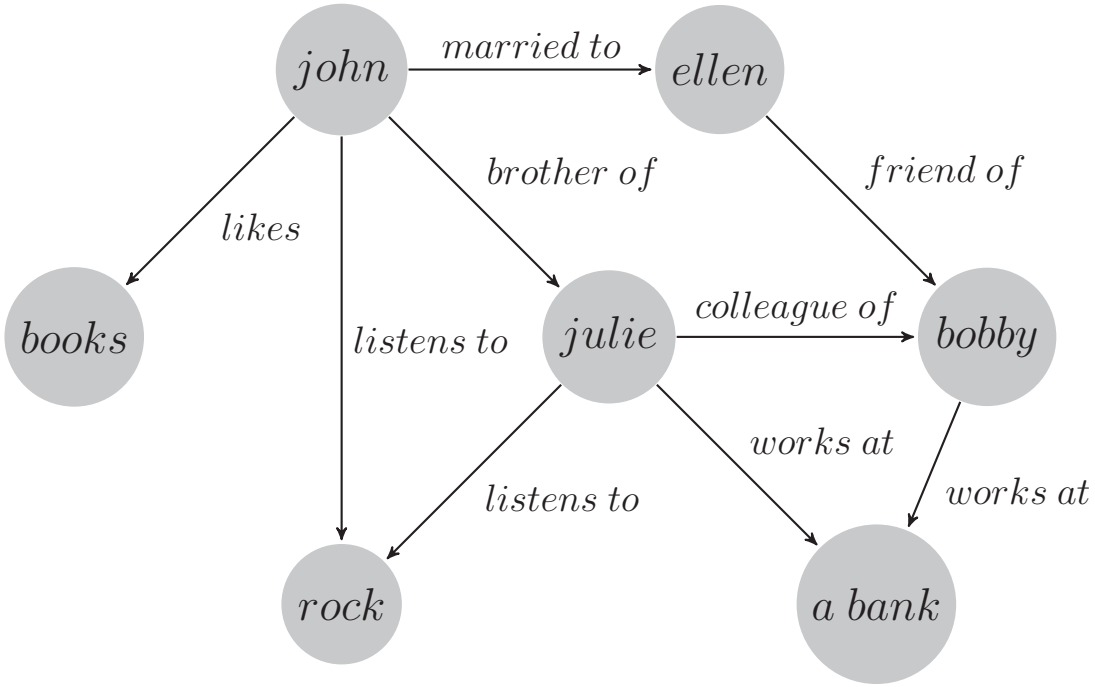

Figure 2.8. Social media data example.

the social media can provide important results for the psychology field. For this reason, social media analysis is an important field of study.

CHAPTER 3

RELATED WORK

This chapter first describes the FSM processes and the methods used in these processes and provides a categorization FSM algorithm according to algorithmic approach they use, algorithmic design, graph representation, input type, graph type, nature of graph and result type. Finally, the popular FSM algorithms are analyzed according to both the FSM process and their outstanding characteristics are discussed in this chapter.

3.1. Frequent Subgraph Mining (FSM) Process

The aim of FSM is to find all frequent subgraphs in a graph dataset. Generally, FSM algorithms consists of two phases: Generation of candidate subgraphs and computing the support of this candidates to determine whether they are frequent or not.

3.1.1. Candidate Generation

FSM algorithms generate a (n+1)-edge candidate subgraph by adding a new edge to a n-edge frequent subgraph. This process starts with 1-edge graphs and continues until all the nodes and edges of a graph are included. The point to be noted during candidate generation is that each candidate should be generated only once. Since the cost of the finding isomorphic graphs grows exponentially with the number of candidates, unnecessary candidate generations should be avoided to narrow the search space. There are four popular candidate generation strategies: level-wise join, extension, join and extension and rightmost extension strategy. Table 3.1 shows the popular FSM algorithms candidate generation approach they used.

Level-wise join strategy: In level-wise join, two k-size subgraphs are combined to generate a new (k+1)-size subgraph. These two k-size subgraphs can be joined if they have a common (k-1)-size subgraph. This method has the following disadvantages:

- A single join operation can generate more than one candidates.
- The different join operations can generate same candidates.
- Generated candidates may not satisfy the downward closure property.

Table 3.1. Candidate generation approaches of FSM algorithms.

Algorithms	**Level-wise join**	**Extension**	**Join & extension**	**Rightmost extension**
AGM	√			
FSG	√			
FFSM			√	
MOFA		√		
gSpan				√
CloseGraph				√
GERM				√
Stream FSM				
Time Evolving Graph				√
Subdue	√			
SEuS	√			
HSIGRAM	√			
VSIGRAM		√		
FSM-H	√			
gSpan-H	√			
p-MOFA		√		
p-gSpan				√

Extension strategy: In extension strategy, new connected (k+1)-size subgraph is obtained by adding an edge to all possible k-size embeddings.

Join and extension strategy: The FFSM (Huan et al., 2003) algorithm presents two new methods on the deficiencies of level-wise and extension strategies: FFSM-join and FFSM extension. In the FSM-join method, up to two candidates are generated instead of too many candidates. However, this method may not always enumerate all the subgraphs. In the FFSM-extension method, a single fixed node is specified instead of candidate nodes. The new edge is added between an additional node and this fixed node.

Rightmost path extension strategy: In the rightmost path extension strategy, a new (k+1)-size sub-tree is generated by adding an edge to the rightmost path of a k-size sub-tree. This edge can be added between two existing rightmost nodes or between the existing rightmost node and a new introduced node. This method has the following disadvantages:

- There may be too many nodes to which the new edge can be added.
- This method greatly increases the complexity of this algorithm.

3.1.2. Frequency Evolution

The frequency of a subgraph is the number of isomorphic graphs of this subgraph in the graph dataset. The subgraph isomorphism problem is NP complete and the computational cost increases exponentially as the problem size grows. For this reason, the subgraph isomorphism test can be applied on small datasets. To calculate the frequency of a graph, many frequent subgraphs algorithms perform subgraph isomorphism testing, but some have suggested different methods to avoid this test, or some intuition to speed up this test. The method used by the algorithms examined in this study to calculate the frequency is given in Table 3.2.

Database scan: To calculate the frequency of candidate subgraphs, database is scanned from begin to end to determine how many different transactions include this candidate. This procedure is repeated for each candidate and re-scanning the entire database is not a very efficient method. Especially scanning the large databases significantly affects the runtime of the algorithm.

Transaction list: There is a transaction identifier list for each frequent subgraph. To calculate the frequency of a k-size graph, the intersection of the TID (transaction identifier list) lists of all its (k-1)-size subgraphs are checked. If the intersection size satisfies the user-defined support value, the frequency is calculated by performing a subgraph isomorphism test on the transactions at this intersection. However, this method has a disadvantage. These TID lists require a lot of memory to keep them in memory. Also, these lists may not fit in memory for large datasets.

Embedding list: While calculating the frequency of candidate subgraphs, embedding lists of discovered subgraphs are stored to avoid subgraph isomorphism testing. The frequency of a candidate subgraph is determined from the number of different graphs in its embedding list. Since this method requires a lot of memory to keep embedding lists in memory, this method also is not suitable for large datasets.

Table 3.2. Frequency calculation approaches of FSM algorithms.

Algorithms	Database scan	Transaction list	Embedding list	MIS	MNI	MDLP	Occurrence list
AGM	√						
FSG		√					
FFSM			√				
MOFA			√				
gSpan		√					
CloseGraph		√					
GERM					√		
Stream FSM							
Time Evolving Graph					√		
Subdue						√	
SEuS	√						
HSIGRAM				√			
VSIGRAM				√			
FSM-H							√
gSpan-H							√
p-MOFA			√				
p-gSpan			√				

Maximum independent set (MIS): Since there are no transactions in single large graph that can be scanned to find the frequency of a subgraph. Firstly all the embeddings of this subgraph in the graph are found and their overlap graph is constructed. Then, on this overlap graph, an exact or approximate maximum independent set is found.

Minimum image based support (MNI): One of the methods used to calculate the frequency of a candidates in a single large graph is the minimum image based measure (Bringmann and Nijssen, 2008). In this method, the number of unique nodes in the graph dataset that can be mapped to the for each node of candidate subgraph are found and the minimum one is considered as the frequency of this candidate subgraph.

Minimum description length principle (MDLP): The purpose of the minimum description length principle is to reduce the description length of the entire dataset (Holder

et al., 1994). The entire dataset is compressed by representing the graphs with bits. The minimum description length of a graph is the number of bits used to represent this graph.

Occurence list: An occurrence list contains all the embeddings of a subgraph and information about the graphs that correspond to these embeddings in the transactional graph database (Bhuiyan and Al Hasan, 2015). While calculating the frequency of a k-size graph, instead of solving the subgraph isomorphism problem, the intersection of occurrence lists of its (k-1)-size subgraphs is checked to avoid subgraph isomorphism checking.

3.2. Categorization of FSM Algorithms

FSM algorithms can be categorized according to the some properties: algorithmic approach, algorithmic design, graph representation, input type, graph type, nature of graph and result type as shown in Table 3.3.

3.2.1. Algorithmic Approach

FSM algorithms can be divided into two different categories according to their algorithmic approach: Apriori based approach and pattern-growth based approach.

Apriori based approach: Apriori based algorithms generally find all the connected frequent subgraphs and consist of two steps: candidate generation and subgraph isomorphism test to calculate the frequencies of graphs. In the first step, these algorithms use the level-wise strategy for candidate generation. Apriori based algorithms suffer from too many candidate generation for large datasets. For this reason, these algorithms narrow the search space using the downward closure property. According to this property, if a subgraph is not frequent, an upper set containing it is not frequent. In the next step, it is no longer necessary to check whether any candidate graph containing this subgraph is frequent or not. Apriori based algorithms reduce the number of candidates significantly but, these algorithms do not work efficiently for large datasets especially when the minimum support threshold is small. Because too many candidates are generated, and this process requires a lot of database scanning. Apriori based algorithms also suffer from subgraph isomorphism testing. The figure 3.1 is an example of Apriori based growth.

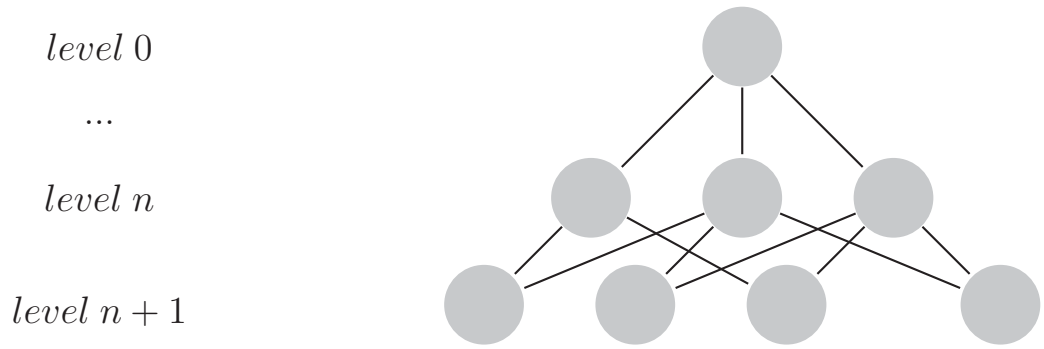

Figure 3.1. Apriori based growth.

Pattern-growth based approach: The aim of pattern-growth based algorithms is to find all the frequent subgraphs without the candidate generation and subgraph isomorphism test. This approach is based on the divide and conquer method. Instead of generating all the candidates, a new edge is added to every possible position of the existing frequent subgraph. This process is continued until there is no more frequent subgraphs. One of the most important issues that apriori based algorithms suffer from is that they make too many database scanning. For this reason, pattern-growth based algorithms use a more compact and smaller data structure instead of processing in the database. The number of candidates generated in this approach are reduced considerably and the subgraph isomorphism test is better than the apriori based algorithms. However, this approach has a disadvantage. The same subgraph can be produced many times while adding a new edge to every possible position in the current frequent subgraph. This problem has been tried to be avoided by using the rightmost path extension strategy. Pattern-growth based FSM algorithms generally use rightmost extension strategy while generating candidates and to avoid subgraph isomorphism testing using minimum DFS code while calculating the frequencies of subgraphs. The figure 3.2 represents the pattern based growth.

3.2.2. Algorithmic Design

Most of the algorithms that propose solutions to the frequent subgraph mining problem are not suitable for working on large datasets. The size of the input data may not be suitable for mining on a single machine, or generated candidates may not fit into

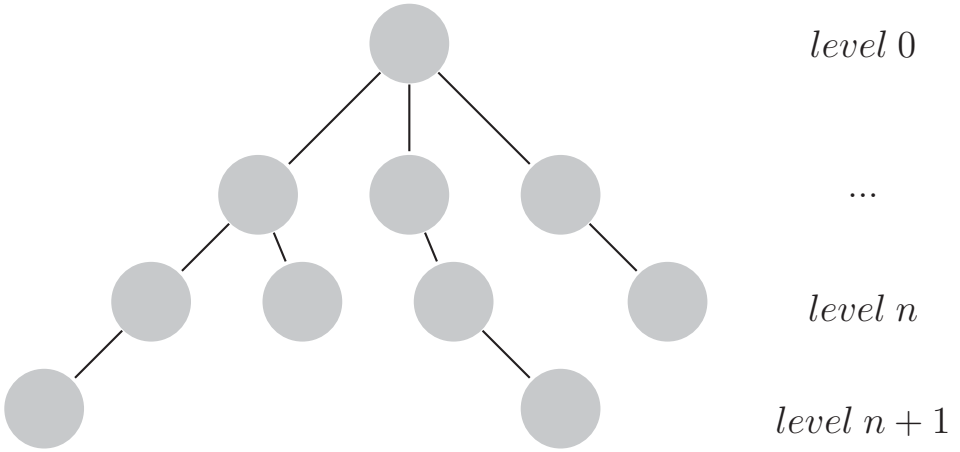

Figure 3.2. Pattern-growth based growth.

the memory of single machine or it may not be an efficient method. For this reason, algorithms that work parallel on more than one machine have been proposed. In the parallel algorithm, the work to be done is assigned to the processes that will run parallel to each other. In addition to FSM, there are three important issues to consider when developing a parallel algorithm: First, a parallel algorithm should be memory scalable. That is, process should have large enough data to fit in memory and as the number of processes to run in parallel increases, the memory required to operate these processes should also be enough. The second issue to consider is that the tasks should be distributed equally in each process. A working time of a process may not always be predictable, so a task on a process may end up before others. The last issue that needs to be taken into consideration is the remaining task in the other processes should be dynamically distributed to the idle processes. In this way both the idle time of the processes is reduced, and the work is completed in a shorter time.

There are two different memory systems used when developing a parallel algorithm: Shared memory systems and distributed memory systems.

Shared memory systems: Processes running on different machines share a common memory address space in shared memory systems. The most important advantage of these systems is that processes can communicate with each other through this memory address space. One of the most important problems of these systems is the race conditions. The two processes may want to access an address in the common address space and change the

value of an item in this address at the same time. There may be delays when the common address is on different machines with the running process.

Distributed memory systems: In distributed memory systems, processes communicate with each other by network transmission or writing to a file or reading a file instead of sharing a memory space to communicate with each other. There are two programming models used in distributed memory systems: The first is the message passing. With this method, the processes communicate with each other by sending messages over the network. For this reason, network bandwidth is one of the important factors affecting the system and network traffic should be reduced as much as possible. The other method is Map Reduce. The map reduce model consists of two functions: Map and Reduce. The input of algorithm is assumed a (key, value) pair sets. The Map function implements the map function to each (key, value) pair and emits the other (key, value) pairs. During the reduce phase the pairs that have the same key value are aggregated after the map function. Reduce function keeps these values in a sorted list and implements the reduce function to this list. In the message transfer model, the communication of process in the system is transparent to the user, but in the Map Reduce method the user does not need to have detailed knowledge about the communication in an address field.

3.2.3. Graph Representation

There are 3 different ways to represent graphs: Adjacency matrix, adjacency list and canonical labelling.

Adjacency matrix: In an adjacency matrix, rows and columns represent the graph nodes. Let v_a and v_b two nodes of a graph. If there is an edge between these two nodes, they are called adjacent to each other and (a,b). position of an adjacency matrix is represented by 1 or edge label, otherwise this position is represented by 0. Same graph can be represented by more than one adjacency matrix depending on the position of the vertices in the row and column.

Adjacency list: All nodes of a graph are stored in an array. Each element (node) of this array points a node list of all adjacent nodes to this node.

Canonical Labelling: The purpose of canonical labelling is to uniquely identifying a graph. A canonical label is obtained from adjacency matrix of a graph by concatenating its rows and columns. Since there are multiple adjacency matrices representing a graph, there can be more than one canonical label representing this graph. Since the purpose of the canonical labelling is to uniquely identify a graph, the graphs are represented by minimum or maximum canonical labels according to the lexicographic order. To solve the problem of subgraph isomorphism, graphs must be represented uniquely. The canonical labels of isomorphic graphics are the identical. In FSM algorithms, graphs are usually represented by minimum canonical labels, and labels of two graphs are used to determine whether two graphs are isomorphic. Several different canonical labelling methods have been proposed:

- **Canonical adjacency matrix:** In a adjacency matrix, if there are labels of nodes on diagonal entries and label of edges on off-diagonal entries, this matrix is called canonical adjacency matrix (Huan et al., 2003). The canonical code of a subgraph is obtained by concatenating the upper or lower triangular entries of the adjacency matrix. Since there can be multiple adjacency matrices of this subgraph, the matrix which has the minimum or maximum canonical code is the canonical adjacency matrix.

- **Minimum DFS code:** The minimum DFS code method has been proposed to represent the graphs uniquely in the gSpan algorithm (Yan and Han, 2002). In this algorithm, the graph is traversed using depth first search method. Since a graph can be represented by more than one DFS code, the graph is assigned to the first DFS code found by pre order search in the DFS code tree. This code is called the minimum DFS code and is used as an canonical label of this graph.

3.2.4. Input Type

There are two different types of graph datasets used in frequent subgraph mining: Transactional dataset and single large dataset. When calculating the frequency of a graph in the transactional graphs, it is necessary to calculate the number of transactions that contain this graph. However, since there are no transactions in the single large graph, how the frequency of a subgraph is calculated is an important issue. While calculating

the frequency of a subgraph in a single large graph, it is calculated how many times the embeddings of this subgraph are encountered in the single large graph. The isomorphic graphs to a graph in a single large dataset is called the embeddings of this graph. One of the most important problems with single large graphs is the overlap of embeddings of a subgraph. Because the overlap graphs can cause the failure of downward closure property (Kuramochi and Karypis, 2005). Another feature that separates single large graphs from transactional graphs is the need for more memory.

3.2.5. Graph Type

Graphs used in the problem of frequent subgraph mining can be undirected or directed graphs and multiple edges can be allowed between the graph nodes. Because of the direction between the nodes of a directed subgraph, there are more subgraphs than the same undirected graph. For this reason, the subgraphs obtained from a directed graph are less frequent and their computation time is shorter.

3.2.6. Nature of Graph

There are two types of graphs used in the problem of frequent subgraph mining: Static graphs and dynamic graphs. Static graphs do not change over time and can be stored in a database. Algorithms that provide solutions for frequent subgraph mining problems are usually suitable for static graphs. Dynamic graphs are constantly changing graphs. In these graphs, the change can be in the form of new nodes or edges addition, deletion or modification of the existing nodes or edges. Since dynamic graphs vary continuously over time, it is not predictable how much memory is needed to hold these graphs after the changes. For this reason, it is difficult to store dynamic graphs in a database. In frequent subgraph mining algorithms, while the occurrences of candidates are calculated, the database is scanned from beginning to end for each candidate. Because dynamic graphs are constantly changing, for example, when graphs are updated with a stream, it is difficult to keep these streams in a database because each stream only arrives once, and it is not possible to scan the database more than once while calculating the frequencies of graphs. Another issue that needs attention in dynamic graphs is that when the incoming

changes are added to the graphs, each edge and node in this change must be considered separately. Each change should be made as soon as possible. One of the most important issues is that an infrequent subgraph may be frequent with updating later, or a frequent subgraph may not be frequent later. The obtained frequent subgraphs should be found with as few errors as possible.

3.2.7. Result Type

Frequent subgraph mining algorithms can be categorized according to the result set. Some frequent subgraph mining algorithms find all frequent subgraphs. This result set is called *exact* result. However, in some cases it is not useful to have all the frequent subgraphs. Instead of finding all frequent subgraphs, smaller and more meaningful frequent subgraphs such as closed frequent subgraphs (if a subgraph is closed, none of its superset have same support value with this subgraph) or approximate frequent subgraphs (superset of all frequent subgraph) are found.

3.3. Popular FSM Algorithms

In this section popular FSM algorithms are introduced and compared according to the approaches they use in FSM process and categorization. Limitations and performance comparison of algorithms are also given.

3.3.1. AGM Algorithm

The AGM algorithm (Inokuchi et al., 2000) is an apriori based algorithm that tries to find all frequent induced subgraphs in a graph dataset. The graph dataset consists of transactional sets and static graphs. The graphs used in this algorithm may be undirected or directed graphs and they are represented by adjacency matrix. Labels used in this algorithm are integer numbers, so the adjacency matrix is constructed according to the order of these vertices labels. But this method is not enough to make the adjacency matrix unique. In this study, canonical form is proposed for normal forms of adjacency matrices. Since a graph can be represented by more than one adjacency matrix, it has more than one

canonical form. A minimum canonical form according to the lexicographic order is used to represent the graph.

- **Candidate generation:** AGM algorithm uses the level-wise strategy while generating candidates.
- **Frequency evolution:** AGM algorithm scans entire database and apply subgraph isomorphism test for every candidate to calculate its frequency.
- **Result:** This algorithm finds all induced frequent subgraph.
- **Limitations:** Multiple database scan and not scalable.

3.3.2. FSG Algorithm

The FSG algorithm (Kuramochi and Karypis, 2004) is an apriori based algorithm and finds all frequent connected subgraphs in a transactional dataset. The graphs used in this algorithm are undirected, labelled, static graphs and they are represented by adjacency list. This algorithm calculates the canonical labels of subgraphs from its adjacency matrix. To calculate the canonical label of a graph, the algorithm first transforms the adjacency list into an adjacency matrix and uses some heuristics, such as vertex invariants, to narrow the complexity of finding the canonical label of a graph. It uses the minimum canonical label according to the lexicographic order to represent the graph.

- **Candidate generation:** FSG algorithm uses the level-wise strategy while generating candidates.
- **Frequency evolution:** The algorithm uses transaction lists to calculate the frequency of a graph.
- **Result:** This algorithm finds all connected frequent subgraphs.
- **Limitations:** The algorithm requires a large number of different edge labels to facilitate calculation of canonical label. Not scalable.

When the minimum support threshold is small, the FSG algorithm requires less computation time than the AGM algorithm. Both algorithms are not suitable for very large datasets.

Table 3.3. Categorization of FSM algorithms.

Algorithms	Algorithmic Approach	Algorithmic Design	Graph Representation	Input Type	Graph Type	Nature of Graph	Result Type
AGM	apriori based	serial algorithm	adjacency matrix	set of graphs	undirected/ directed & single edge	static	all frequent subgraphs
FSG	apriori based	serial algorithm	adjacency list	set of graphs	undirected & single edge	static	all frequent connected subgraphs
FFSM	apriori based & pattern growth	serial algorithm	CAM	set of graphs	undirected & single edge	static	all frequent subgraphs
MOFA	pattern growth	serial algorithm	adjacency list	set of graphs	undirected/ directed & single edge	static	approximate/ all frequent subgraphs
gSpan	pattern growth	serial algorithm	min DFS code	set of graphs	undirected & single edge	static	all frequent subgraphs
CloseGraph	pattern growth	serial algorithm	min DFS code	set of graphs	undirected/ directed & single edge	static	all closed frequent subgraphs
GERM	pattern growth	serial algorithm	min DFS code	single large graph	undirected & single edge	dynamic	all frequent subgraphs
Stream FSM		serial algorithm		single large graph	undirected & single edge	dynamic	all frequent subgraphs
Time Evolving Graph	pattern growth	serial algorithm	min DFS code	single large graph	undirected & multi edge	dynamic	all frequent subgraphs
Subdue	pattern growth	serial algorithm	adjacency matrix	single large graph	undirected/ directed & single edge	static	approximate/ all frequent subgraph
SEuS	pattern growth	serial algorithm		single large graph	directed & single edge	static	approximate/ all frequent subgraphs
HSIGRAM	apriori based	serial algorithm	CAM	single large graph	undirected & single edge	static	approximate/ all frequent subgraphs
VSIGRAM	pattern growth	serial algorithm	CAM	single large graph	undirected & single edge	static	approximate/ all frequent subgraphs
FSM-H	apriori based	parallel algorithm	adjacency list	set of graphs	undirected & single edge	static	all frequent subgraphs
gSpan-H	apriori based	parallel algorithm	min DFS code	set of graphs	directed & single edge	static	all frequent subgraphs
p-MOFA	pattern growth	parallel algorithm	adjacency list	set of graphs	undirected & single edge	static	all frequent subgraphs
p-gSpan	pattern growth	parallel algorithm	min DFS code	set of graphs	undirected & single edge	static	all frequent subgraphs

3.3.3. FFSM Algorithm

The FFSM algorithm (Huan et al., 2003) is both an apriori based algorithm and a pattern-growth based algorithm. Apriori-based algorithms use the join operation (level-wise strategy) to generate candidates, while Pattern-growth based algorithms use the extension operation to generate candidates. The FFSM algorithm proposes two new operations by using join and extension operations: FFSM-join and FFSM-extension. The graphs in this work are single, undirected, labelled and connected/unconnected, static graphs and they are represented by the canonical adjacency matrices(CAM). In this study, the graphs are transformed into canonical forms to describe the graphs uniquely. A graph can have more than one canonical form, depending on the adjacency matrices that define it. Among these canonical forms, the maximal code according to the lexicographic order is determined as the canonical form of this graph. This maximal code is used to determine if two graphs are isomorphic.

- **Candidate generation:** FFSM algorithm uses join and extension strategy while generating candidates.
- **Frequency evolution:** The algorithm uses embedding lists to calculate the frequency of a graph.
- **Result:** This algorithm finds all connected frequent subgraphs.
- **Limitations:** Not scalable.

The FFSM algorithm performs better than the gSpan algorithm according to tests on real and synthetic datasets (Huan et al., 2003).

3.3.4. MOFA Algorithm

MOFA (Borgelt and Berthold, 2002) is a pattern-growth based algorithm that finds connected subgraphs in a transactional molecule dataset. The graphs used in this study are static and they are represented by adjacency list.

- **Candidate Generation:** All discovered embeddings are stored in an embedding list. This algorithm uses extension strategy to generate candidates but this extension strategy is restricted to only this embedding list.
- **Frequency Evolution:** This algorithm applies the subgraph isomorphism test to embedding list to calculate the frequency of graphs and uses some pruning methods to facilitate the frequency calculation.
- **Result:** This algorithm finds all frequent subgraphs.
- **Limitations:** Because the MOFA algorithm generates many duplicates, the frequent subgraphs generated may not be frequent actually.

3.3.5. gSpan Algorithm

The gSpan algorithm (Yan and Han, 2002) is a pattern-growth based algorithm that finds all the frequent subgraphs in a dataset. The graphs used in this study are undirected simple graphs and they are represented by adjacency list. The general properties of the Apriori-based algorithms and the problem they face are the costly candidate generation and the subgraph isomorphism test. To overcome these problems, gSpan algorithm proposes a new frequent subgraph mining algorithm that does not generate candidate subgraph. The gSpan algorithm propose a new method called DFS code to represent graphs. Since the aim is to represent a graph uniquely, minimum DFS code according to the lexicographic order is used.

- **Candidate generation:** While apriori based algorithms are based on the breadth first strategy to generate candidates, the gSpan algorithm constructs a hierarchical search tree called the DFS code tree instead of generating candidates. Each node of DFS code tree is a minimum DFS code that represents a graph. The minimum DFS code of a graph is the first code obtained when encounter this graph while traversing DFS-tree according to the pre-order search. The DFS code is grown by adding one edge to each node until represent all dataset. However, each node of DFS-tree must represent the minimum DFS code of a graph, otherwise this node is pruned. The subgraphs are discovered by traversing DFS tree according to the pre-order.

- **Frequency evolution:** The transaction lists of each discovered subgraph are stored and these lists are used to calculate the frequency of a subgraph.
- **Result:** This algorithm finds all frequent subgraph.
- **Limitations:** Not scalable.

GSpan algorithm outperforms the FSG and MOFA algorithms.

3.3.6. CloseGraph Algorithm

The CloseGraph algorithm (Yan and Han, 2003) is an algorithm based on the gSpan algorithm, which finds only closed frequent subgraphs. The graphs used in this algorithm may be simple or non-simple, labelled or unlabelled, directed or undirected, connected or unconnected graphs and they are represented by adjacency list. There can be more than one edge between nodes. This algorithm finds only closed frequent subgraphs instead of all frequent subgraphs to mine large datasets more efficiently and it also narrows the search space by suggesting two new concepts: equivalent occurrence and early termination. The aim of the algorithm is to find all closed frequent subgraphs but, in some cases, early termination may fail, and all closed frequent subgraphs may not be found. The completeness of the algorithm is guaranteed by detecting situations in which early termination may fail.

- **Candidate generation:** Since the CloseGraph algorithm is a gSpan based algorithm, it uses the rightmost extension while generating candidate subgraphs.
- **Frequency evolution:** This algorithm uses the transaction lists to calculate the frequency of a graph.
- **Result:** CloseGraph algorithm finds all closed frequent subgraphs.
- **Limitations:** Failure detection takes a lot of time.

CloseGraph algorithm outperforms the gSpan algorithm, but it may miss some important patterns.

3.3.7. SIGRAM Algorithm

The aim of this algorithm (Kuramochi and Karypis, 2005) is to find frequent subgraphs in single large sparse graphs. The graphs used in the algorithm are undirected, labelled and connected graphs. In this study, three different formulations are used to find frequent subgraphs. First formulation is exact discovery problem. In this formulation, all frequent subgraphs that satisfy the specified threshold value are determined in single large graph. Second formulation is approximate discovery. In this formulation, instead of all frequent subgraphs in a single large graph, as many frequent subgraphs as possible that satisfy the certain threshold values are determined. Third is upper bound discovery problem. According to this formulation, an upper limit is specified and instead of finding all frequent subgraphs, as few frequent subgraphs as possible are determined satisfying the upper limit. Two new methods are proposed in this study: HSIGRAM and VSIGRAM.

HSIGRAM algorithm: HSIGRAM algorithm is an apriori based algorithm and it is applied on single large graph. Graphs are represented by canonical adjacency matrix. This algorithm depends on the breadth first strategy.

- **Candidate generation:** HSIGRAM algorithm uses the level-wise strategy while generating candidates.
- **Frequency evolution:** HSIRAM algorithm uses MIS method to calculate frequency of a candidate. There are two different frequency calculation in this method exact and approximate. The first method, frequency calculation is done as follows: The frequency of a subgraph is equal to the size of the overlap graph of maximum independent set of this subgraph. The second frequency calculation is done as follows: The frequency of a k-size subgraph is equal to the frequency of the subgraph which has the lowest frequency among all (k-1)-size subgraphs of this k-size subgraph. The minimum value of these two frequency is determined as the frequency of the subgraph.
- **Result:** HSIGRAM algorithm finds all or approximate frequent subgraphs.
- **Limitations:** HSIGRAM algorithm is not an efficient method.

HSIGRAM algorithms outperforms the SEuS and Subdue algorithms.

VSIGRAM algorithm: VSIGRAM algorithm is a pattern-growth based algorithm and it is applied on single large graph. Graphs are represented by canonical adjacency matrix. VSIGRAM algorithm depends on the depth first search strategy.

- **Candidate generation:** VSIGRAM algorithm uses the extension strategy while generating candidates.
- **Frequency evolution:** In the VSIGRAM algorithm based on the MIS measure while calculating the frequency of a (k+1)-size subgraph.
- **Result:** VSIGRAM algorithm finds all frequent subgraphs.

Since VSIGRAM algorithm requires less subgraph isomorphism checking, VSIGRAM algorithm outperfoms the HSIGRAM algorithm.

3.3.8. GERM Algorithm

The aim of GERM algorithm (Berlingerio et al., 2009) is to find graph evolution rules that describe the local and structural changes in a dynamic graph that constantly changing graphs over time. This graph evolution rules are derived from frequent subgraphs. Single large graphs are used as input and this graph is connected labelled graph. Edge labels specify the time stamps that indicate first appearance of this edge. The GERM algorithm finds frequent subgraphs that have different time stamps. That is, if two isomorphic subgraphs where all edges have the same time stamps, one of them is not considered while calculating the frequency. The purpose is to find frequent subgraphs that appear in different time snapshots. The GERM algorithm is a modification of the gSpan algorithm. It uses the main properties of the gSpan algorithm and adapts the gSpan algorithm to a single large graph. Since the frequency calculation for a transactional graph dataset and single large graph are different from each other, it also adapts the frequency calculation of gSpan algorithm.

- **Candidate generation:** The GERM algorithm uses the rightmost extension strategy while generating candidates.
- **Frequency evolution:** The GERM algorithm uses the minimum image based support computation strategy while calculating the frequency of a graph.

- **Result:** The GERM algorithm finds frequent subgraphs and derives graph evolution rules from these frequent subgraphs.

3.3.9. Stream FSM Algorithm

Stream FSM algorithm (Ray et al., 2014) finds frequent subgraph in a single large graph. The graph used in this study are undirected, labelled graph and multiple edges are not allowed. The input of this algorithm is the single large graph with stream updates. So, this single large graph is a dynamic graph. The algorithm is assumed that the node or edges are not deleted or modified. Only nodes and edges are added with a new stream. Updates come in batches and contain labelled nodes and edges. The aim of the algorithm is to transform the single large dynamic graphs into graph transactions. These graph transactions are mined by another frequent subgraph mining algorithms that are used graph transactions as input to find frequent subgraphs. Whenever a new batch arrives, this algorithm deals with the region that changes with the update of the graph, the frequent subgraphs in the updated graph are found and reported.

In this algorithm, first, a new incoming batch of updates is added to the graph. Each edge is selected as the anchor point and the neighborhoods around it are found. Each edge in the neighborhood is marked as extracted so that it will not come back in the next neighborhood. Extracted neighborhoods are considered graph transactions. Another frequent subgraph mining algorithms that mine graph transaction is used to find frequent subgraphs in these extracted graph transactions. Canonical labels and frequencies of these frequent subgraphs that found after the update are stored in a dictionary data structure. These operations are repeated for each new batch updates. At the end of each update, frequent patterns and their frequencies are updated.

Stream FSM algorithm outperforms the GERM and Subdue algorithms.

3.3.10. Mining Interesting Patterns and Rules in a Time Evolving Graph

The purpose of this algorithm (Miyoshi et al., 2011) can be summarized in three steps: The first is to find the frequent subgraphs in the dynamic graphs, the second is

to extract the graph evolution rules from these frequent patterns, and to make a graph-based summarization of these rules. The graphs used in this algorithm are undirected and labelled graphs. This algorithm is a modification of GERM algorithm and the input of this algorithm is single graph. The difference between them, the graphs in this algorithm may have multiple edges with different labels. Multiple edges are allowed between nodes because more than one relationship may have been established at different times.

- **Candidate generation:** Since this algorithm based on GERM algorithm, it uses the rightmost extension strategy while generating candidates.
- **Frequency evolution:** This algorithm stores occurrence lists that holds all occurrences of a subgraph in a single large graph. For all occurrence, the ratio of unique vertices of this occurrence to total number of vertices in single large graph is calculated. The minimum one is the support of this subgraph.
- **Result:** This algorithm extracts frequent subgraphs and graph evolution rules from these frequent subgraphs, as GERM algorithm does.

3.3.11. SUBDUE Algorithm

The Subdue algorithm (Holder et al., 1994) tries to find frequent subgraph in directed or undirected single large graph. All isomorphic graphs of discovered subgraph in the input graph is represented by a single vertex that denotes the code of discovered subgraph. The input data is encoded and compressed in this way.

- **Candidate generation:** In this algorithm, the candidate generation starts with a single vertex and a new edge is added to all possible ways as in the extension strategy. The minimum description length of the candidate graph generated after extension should not be less than the minimum description length of graph that prior the extension. Furthermore, other background knowledge provided by the user can be used to narrow the search space.
- **Frequency evolution:** The SUBDUE algorithm uses the minimum description length of a subgraph to calculates its frequency.
- **Result:** Since the SUBDUE algorithm uses the inexact graph matching while discovering the graphs, finds approximate frequent subgraphs.

3.3.12. SEuS Algorithm

SEuS algorithm (Ghazizadeh and Chawathe, 2002) tries to find frequent subgraph in single large graph. The graphs used in this algorithm are labelled, directed graphs. The subgraph of this single large graph should be connected. SEuS algorithm consist of three phases: summarization, candidate generation and counting phase. Especially for large graph, database scanning is too costly and inefficient. For this reason, int the summarization phase, SEuS algorithm summarizes the graph dataset and stores this summary in-memory. In the candidate generation phase, the algorithm tries to find frequent subgraph in this summarization to avoid scanning database. In the counting phase, the frequencies of subgraphs are calculated.

- **Candidate generation:** SEuS algorithm generates candidates by using extension strategy.
- **Frequency evolution:** The frequency calculation of the any candidate subgraph consists of two steps. First, the estimated support is calculated by using data summaries. If this estimated support satisfies the minimum support threshold, the exact support is calculated by using pointer to the parent of this subgraph on disk.
- **Result:** Since the frequent subgraph are searched in a summary of graph dataset, the output of algorithm is approximate frequent subgraphs which are the supper set of complete frequent subgraphs.

3.3.13. FSM-H Algorithm

FSM-H algorithm (Bhuiyan and Al Hasan, 2015) is a parallel FSM algorithm based on MapReduce, a distributed platform. MapReduce framework consist of two phases: Map and Reduce. In the Map Phase, input data is partitioned over the worker nodes and these worker nodes calculate local support. In the Reduce phase, the actual support of any subgraphs is calculated by collecting local support of this subgraph from worker nodes. The FSM-H algorithm runs iteratively. The output of iteration n is the n-size frequent subgraphs. To calculate the (n+1)-size frequent subgraph, the output of iteration n is used.

The FSM-H algorithm consists of three phases: Data partition, preparation and mining. In the data partition phase, the input is divided into partitions with equal number of graphs. Also, at this partition, all infrequent edges are omitted from the input. During the preparation phase, specific data structures are prepared for each partition and frequent subgraphs of this partitions are found. Then, the output of each iteration is written in HDFS. During the mining phase all possible frequent subgraphs are found.

- **Candidate generation:** To generate the candidates, this algorithm uses breadth first strategy and extension strategy. (n+1)-size candidate subgraph is obtained by adding a new edge between an existing node and newly introduced node, or between two existing nodes.
- **Frequency evolution:** The FSM-H algorithm uses the occurrence lists to calculate frequencies of graphs.
- **Result:** The FSM-H algorithm finds all frequent subgraphs.

3.3.14. gSpan-H Algorithm

The gSpan-H algorithm (Sangle and Bhavsar, 2016) is a parallel modification of gSpan algorithm that tries to find frequent subgraphs in large datasets based on the MapReduce method. The graphs used in the algorithm are simple, connected, labelled and directed graphs. gSpan-H algorithm consists of three phases as in FSM-H algorithm. The difference between the gSpan-H algorithm and the FSM-H algorithm, the gSpan-H algorithm avoids costly candidate generation and subgraph isomorphism checking.

- **Candidate generation:** The gSpan-H algorithm uses breadth first strategy and extension strategy, as FSM-H algorithm does.
- **Frequency evolution:** The gSpan-H algorithm uses the minimum DFS code to check whether two graphs are isomorphic, as gSpan algorithm does.
- **Result:** This algorithm finds all frequent subgraphs.

The gSpan-H algorithm outperforms the FSM-H algorithm.

3.3.15. p-MOFA and p-gSpan Algorithm

The previously developed MOFA and gSpan algorithms are suitable for small datasets. In this work, parallel versions of algorithms are proposed to work on large datasets: p-MOFA and p-gSpan (Meinl et al., 2006). This algorithm is a thread-based algorithm that runs on a shared memory system with 12 processors. The general concepts of p-MOFA and gSpan algorithms are the same as the algorithms presented earlier (MOFA, gSpan) and only the work to be done is distributed among multiple workers. For each worker(thread) there is a stack that holds the nodes that have not been checked before. Each thread tries to find frequent subgraphs in its stack and keeps these frequent subgraphs. Since it is not known exactly how much work the thread will do, a proper load balancing needs to be done. If any work of thread is done before the other threads, half of the work in the stack of any running threads is assigned to empty thread. In this case, however, there will still be a load imbalance because this thread will finish its work before others. In the MOFA algorithm, idle threads are kept in a list that each thread can access. Each thread checks this list when it completes its work in its main loop, and if there is an idle thread, it sends half of the work in its stack to this idle thread. Idle threads wait the end of work in main loop of running threads, if the threads are too busy, the idle threads may have to wait too long. In p-gSpan algorithm, global list keeps running threads instead of idle threads. If any thread finishes its work, it omits itself from list. It then takes half the work of the other running thread in the list. This process is carried out iteratively. The p-gSpan algorithm outperforms the p-MOFA algorithm.

CHAPTER 4

PARALLEL FSM ALGORITHM IMPLEMENTATIONS

Social media data is a large dataset that have a lot of trivial data and represented by graphs. Mining social media data may provide a different perspective and may find previously unknown relations on this data. Frequent subgraphs on social media can be used to detect communities and their social structure, to analyze human behavior and to produce suggestions for a recommendation system.

According to the algorithms examined in this thesis, most FSM algorithms suffer from too many candidate generation and subgraph isomorphism testing to calculate the frequencies of candidates. FFSM and gSpan algorithms are the most efficient algorithms among popular FSM algorithms. However, the FFSM algorithm has a significant disadvantage compared to the gSpan algorithm. The FFSM algorithm keeps all the discovered embeddings in an embedding list to avoid the subgraph isomorphism test. This situation requires a lot of memory usage. Especially for large datasets, these embedding may not fit in memory. gSpan algorithm significantly reduces search space using a more compact data structure called DFS tree instead of candidate generation. It also significantly simplifies the subgraph isomorphism test with a new introduced canonical labelling system, called minimum DFS code. Since this algorithm does not store the isomorphic graphs, it requires less memory than the FFSM algorithm.

The general limitation of most FSM algorithms is that they are not suitable for large datasets. The generated candidates may not fit in a single machine or working on a single machine is not an efficient method. Because social media data is so large and gSpan algorithm is also not suitable for very large datasets as the other FSM algorithms. A parallel FSM algorithm is needed to mine social media data. Since the gSpan algorithm performs better than the other FSM algorithms, in this thesis, two different parallel implementations of the gSpan algorithm have been performed. One of the implementation is a Spark based algorithm that can operate both locally on a single machine or on a cluster. The second algorithm is a multi-threaded algorithm based on Fork/Join framework that works on a single machine.

The most important part of the gSpan algorithm is the construction of the DFS

tree. For this reason, the DFS code tree is described in more detail in this section:

DFS code and DFS code tree: Using the new canonical labelling system, while traversing a graph according to the depth first strategy, each vertex is assigned an edge code according to the discovery time. Let v_i and v_j be two vertices with an edge between them. i and j give information about the discovery time of these vertices. If $i < j$, the node v_i is the previously discovered before node v_j. According to the discovery times of nodes, each edge is represented 5-tuple (Yan and Han, 2002):

$\langle$ node v_i identifier, node v_j identifier, node v_i label, edge label, node v_j label $\rangle$.

A DFS code of a graph consist of all edge codes representing the edges of this graph. At the level m in the DFS code tree, there are DFS codes belonging to (m-1)-size subgraphs. As the DFS code tree grows, child nodes ((m + 1)-th level nodes) are obtained by adding a new edge to the m-th level nodes. However, in this case the same subgraphs can be obtained more than once. These graphs are called duplicated graphs. The gSpan algorithm uses the rightmost extension technique to avoid duplicate graphs, and only extension on the rightmost path is allowed.

The nodes that do not have the minimum DFS code in the DFS code tree are pruned. This reduces the size of the DFS code tree and avoids the generation of redundant candidates.

If two subgraphs are assigned the same canonical label, these two graphs are isomorphic to each other. In this algorithm, the DFS code growth and subgraphs isomorphism test are performed in a single procedure instead of a separate procedures, so that the computation time is significantly reduced.

Both implementations are based on gSpan algorithm. Therefore, the input of these inplementations are transactional dataset and they constructs DFS tree by using righmost extension strategy to represent the all graphs in a dataset. The graphs used in these implementations are simple, undirected, labelled and connected graphs. Both implementations use min DFS code strategy to facilitate the subgraph isomorphism and use transaction list strategy to calculate the frequency of subgraphs.

4.1. Spark based Parallel gSpan algorithm Implementation

This algorithm is a parallel modification of the gSpan algorithm implemented using the Apache Spark framework and the GraphFrame package provided by this framework. This algorithm can operate both locally on a single machine with multiple threads and on a standalone cluster of multiple machines. Algorithm 1 demonstrates the parallel

Algorithm 1 Spark based Parallel gSpan Algorithm

```
1:  Create Spark configuration and context;
2:  GF ← readData (sc, fileName); sc;                    ▷ (GF represents the GraphFrames
3:  GF, E ← removeInfrequents (GF, minSupport);          ▷ (E represents EdgeCodes)
4:  Initialize S to empty DFS code list;
5:  foreach edge code e ∈ E do
6:      g_f ← ConstructOneEdge (GF, e)
7:      support ← size of g_f
8:      Initialize s with e, g_f,support                 ▷ (s is DFS Code)
9:      SubgraphMining (GF, S, e, s)
10:     removeEdge (GF, e)
11: end foreach
12: procedure CONSTRUCTONEEDGE(GF, e)
13:     foreach graph G in GF do
14:         S ← all frequent one-edge frequent subgraphs that contain e in G;
15:         S^1 ← S^1 ∪ S
16:     end foreach
17:     return S^1
18: procedure SUBGRAPHMINING(GF, S, e, s)
19:     if isMinDFScode(s) then
20:         S ← S ∪ {s}
21:     Enumerate(GF, s);
22:     foreach child c of s do
23:         if support(c) ⩾ minSupport then
24:             s ← c
25:             SubgraphMining(GF, S, e, s)
26:     end foreach
27: procedure ENUMERATE(GF, s)
28:     foreach g ∈ s.g_f do                 ▷ (s.g_f is the GraphFrames that contain s)
29:         find all children c of s occurs in g
30:         foreach child c do
31:             c.g_f ← c.g_f ∪ g.id         ▷ (c.g_f is the graph id set that contain c)
32:         end foreach
33:     end foreach
```

gSpan algorithm that works as Spark application.

Step 1 (line 1): Spark properties, such as master URL, memory, cores and application name are configured in the first step of the algorithm. Then, a Spark configuration

is created that represents a connection to work in parallel on a Spark cluster. By default Spark starts up in local mode on a single machine rather than distiributed environment. Spark can also start on a Spark standalone cluster to run Spark against multiple machines. The master url decides whether a spark application will run in a cluster or local mode. In the Spark implementation, the collections are parallelized by using Spark context's *parallelize* method. A distributed dataset that can be operated on in parallel is created with specified number of partitions and spark runs one task for each partitions.

Step 2 (line 2): The graphs are read from the text document. "$t\ \#\ 0$" specifies a graph with id is 0. $v\ 0\ 6$ indicates a vertex of the graph 0. The first entry after the expression v specifies the id of this vertex, the second entry specifies the label of this vertex. e $0\ 1\ 2$" indicates an edge of the graph 0. The first entry after the expression *e* specifies the id of source vertex of this edge, the second entry specifies the id of destination vertex of this edge and the third entry specifies the label of this edge. The vertices informations, ids and labels, are stored in a DataFrame. The edges informations, vertices ids and labels that forming this edge and edge labels, are stored in another DataFrame. *GraphFrames* are constructed by using vertex and edge *DataFrames* to represent the graphs in the text document.

Step 3 (line 3): After the construction of graphs, infrequent edges and vertices that do not satisfy the minimum support are extracted from the graph dataset. Then frequent vertices and edges are relabelled and sort in decreasing frequency. Edge codes are created to represent frequent edges. These edge codes are stored in the edge code set. The edge codes that do not satisfy minimum support are removed from edge code set.

Step 4 (line 6-7): In the ConstructOneEdge (GF, e) method, for each edge that pointed by the edgecode, all graphs that contain current edge are found in the graph dataset. The number of these graphs indicates the support of this edge.

Step 5 (line 8): The DFS Code of current edge is constructed by using edge code, graph sets contain this edge and its support.

Step 6 (line 9): Subgraph mining is the one of most important part of gSpan algorithm. The first step in the subgraph mining phase is to check whether the DFS code representing a graph is minimum. In the isMinDFSCode (s) method, the DFS tree that represents the graphs is traversed according the pre order search.

If a code is minimum, in the Enumerate (GF, s) method, for each graph where this DFS code takes place in the graph dataset, all forward or backward edges are found by adding one new edge to this code. They are called children of this DFS code. In

this method, to find the frequency of a DFS code, the number of children of this code is calculated for each graph containing that code in the graph dataset. Each graph has a unique id number to represent it. To find the number of children, the ids of the graphs containing the discovered children are stored.

These children are obtained by adding a new edge to previous DFS code. If any children are satisfied the minimum support, the edge code of new edge is added to DFS code. This procedure continues by calling SubgraphMining (GF, S, e, s) method until all graphs in the graph dataset are represented in the DFS code Tree.

Step 7 (line 10): The edge that corresponds the current edge code is removed from graph dataset to reduce the search space.

Steps 4, 5, 6 and 7 are repeated for each edge code to represent the all graphs in DFS code tree.

4.2. Multi-thread based Parallel gSpan Algorithm Implementation

This algorithm is a multi-threaded parallel modification of the gSpan algorithm implemented using the Java Fork/Join framework that provide thread pool. The threads and jobs to be done are managed by this framework. One of the most important differences between these two algorithms is the data structures used. The Spark Framework provides GraphFrame data structure to represent graphs and its properties, but in this algorithm, custom classes are defined to represent vertices, edges and graphs properties in this algorithm. To create distributed collections, while the Spark context's *parallellize* method is used in the Spark implementation, but in this implementation, *par* method that provided by the SCALA language is used. Apart from these, both algorithms consist of methods which do the same work but use different data structures.

Step 1 (line 1): In the first step, thread pool is configured.

Step 2 (line 2): In the second step, the data is read from the text document and stored using the custom classes. An instance from vertex class represents the vertex id and its label, an instance from edge class represents the vertices ids and edge label that form this edge.

Step 3 (line 3): Infrequent vertices and edges are removed graph set by using *removeInfrequents* ($GS, minSupport$) method. In addition, in this method, frequent vertices and edges are relabelled and edge codes are created for each frequent subgraph.

Step 4 (line 6): All one-edge frequent subgraphs and its support are found for each graph in the graph set that contain this edge.

Step 5 (line 8): DFS code is generated in a similar way to the previous algorithm.

Step 9 (line 9): Subgraph mining is performed in this step. First, it checks whether the DFS code is minimum. If the code is minimum, all children of this one-edge frequent subgraph are discovered using the *Enumerate* (GS, s) method. The DFS code is enlarged by adding all children that provide the minimum support to frequent subgraph.

Step 10 (line 10): The discovered edge is removed from graph dataset to reduce search space.

The steps between 5 and 11 are repeated for each edge code.

Algorithm 2 Multi-thread based Parallel gSpan Algorithm

```
1:  Create threads and thread pool;
2:  GS ← readData (fileName);                          ▷ (GS represents graph set)
3:  GS, E ← removeInfrequents (GS, minSupport);        ▷ (E represents EdgeCodes)
4:  Initialize S to empty DFS code list;
5:  foreach edge code e ∈ E do
6:      g_s ← ConstructOneEdge (GS, e)
7:      support ← size of g_s
8:      Initialize s with e, g_s,support                ▷ (s is DFS Code)
9:      SubgraphMining (GS, S, e, s)
10:     removeEdge (GS, e)
11: end foreach
12: procedure CONSTRUCTONEEDGE(GS, e)
13:     foreach graph G in GS do
14:         S ← all frequent one-edge frequent subgraphs that contain e in G;
15:         S^1 ← S^1 ∪ S
16:     end foreach
17:     return S^1
18: procedure SUBGRAPHMINING(GS, S, e, s)
19:     if isMinDFScode(s) then
20:         S ← S ∪ {s}
21:     Enumerate(GS, s);
22:     foreach child c of s do
23:         if support(c) ⩾ minSupport then
24:             s ← c
25:             SubgraphMining(GS, S, e, s)
26:     end foreach
27: procedure ENUMERATE(GS, s)
28:     foreach g ∈ s.g_s do                  ▷ (s.g_s is the graph set that contain s)
29:         find all children c of s occurs in g
30:         foreach child c do
31:             c.g_s ← c.g_s ∪ g.id          ▷ (c.g_s is the graph id set that contain c)
32:         end foreach
33:     end foreach
```

The complexity of subgraph generation is $O(2^{n^2})$ (n is the number of vertices of a graph). Because there are $\binom{n}{2} = \frac{1}{2}n(n-1)$ pairs of distinct points. If we do not allow loops or multiple edges, each of these pairs determines one possible edge, and we can have any subset of those possible edges. A set with $\binom{n}{2}$ members has $2^{\binom{n}{2}}$ subsets, so there are $2^{\binom{n}{2}}$ possible graphs without loops or multiple edges.

In the subgraph mining phase, all the children of a graph are found. And the subgraph mining phase is repeated recursively to calculate the frequency of each child and decide whether to add it to the DFS code tree. So, the time required for a subgraph support evaluation is $O(n^m)$ (n is the number of graph's vertices, m is the number of subgraph's vertices). Total complexity of this algorithm is $O(2^{n^2} * n^m)$.

CHAPTER 5

EXPERIMENTAL RESULTS

In this thesis, two different parallel gSpan algorithms are implemented. The first is an algorithm that is thread-based and use the Fork/Join framework. The second algorithm is based on Spark framework, that is, it can operate locally or on the cluster which is based on the cluster manager. The performance evaluation of the parallel gSpan algorithms have been performed on two real world datasets. First is real chemical compound dataset is used to evaluate the performance of the parallel gSpan algorithms. As social media data, the friendster social network dataset (Leskovec and Krevl, 2015) provided by Stanford University is used.

Nodes for chemical compound dataset represent atoms, edges represent bonds between atoms. In the chemical compound dataset, the edges representing bonds have different edge labels, since different atoms have different bonds. The label of a node represents atom name. Each node has a unique id. This dataset contains 340 graphs. Friendster is an online gaming networking site where users can make friends with each other, and the Friendster social network allows users to create a group that other members can join later. The Friendster dataset needs to be transformed into the input form of the parallel gSpan algorithm. There are about 5000 communities in this dataset and each community is regarded as a different graph. However, all edge labels are the same because there is only a friend relationship between the users in the friendship dataset. Nodes represent users, and the user id is used both as a node id and as a node label.

Friendster dataset contains only one edge label. But the different edge label significantly affects the finding isomorphic graphs in a dataset. To monitor the affect of different edge labels, both implementations are also tested on chemical compound dataset as well as social media dataset.

All experiments are done on two different machines. The first machine has 8GB RAM, windows 7 operating system and i7- 6700HQ processor with 8 cores. The second machine has 8GB RAM, a windows 10 operating system and i7-7500U processor with 4 cores. The datasets are first tested for multi-thread-based algorithm that based on Fork/Join framework. Second, the datasets are tested while running the Spark based al-

gorithm locally. In addition, both algorithms have been tested on two different computers with 8 and 4-core processors. Third, the datasets are tested for a Spark-based algorithm running on a standalone cluster that consist of two different machines.

5.1. The Results for Chemical Compound Dataset

The chemical compound dataset is evaluated from 4 different perspectives. First, the number of frequent subgraphs obtained according to different minimum support values is evaluated. Second, the run-time of the multi-thread based implementation is evaluated. Then, run times of Spark based implementation that performs both in local and on a cluster is evaluated for this dataset.

The number of discovered frequent subgraphs for chemical compound dataset when minimum support changed are shown in Fig 5.1. This figure shows the number of frequent subgraphs with at least two edges. As the minimum support threshold is increased, the number of discovered frequent subgraphs decreases.

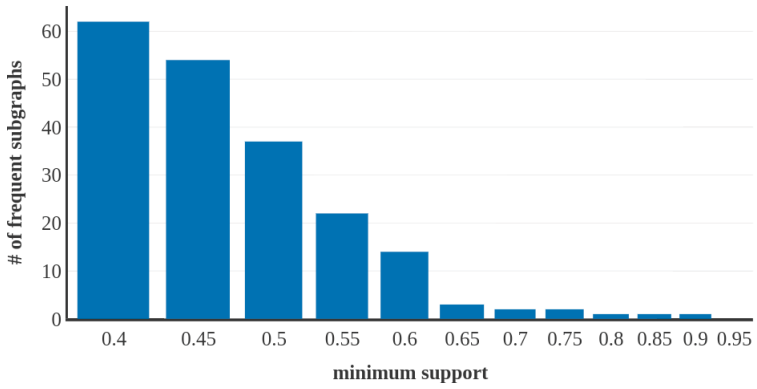

Figure 5.1. The number of frequent subgraphs for chemical dataset.

Figure 5.2 shows the run times on two different computers with 8 and 4-core processors of the multi-threaded implementation for chemical compound dataset. The performance of the multi-threaded implemenration is monitored according to a minimum support value that varies over a wide range. There is no frequent subgraph when the minimum support exceeds 0.9. But there are also frequent subgraphs when the minimum

support value drops below 0.4. Figure 5.3 shows the run times on two different computers

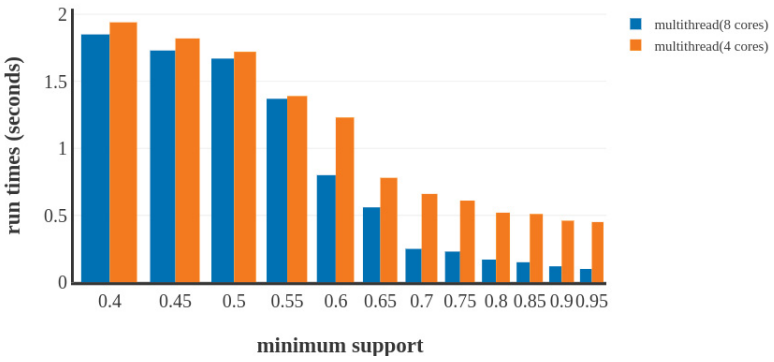

Figure 5.2. Run times of multi-threaded implementation for chemical compound dataset.

with 8 and 4-core processors and run times on standalone cluster with two machines of the Spark based implementation. The Spark implementation cannot perform when the minimum support value drops below 0.6. For this reason, performance evaluation in a much smaller range is possible.

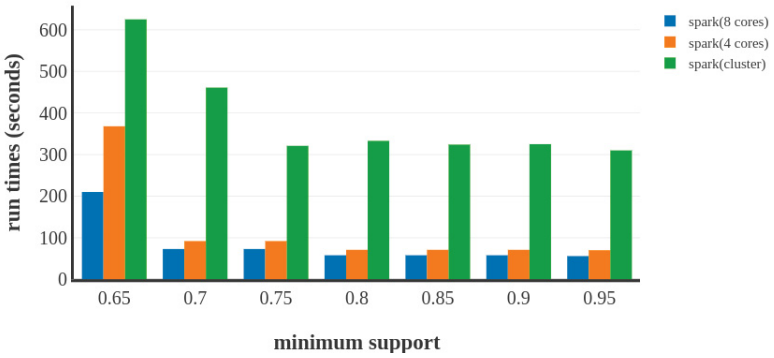

Figure 5.3. Run times of Spark-based implementation on different environment for chemical dataset.

The Spark based implementation running on the cluster has shown performance only for very small ranges and very large thresholds. The implementation cannot perform

when the minimum support value drops below 0.7. Table 5.1 shows the result set of two implementations for chemical compound dataset.

Table 5.1. Result set for chemical compound dataset.

minimum support	**multi threaded run time (seconds) (8 cores)**	**multi threaded run time (seconds) (4 cores)**	**Spark run time (seconds) (local) (8 cores)**	**Spark run time (seconds) (local) (4 cores)**	**Spark run time (cluster) (seconds)**	**# of frequent subgraphs**
0.95	0.10	0.45	56	70	310	0
0.9	0.12	0.46	58	71	325	1
0.85	0.15	0.51	58	71	324	1
0.8	0.17	0.52	58	71	333	1
0.75	0.23	0.61	73	92	321	2
0.7	0.25	0.66	73	92	461	2
0.65	0.56	0.78	210	368		3
0.6	0.8	1.23				14
0.55	1.37	1.39				22
0.5	1.67	1.72				37
0.45	1.73	1.82				54
0.4	1.85	1.94				62
0.35	2.02	2.09				67
0.3	2.12	118.77				75

5.2. The Results for Friendster Dataset

The friendster dataset is also evaluated from 4 different perspectives according to the number of frequent subgraphs for different minimum support threshold and the run-times for both two different implementations on two different environments.

The number of discovered frequent subgraphs for friendster dataset according to the different support value are shown in Fig 5.4. As the minimum support threshold increases the number of discovered frequent subgraphs decrease, as in the chemical compound dataset. The results can be monitored in the friendster dataset for very small minimum support values. Frequent subgraphs are not found when the minimum support value exceeds 0.01.

Figure 5.5 shows the run times on two different computers with 8 and 4-core processors of the multi-threaded implementation. Since the multi-thread based implementation performs well on this dataset, performance evaluation can be done in a wide range

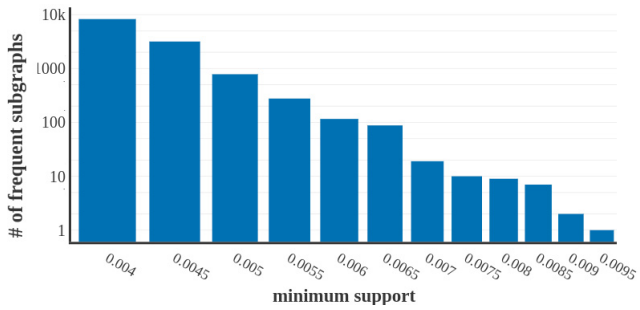

Figure 5.4. The number of frequent subgraphs for friendster dataset.

for minimum support value. As the number of subgraph isomorphism test decreases with the decrease of the support value, the run time of implementation also decreases.

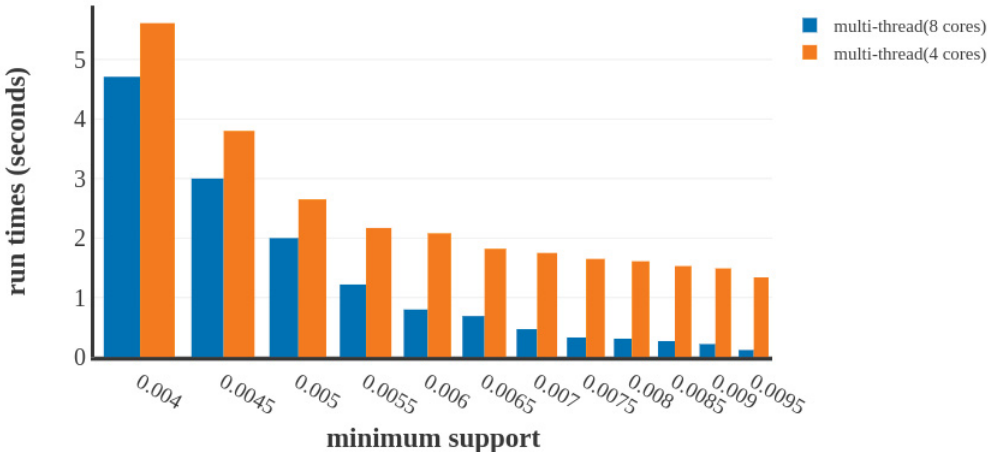

Figure 5.5. Run times of multi threaded implementation for friendster dataset.

Figure 5.6 shows the run times of the Spark based implementation on two different computers with 8 and 4-core processors for different support values. This figure also shows the run times of Spark based implementation on cluster. The performance of the Spark based implementation can be tested in a very small range as in the chemical compound data set. When the minimum support value is less than 0.007 value, the per-

formance of the Spark based implementation cannot be observed both in locally and on cluster. Table 5.2 shows the result set of two implementations for friendster dataset.

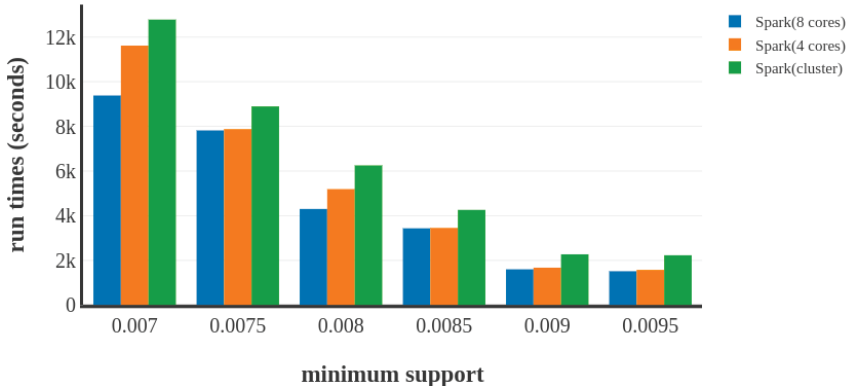

Figure 5.6. Run times of Spark-based implementation for friendster dataset.

Table 5.2. Result set for friendster dataset.

minimum support	**multi threaded run time (seconds) (8 cores)**	**multi threaded run time (seconds) (4 cores)**	**Spark run time (seconds) (local) (8 cores)**	**Spark run time (seconds) (local) (4 cores)**	**Spark run time (cluster) (seconds)**	**# of frequent subgraphs**
0.0095	0.32	1.54	1516	1573	2230	1
0.009	0.22	1.49	1602	1673	2272	2
0.0085	0.27	1.53	3437	3453	3261	7
0.008	0.31	1.61	4303	5192	3254	9
0.0075	0.33	1.65	7821	7877	5895	10
0.007	0.47	1.75	9385	11617	6783	19
0.0065	0.69	1.82				88
0.006	0.8	2.08				116
0.0055	1.22	2.17				277
0.005	2	2.65				784
0.0045	3	3.80				3170
0.004	4.71	5.61				8283

As the multi threaded algorithm gives good results for two datasets, it also tested for different dataset sizes for friendster dataset. First, the number of frequent subgraphs is evaluated according to different dataset sizes. The results are displayed for 6 different sizes of friendster dataset. Then the run time of implementation is evaluated. As the size of dataset increase, the number of frequent subgraphs and run time of implementation

also increase. Figure 5.7 shows the number of frequent subgraphs. Figure 5.8 shows the run times for different dataset sizes. The minimum support value during these evaluations is assumed to be 0.003.

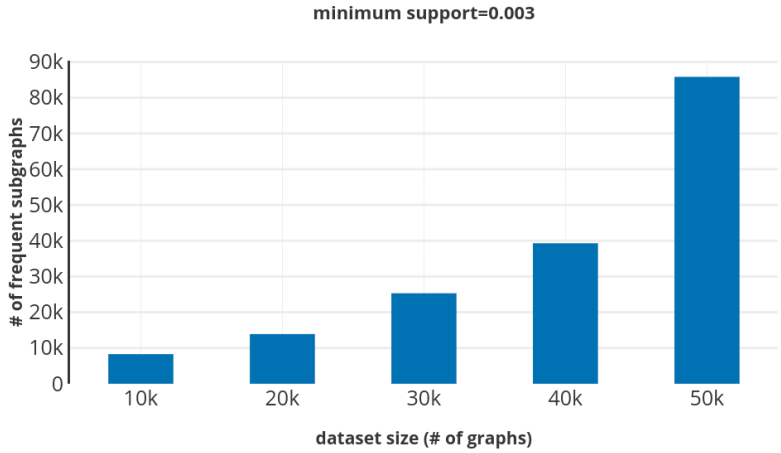

Figure 5.7. The number of frequent subgraphs for different dataset sizes.

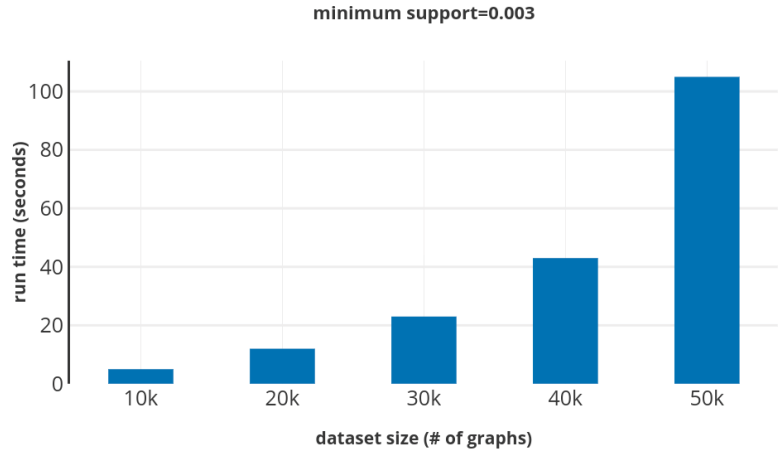

Figure 5.8. Run times of multi thread implementation for different dataset sizes.

The multi thread based algorithm is also tested against different thread counts. When the number of threads increases, the run time of the algorithm decreases. Figure 5.9 shows the run times for different thread counts.

Figure 5.10 represents the number of frequent subgraphs according to the different support threshold for different dataset size.

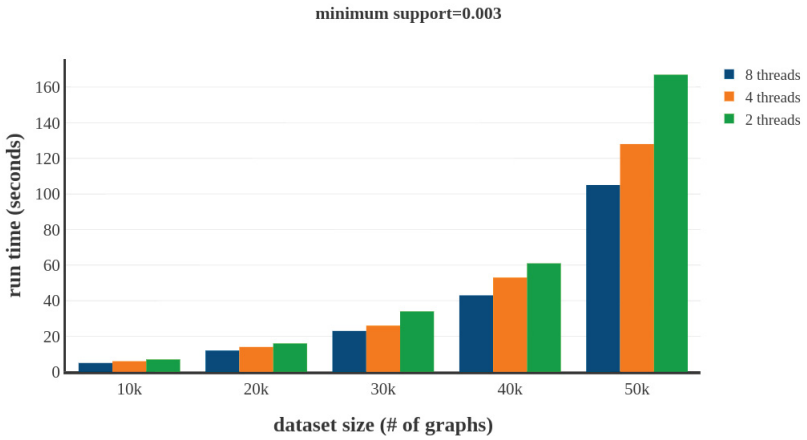

Figure 5.9. Run times of multi thread implementation for diferent thread counts.

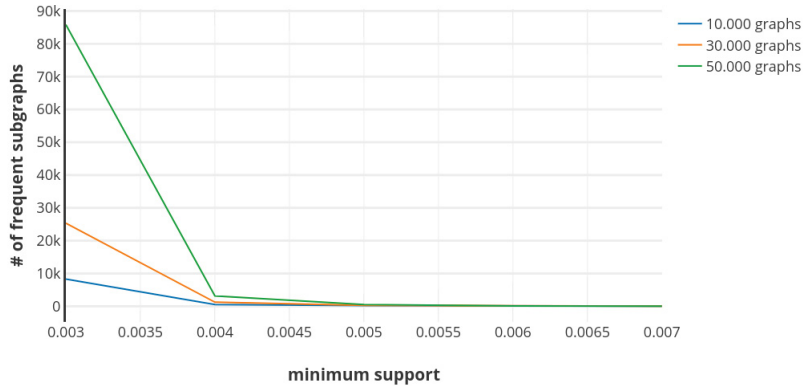

Figure 5.10. The number of frequent subgraphs for different supports and dataset sizes.

5.3. Discussion on Results

The multi-thread based implementation performs much better than the Spark based algorithm both datasets. For chemical compound dataset, it is roughly 100 times faster

than spark based implementation in locally and roughly 150 times faster than spark based implementation on cluster. For friendster dataset, multi threaded implementation is roughly 4500 times faster than spark based implementation in locally and roughly 5500 times faster than spark based implementation on cluster. When the dataset size and the number of vertices and edges in a graph increase, the detecting of isomorphic graphs becomes more difficult and the run time of algorithm grows substantially. Since there are too many factors that affect the algorithm's running time, it is difficult to exactly know how much better the multi threaded implementation is better than the spark implementation.

Although the chemical compound dataset contains less graph than the friendster dataset, the run time for both algorithms is higher for the chemical dataset. This is because, while all edge labels are the same in the friendster dataset, there are multiple distinct edge labels in the chemical dataset. Multiple different edge labels also make it difficult to finding isomorphic graphs and it is significantly affects the run time of algorithm.

The performance of Spark based algorithm while running separately in local of two computers is better than while running on a cluster with these two computers. Since the computer are in distiributed environment on the cluster, the communication of computers requires extra time. Spark framework is suitable not suitable for FSM on transactional dataset and it gives bad results. Because the Spark framework performs better on large datasets rather than small datasets. The size of these datasets is not enough for Spark framework to give good results. However when we increase the dataset size, the number of subgraph isomorphism tests and complexity increases exponentially. Because subgraph isomorphism problem is an NP-complete problem.

When the processor core count increases the number of available threads that run at the same time increases. 8 threads are executed on the 8-core machine, 4 threads are executed on the 4-core machine. So the performance of both algorithms is better on a computer with an 8-core processor for both sets of data. The thread counts also affect the run time of implementation because when we increase the thread counts, run time of multi threaded algorithm decreases.

CHAPTER 6

CONCLUSION

A frequent pattern is a pattern that has more than a certain number of occurrences in a data set. Frequent patterns provide a different perspective by discovering the unforeseen, meaningful information and relations in huge dataset. Social media is an important example of huge datasets and mining a social media data provides interesting information about human behavior and human interaction. Generally itemsets, sequences and graphs are used as patterns in frequent pattern mining area. However, since the graphs are better represent the social media data, they are used as the pattern in social media mining area.

In this thesis, FSM on social media data is focused. The processes of FSM and the methods used in these processes are examined. In addition, a categorization is presented by examining popular subgraph algorithms and considering the different aspects of these algorithms. Within the scope of this thesis, by considering limitations of algorithms and the challenges of the FSM as the too many candidate generation and subgraph isomorphism test, an existing FSM algorithm has been re-implemented based on two different parallel implementations to overcome these deficiencies.

The algorithms have been tested on two different real datasets. Frequently encountered chemical compounds are discovered in chemical compound dataset. By applying the frequent subgraph mining algorithms to the friendster dataset, that is a social media dataset, frequent friendship associations between two or more users in all communities are discovered.

It has been observed that the performance of a multi-thread based algorithm on a single machine is better than the two performances of the Spark based algorithm that run on both single machine and distributed environments, since the size of datasets are not sufficient to see the advantage of Spark framework.

It has been also observed that the multi-threaded algorithm based on the Fork/Join framework is suitable for large datasets and subgraph mining algorithms, whereas the algorithm based on the Spark framework is not suitable for the subgraph mining algorithms on transactional dataset. It has been also observed that the dataset size and thread counts also affects the run time of this implementation.

As a future work, since the social media data is constantly changing, these data can be represented by dynamic graphs or streams. Existing FSM algorithms can be extendend to work with both dynamic and static graphs or new FSM algorithm can be proposed. Within the scope of this thesis transactional dataset has been studied. But most social network are represented by single large graph. An existing FSM algorithm that work with single large graph can be re-implemented to work with in parallel manner. And simple graphs are used in this algorithm, that is there is at most one edge between two nodes. But in any social network there can be different edges between two nodes. Existing FSM algorithms can be modified to allow multiple edges between graphs.

REFERENCES

Acosta-Mendoza, N., A. Gago-Alonso, and J. E. Medina-Pagola (2012). Frequent approximate subgraphs as features for graph-based image classification. *Knowledge-Based Systems 27*, 381–392.

Aggarwal, C. C. and H. Wang (2010). Graph data management and mining: A survey of algorithms and applications. In *Managing and mining graph data*, pp. 13–68. Springer.

Barbier, G. and H. Liu (2011). Data mining in social media. In *Social network data analytics*, pp. 327–352. Springer.

Berlingerio, M., F. Bonchi, B. Bringmann, and A. Gionis (2009). Mining graph evolution rules. In *joint European conference on machine learning and knowledge discovery in databases*, pp. 115–130. Springer.

Bhuiyan, M. A. and M. Al Hasan (2015). An iterative mapreduce based frequent subgraph mining algorithm. *IEEE Transactions on Knowledge and Data Engineering 27*(3), 608–620.

Borgelt, C. and M. R. Berthold (2002). Mining molecular fragments: Finding relevant substructures of molecules. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pp. 51–58. IEEE.

Bringmann, B. and S. Nijssen (2008). What is frequent in a single graph? In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pp. 858–863. Springer.

Chakrabarti, D. and C. Faloutsos (2006). Graph mining: Laws, generators, and algorithms. *ACM computing surveys (CSUR) 38*(1), 2.

Cordella, L. P., P. Foggia, C. Sansone, F. Tortorella, and M. Vento (1998). Graph matching: a fast algorithm and its evaluation. In *Pattern Recognition, 1998. Proceedings. Fourteenth International Conference on*, Volume 2, pp. 1582–1584. IEEE.

Dean, J. and S. Ghemawat (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*(1), 107–113.

Garey, M. R. and D. S. Johnson (2002). *Computers and intractability*, Volume 29. wh freeman New York.

Ghazizadeh, S. and S. S. Chawathe (2002). Seus: Structure extraction using summaries. In *International Conference on Discovery Science*, pp. 71–85. Springer.

Hadoop, A. (2009). Hadoop.

Han, J., J. Pei, and M. Kamber (2011). *Data mining: concepts and techniques*. Elsevier.

Holder, L. B., D. J. Cook, S. Djoko, et al. (1994). Substucture discovery in the subdue system. In *KDD workshop*, pp. 169–180.

Huan, J., W. Wang, and J. Prins (2003). Efficient mining of frequent subgraphs in the presence of isomorphism. In *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pp. 549–552. IEEE.

Inokuchi, A., T. Washio, and H. Motoda (2000). An apriori-based algorithm for mining frequent substructures from graph data. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pp. 13–23. Springer.

Jiang, C., F. Coenen, and M. Zito (2013). A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review 28*(1), 75–105.

Kuramochi, M. and G. Karypis (2004). An efficient algorithm for discovering frequent subgraphs. *IEEE Transactions on Knowledge and Data Engineering 16*(9), 1038–1051.

Kuramochi, M. and G. Karypis (2005). Finding frequent patterns in a large sparse graph. *Data mining and knowledge discovery 11*(3), 243–271.

Lakshmi, K. and T. Meyyappan (2012). A comparative study of frequent subgraph min-

ing algorithms. *International Journal of Information Technology Convergence and Services 2*(2), 23.

Lea, D. (2000). A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pp. 36–43. ACM.

Leskovec, J. and A. Krevl (2015). {SNAP Datasets}:{Stanford} large network dataset collection.

McKay, B. D. et al. (1981). Practical graph isomorphism.

Meinl, T., M. Worlein, I. Fischer, and M. Philippsen (2006). Mining molecular datasets on symmetric multiprocessor systems. In *Systems, Man and Cybernetics, 2006. SMC'06. IEEE International Conference on*, Volume 2, pp. 1269–1274. IEEE.

Miyoshi, Y., T. Ozaki, and T. Ohkawa (2011). Mining interesting patterns and rules in a time-evolving graph. *Proc. Int. MultiConf. Engineers and Computer Scientists 2011 1*, 448–453.

Muttipati, A. S. and P. Padmaja (2015). Analysis of large graph partitioning and frequent subgraph mining on graph data. *International Journal of Advanced Research in Computer Science 6*(7).

Ray, A., L. Holder, and S. Choudhury (2014). Frequent subgraph discovery in large attributed streaming graphs. In *Proceedings of the 3rd International Workshop on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pp. 166–181.

Rehman, S. U., A. U. Khan, and S. Fong (2012). Graph mining: A survey of graph mining techniques. In *Digital Information Management (ICDIM), 2012 Seventh International Conference on*, pp. 88–92. IEEE.

Sangle, M. M. H. and P. S. A. Bhavsar (2016). gspan-h: An iterative mapreduce based frequent subgraph mining algorithm.

Schmidt, D. C. and L. E. Druffel (1976). A fast backtracking algorithm to test directed graphs for isomorphism using distance matrices. *Journal of the ACM (JACM) 23*(3), 433–445.

Shvachko, K., H. Kuang, S. Radia, and R. Chansler (2010). The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pp. 1–10. Ieee.

Spark, A. (2016). Apache spark: Lightning-fast cluster computing. *URL http://spark. apache. org*.

Ullmann, J. R. (1976). An algorithm for subgraph isomorphism. *Journal of the ACM (JACM) 23*(1), 31–42.

Yan, X. and J. Han (2002). gspan: Graph-based substructure pattern mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on*, pp. 721–724. IEEE.

Yan, X. and J. Han (2003). Closegraph: mining closed frequent graph patterns. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 286–295. ACM.

Yan, X., P. S. Yu, and J. Han (2004). Graph indexing: a frequent structure-based approach. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pp. 335–346. ACM.

Zimek, A., I. Assent, and J. Vreeken (2014). Frequent pattern mining algorithms for data clustering. In *Frequent Pattern Mining*, pp. 403–423. Springer.