

# ScholarWorks@GSU

## High Performance Frequent Subgraph Mining on Transactional Datasets

Authors	Jena, Bismita
Citation	Jena, Bismita (2019). High Performance Frequent Subgraph Mining on Transactional Datasets. Dissertation, Georgia State University. <a href="https://doi.org/10.57709/13988716">https://doi.org/10.57709/13988716</a>
DOI	<a href="https://doi.org/10.57709/13988716">https://doi.org/10.57709/13988716</a>
Download date	2026-04-11 00:13:47
Link to Item	<a href="https://hdl.handle.net/20.500.14694/3917">https://hdl.handle.net/20.500.14694/3917</a>

# HIGH PERFORMANCE FREQUENT SUBGRAPH MINING ON TRANSACTIONAL DATASETS

by

Bismita Jena

Under the Direction of Rajshekhar Sunderraman, PhD

## ABSTRACT

Graph data mining has been a crucial as well as inevitable area of research. Large amounts of graph data are produced in many areas, such as Bioinformatics, Cheminformatics, Social Networks, and Web etc. Scalable graph data mining methods are getting increasingly popular and necessary due to increased graph complexities. Frequent subgraph mining is one such area where the task is to find overly recurring patterns/subgraphs. To tackle this problem, many main memory-based methods were proposed, which proved to be inefficient as the data size grew exponentially over time. In the past few years several research groups have attempted to handle the frequent subgraph mining (FSM) problem in multiple ways. Many authors have tried to achieve better performance using Graphic Processing Units (GPUs) which has multi-fold

improvement over in-memory while dealing with large datasets. Later, Google's MapReduce model with the Hadoop framework proved to be a major breakthrough in high performance large batch processing. Although MapReduce came with many benefits, its disk I/O and non-iterative style model could not help much for FSM domain since subgraph mining process is an iterative approach. In recent years, Spark has emerged to be the De Facto industry standard with its distributed in-memory computing capability. This is a right fit solution for iterative style of programming as well. In this work, we cover how high-performance computing has helped in improving the performance tremendously in the transactional directed and undirected aspect of graphs and performance comparisons of various FSM techniques are done based on experimental results.

**INDEX WORDS:** Frequent Subgraphs, Isomorphism, Spark, Scala, Hadoop MapReduce, DB4O

HIGH PERFORMANCE FREQUENT SUBGRAPH MINING ON TRANSACTIONAL  
DATASETS

by

Bismita Jena

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

PhD

in the College of Arts and Sciences

Georgia State University

2019

Copyright by  
Bismita Srichandan Jena  
2019

HIGH PERFORMANCE FREQUENT SUBGRAPH MINING ON TRANSACTIONAL  
DATASETS

by

Bismita Jena

Committee Chair: Rajshekhar Sunderraman

Committee: Yanqing Zhang

Anu Bourgeois

Hendricus van der Holst

Electronic Version Approved:

Office of Graduate Studies

College of Arts and Sciences

Georgia State University

May 2019

## **DEDICATION**

There are many people who have influenced me during this phase. I would like to dedicate this to a few key people in my life. My academic advisor Dr. Sunderraman, without his trust and support I could not have imagined this happening. My Husband Axaya, who has been my support system throughout the journey. He handled the kids and family very nicely, I barely had to think how my family was managed. My kids Asmit and Ankit, these two have been very supportive children and I could do multiple things while raising them. My Mother, who was always ready to extend help whenever I needed her the most.

## **ACKNOWLEDGEMENTS**

I would like to thank my committee members Dr. Yanqing Zhang, Dr. Anu Bourgeois, and Dr. Hendricus van der Holst for giving their precious time and serving as the review committee members. I thank the entire computer science department faculties and staffs who have educated me and made me eligible to reach this level.

## TABLE OF CONTENTS

<b>ACKNOWLEDGEMENTS .....</b>	<b>V</b>
<b>LIST OF TABLES .....</b>	<b>IX</b>
<b>LIST OF FIGURES .....</b>	<b>X</b>
<b>LIST OF ABBREVIATIONS .....</b>	<b>XII</b>
<b>1 INTRODUCTION .....</b>	<b>1</b>
<b>1.1 Problem Statement.....</b>	<b>3</b>
<b>1.2 Motivation.....</b>	<b>4</b>
<b>1.3 Our Contribution .....</b>	<b>5</b>
<b>2 REVIEW OF FREQUENT SUBGRAPH MINING TECHNIQUES .....</b>	<b>5</b>
<b>2.1 Memory-based Single Machine Techniques.....</b>	<b>9</b>
<b>2.1.1 Apriori Approach .....</b>	<b>9</b>
<b>2.1.2 Pattern Growth Approach.....</b>	<b>15</b>
<b>2.2 Disk-based Techniques .....</b>	<b>19</b>
<b>2.2.1 Partition-based Approach.....</b>	<b>20</b>
<b>2.2.2 Traditional Database Approach .....</b>	<b>22</b>
<b>2.2.3 Parallel and Distributed Approach .....</b>	<b>25</b>
<b>2.3 Distributed In-memory Techniques .....</b>	<b>29</b>
<b>3 OBJECT-ORIENTED APPROACH TO FREQUENT SUBGRAPH MINING</b>	

3.1	Background .....	30
3.2	Related Work .....	31
3.3	An OO-approach to Mine FSGs .....	32
3.3.1	<i>Subgraph Construction and FSG Determination</i> .....	34
3.3.2	<i>Optimization Techniques</i> .....	42
3.3.3	<i>DB-FSG vs OO-FSG Implementation</i> .....	43
3.4	Details of OO-FSG Algorithm .....	44
3.5	Experimental Details .....	47
4	A MAPREDUCE-BASED FREQUENT SUBGRAPH MINING (MRFSM).....	48
4.1	Background .....	49
4.2	Related Work .....	49
4.3	MapReduce Overview .....	50
4.3.1	<i>Map Function</i> .....	51
4.3.2	<i>Reduce Function</i> .....	51
4.4	Frequent Subgraph Mining Using MapReduce .....	52
4.4.1	<i>FSG Determination</i> .....	53
4.5	Subgraph Construction .....	54
4.5.1	<i>Map Function for Gathering Subgraphs with Similar Graph ID</i> .....	54
4.5.2	<i>Reducer for Constructing Subgraphs</i> .....	55
4.5.3	<i>Map Function for Gathering Subgraph Structures</i> .....	55

4.5.4	<i>Reducer for Determining Frequent Subgraphs</i> .....	55
4.6	<b>Details of MapReduce-FSG</b> .....	56
4.6.1	<i>Canonical Ordering of Elements</i> .....	58
4.6.2	<i>Illustrative Example</i> .....	58
4.7	<b>Experimental Details</b> .....	62
4.7.1	<i>Synthetic Datasets</i> .....	62
4.7.2	<i>Biological Datasets</i> .....	64
5	<b>A HIGHLY SCALABLE FREQUENT SUBGRAPH MINING APPROACH USING APACHE SPARK (SPARKFSM)</b> .....	66
5.1	<b>Background</b> .....	67
5.2	<b>Related Work</b> .....	67
5.3	<b>FSM on Undirected Transaction Graphs</b> .....	68
5.4	<b>FSM on Directed Transaction Graphs</b> .....	72
5.5	<b>Experimental Details</b> .....	76
6	<b>CONCLUDING REMARKS</b> .....	80
	<b>BIBLIOGRAPHY</b> .....	83

## LIST OF TABLES

Table 2.1 Composition Relation .....	13
Table 2.2 Vertex table.....	24
Table 2.3 Edge table .....	25
Table 2.4 Example transaction/itemsets .....	26
Table 3.1 Vertex table.....	38
Table 3.2 Edge Table .....	38
Table 3.3 Single-edge Table .....	39
Table 3.4 DB-FSG [84] vs OO-FSG Performance .....	47
Table 4.1 Performance of MapReduce-FSG (time in seconds) .....	63
Table 4.2 Performance on Biological datasets using a support of 50% and clusters of size 2 and 4 (in seconds) .....	65
Table 5.1 SparkFSM [90] performance analysis on biological graphs (time in seconds, threshold frequency: 50%).....	77
Table 5.2 MRFSM [80] performance analysis on biological graphs (time in seconds, threshold frequency: 50%).....	78
Table 5.3 SparkFSM [90] performance analysis on large undirected datasets (time in minutes)	79

## LIST OF FIGURES

Figure 1.1 LinkedIn, Facebook, and Twitter graphs.....	2
Figure 1.2 Program flow and Chemical compound [42] .....	3
Figure 1.3 Internet and Yeast protein network .....	3
Figure 2.1 Undirected and directed labeled graphs .....	6
Figure 2.2 A sample graph dataset [41].....	7
Figure 2.3 Frequent subgraphs (left: support 2, right: support 3).....	8
Figure 2.4 Apriori-based extension.....	9
Figure 2.5 AGM [41] .....	11
Figure 2.6 FSG [41] .....	11
Figure 2.7 Graph and 3 edge-disjoint paths .....	12
Figure 2.8 Example Graphs [33].....	14
Figure 2.9 Canonical adjacency matrices [33].....	14
Figure 2.10 Example graphs G1, G2 and their join [33] .....	14
Figure 2.11 Pattern Growth-based Extension .....	15
Figure 2.12 DFS code [32].....	17
Figure 2.13 Rightmost Expansion [32].....	18
Figure 2.14 Database graphs.....	19
Figure 2.15 Embedding.....	19
Figure 2.16 An ADI structure [37] .....	20
Figure 2.17 PartMiner partition method [35].....	21
Figure 2.18 Data Partition Scheme for PartGraphMining [36].....	22
Figure 2.19 DB-FSG Example Graph [84] .....	24
Figure 3.1 All Major Classes .....	33
Figure 3.2 An Example Subgraph.....	34
Figure 3.3 Representation of graphs in the dataset .....	36
Figure 3.4 Objects of SingleEdge Class .....	37
Figure 3.5 Objects of Subgraph_1 class (satisfying min_sup) .....	40
Figure 3.6 Objects of Single-edge After Pruning .....	41
Figure 3.7 Objects of TwoEdge class (Before pruning) .....	41
Figure 3.8 Objects of Subgraph_2 class (satisfying min_sup) .....	42

Figure 3.9 Comparison with 1% and 3% minimum support .....	48
Figure 3.10 Comparison with 5% and 7% minimum support .....	48
Figure 4.1 A MapReduce Model .....	51
Figure 4.2 Frequent Subgraph mining using MapReduce .....	52
Figure 4.3 Example graphs in the Dataset .....	53
Figure 4.4 Single edge subgraphs that meet support .....	59
Figure 4.5 Double edge subgraphs that meet support.....	61
Figure 4.6 Triple edge subgraphs that meet support (the subgraph strings show on the top) .....	62
Figure 4.7 Comparison with 1%, 4% and 7% Support.....	64
Figure 4.8 Results of Biological datasets. Each graph shows the runtimes for active and inactive outcomes on both clusters of size 2 and 4.....	66
Figure 5.1 Isomorphic Structures.....	68
Figure 5.2 Undirected Graphs.....	69
Figure 5.3 Retained Structures.....	69
Figure 5.4 Pruned subgraphs.....	70
Figure 5.5 a G1, G2                      Figure 5.5 b G1, G2, and G3.....	72
Figure 5.6 a G1, G3      Figure 5.6 b G2, G3 .....	72
Figure 5.7 Directed graphs.....	73
Figure 5.8 Four-edge directed subgraphs.....	74
Figure 5.9 Five-edge directed subgraphs .....	74
Figure 5.10 Box and Whisker plot showing time required to compute each undirected graph size at the varying frequencies .....	79
Figure 5.11 Performance Comparison between Directed and Undirected on Biological Graph Dataset.....	80

**LIST OF ABBREVIATIONS**

FSM: Frequent Subgraph Mining

GPU: Graphic Processing Unit

Transactions: A group of moderate size graphs

## 1 INTRODUCTION

Frequent pattern mining has become one of the major research areas since the appearance of the seminal paper [1] published by Agrawal and Srikant on item sets. The problem was initially defined for market-basket analysis, where given a database consisting of a set of transactions and a user provided frequency threshold, the goal is to find the frequently occurring items in the entire dataset.

The problem is defined as follows:

Let  $D$  be a transaction database consisting of a set of transactions,  $D = \{T_1, T_2, T_3, \dots, T_n\}$ .

Let  $I$  be a set of items,  $I = \{i_1, i_2, i_3, \dots, i_m\}$ .

Each transaction  $T_i$  consists of a subset of items from  $I$ .

Let  $X$  be a subset of  $I$  called an itemset,  $X = \{i_1, i_2, i_3, \dots, i_k\}$ .

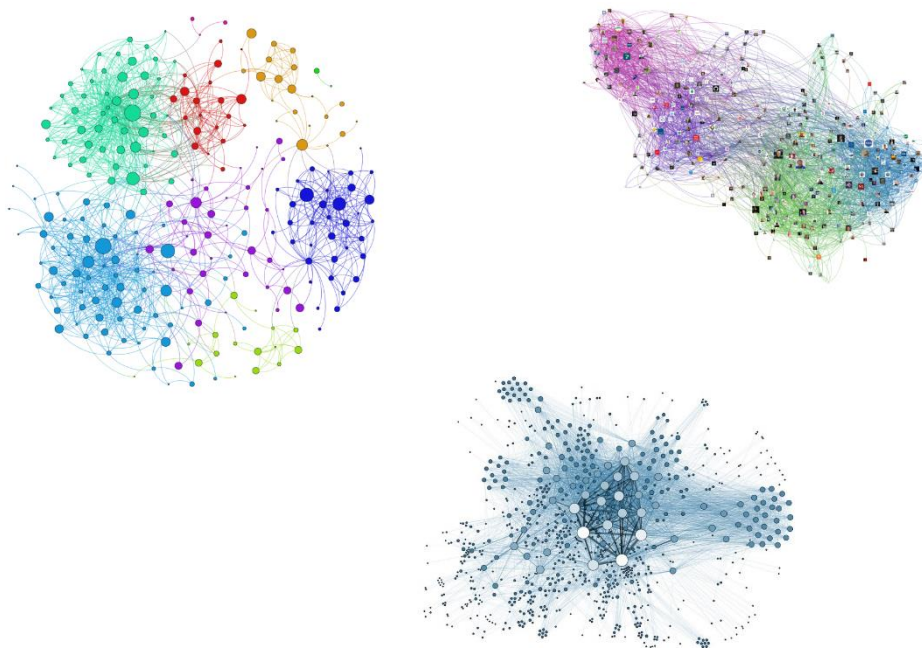
An itemset consisting of  $k$  items is called a  $k$ -itemset.

The Support of an itemset  $X$  is the number of transaction containing  $X$ .

An itemset is called frequent if the support is greater than or equal to the given minimum support determined by the user.

Frequent itemset mining has provided a lot of insight to data mining researchers. Due to the improved computing capability and storage availability, vast amount of data are generated from many different applications. In order to model the generated complicated structures, graphs are considered as the most useful format. Graphs are prevalent in many domains such as protein-protein interaction network in biological networks, chemical compound structures, semi structured XML data, web data, RDF (semantic web), wired/wireless interconnection networks, program traces from software engineering [40]. Graphs are chosen as a common structure in all these domains as modelling complicated structures via graphs are easy. Mining these graphs to

extract knowledge has become the real challenge. Graphs are everywhere. As the social networking sites like Facebook, Twitter, and LinkedIn see member growth, so as the graphs become massive day by day. Figure 1.1 shows the social network graphs, figure 1.2 shows a caller/callee program flow graph and a chemical compound structure, and figure 1.3 shows a yeast protein interaction network and internet network graph. The area of frequent pattern mining from graphs is divided into two categories, one category belongs to a dataset consisting of moderate size graphs, and the second category belongs to single graphs where the dataset contains a single large graph. In the single graph setting (second approach), the purpose is to find the embedding which could be edge-disjoint or share edges (having at least one edge different) with another in the entire graph. There are several solutions proposed for single graph mining in either sequential [2, 3, 4, 25, 26, 27, 28] or parallel computing [5, 6, 7] areas. Our focus is on the first category where the exact counting is done to find the frequent subgraphs on the dataset containing a set of graphs [29, 30, 31, 32, 33, 34].



*Figure 1.1 LinkedIn, Facebook, and Twitter graphs*

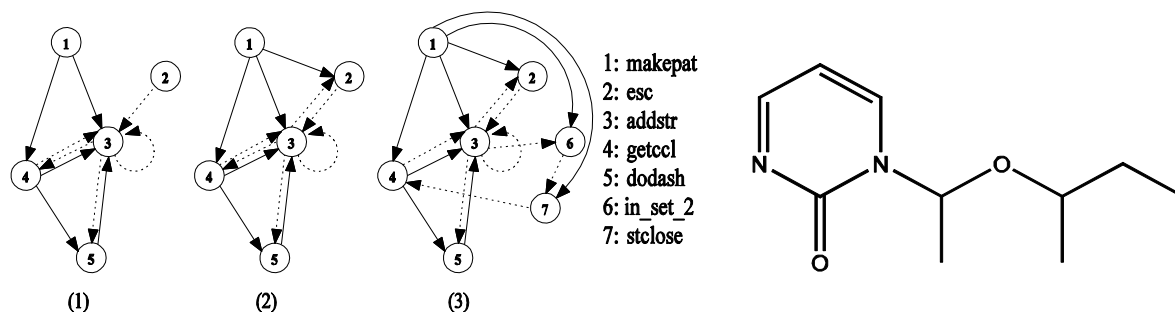


Figure 1.2 Program flow and Chemical compound [42]

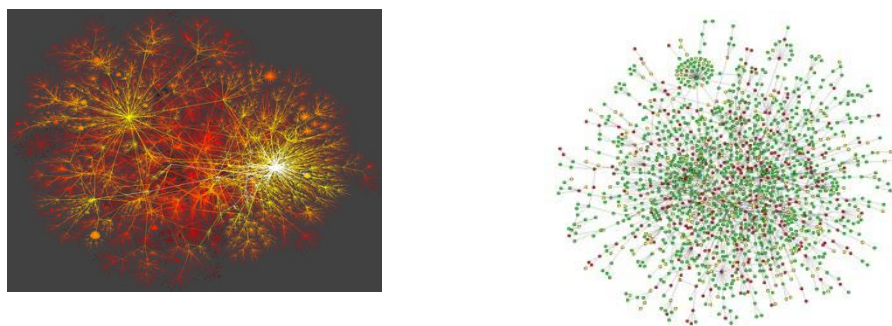


Figure 1.3 Internet and Yeast protein network

## 1.1 Problem Statement

While the complex graphs contain a wealth of knowledge, all that information would merely be a sunken treasure without proper and efficient mining techniques to extract meaning out of these complex structures. Therefore, the technique must be such that not only it produces accurate result, but also does so in a reasonable amount of time. However, with the exponential growth of the size of dataset, this task is becoming increasingly challenging. The problem is defined as follows: given a dataset ( $D$ ) consisting of a set of graphs  $G_1, G_2, G_3, G_4 \dots G_n$ , and a minimum support threshold  $\text{min\_sup}$ , the goal is to find all frequent hidden substructures ( $g$ ). A subgraph ( $g$ ) is frequent if its support is no less than the minimum threshold level. The minimum

support is provided by the user as a percentage. Basically, support of a subgraph is the number of graphs that contain the subgraph. When we discuss about graphs, the graph isomorphism and subgraph isomorphism are the major aspects that needs to be discussed which is known to be an NP-complete [24] problem.

## 1.2 Motivation

Rapid improvement in automated data collection tools have made it possible to generate and collect massive data. Large amount of data is generated from areas such as bioinformatics, cheminformatics, social networks, semantic web, computer vision, etc. Graph pattern mining is an established area of research and we have abundant graph data to mine knowledge from. Knowledge extracted from these data can then be used to develop or model various applications. In software engineering area, bugs in programs can be identified through differential analysis of classification accuracy in program flow graphs [8]. In bioinformatics domain, frequently occurring patterns are introduced as functional building blocks in transcriptional regulatory networks [9, 10]. In the field of cheminformatics, the frequent patterns could potentially help to study the molecules for new drug discovery and chemical synthesis success prediction where the purpose is to find molecular features that inhibit a specific reaction [11]. In social networks finding the frequent patterns can help in understanding the social behavior and relationship among groups. There are many main memory-based approaches which assume data to be contained entirely in memory and computation is done at the same time. As the data grows exponentially, we cannot rely solely on memory-based methods. Memory becomes a bottleneck as the entire data cannot fit in memory. To solve this problem, we proposed to use disk-based approaches which help in large-scale data processing. During our experiments, we found the disk

I/O and non-iterative style of computing of OO-FSG [50] and MRFSM [80] were the major drawbacks and this provided us insight to apply the distributed in-memory Spark engine.

### 1.3 Our Contribution

The following are our contributions:

- (a) We have provided an extensive survey on FSM.
- (b) Since our research is on the same line, we have conducted several different experiments on real life datasets, and
- (c) Provided performance comparisons between them using different types of high-performance computing methods.

We categorize our research into two types, the first category [50, 80] is disk-based where we used the object-oriented database db4o [12] and the Hadoop's MapReduce model [13, 51]. The second category [90] is highly distributed but in-memory processing, for which we used Apache Spark engine. All our approaches are based on the industry standards during the time of publication of the work.

## 2 REVIEW OF FREQUENT SUBGRAPH MINING TECHNIQUES

In this chapter we present various existing frequent subgraph mining techniques. We begin our discussion by providing some notations and definitions used throughout the text.

**Definition 1 (Graph)** A graph is defined as an ordered pair  $G = (V, E)$ .

$V$  is a set of vertices (nodes)

$E \subseteq V \times V$  is a set of edges (links)

**Definition 2 (Labeled Graph)** A labeled graph is represented by four tuples  $G = (V, E, L, I)$ ,

where

$V$  is a set of vertices (nodes)

$E \subseteq V \times V$  is a set of edges, where edges can be directed or undirected

$L$  is a set of labels

$l: V \cup E \rightarrow L$ ,  $l$  is a function assigning labels to the vertices and the edges.

Examples of labeled directed and undirected graphs are shown in figure 2.1. A, B, C, D are the node labels, and a, b, c, d, e are the edge labels.

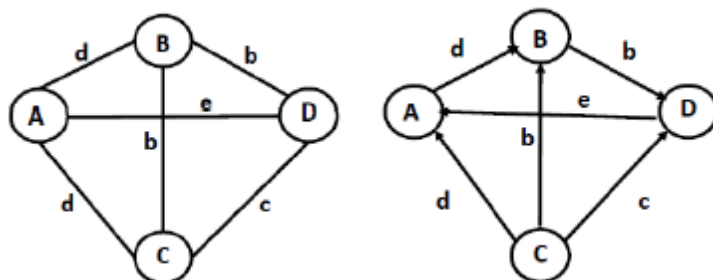


Figure 2.1 Undirected and directed labeled graphs

We discuss directed and undirected type of transaction graphs and performance analysis comparison on both the categories. The nature of directed graphs varies from undirected, for example, airline flight information graphs are directed, and it has a source and a destination, but the chemical compound structures are undirected. Since atoms share bonds with each other, direction has no meaning for chemical compounds. Our approach to handle isomorphism varies due to the different nature of the two categories. These will be explained in detail while covering each approach.

**Definition 3 (Subgraph)** Given a graph  $G (V, E)$ , a graph  $g (V_g, E_g)$  is a subgraph of  $G$  if  $V_g \subseteq V$  and  $E_g \subseteq E$ .

**Definition 4 (Induced Subgraph)** Given a graph  $G (V, E)$ , a graph  $g (V_g, E_g)$  is an induced subgraph of  $G$  if  $V_g \subseteq V$  and  $E_g$  contains all the edges of  $E$  that connect vertices in  $V_g$ .

**Definition 5 (Isomorphism)** Two graphs  $G_a = (V_a, E_a)$  and  $G_b = (V_b, E_b)$  are isomorphic if they are topologically identical to each other. In other words, there is a mapping from  $V_a$  to  $V_b$  and each edge of  $E_a$  is mapped to an edge of  $E_b$  and vice versa.

**Definition 6 (Automorphism)** Two graphs  $G_a = (V_a, E_a)$  and  $G_b = (V_b, E_b)$  are said to satisfy the automorphism property if there is an isomorphism mapping where  $G_a = G_b$ .

**Definition 7 (Subgraph Isomorphism)** Given two graphs  $G_a = (V_a, E_a)$  and  $G_b = (V_b, E_b)$ , the problem is to find if  $G_a$  contains a subgraph which is isomorphic to  $G_b$ .

**Definition 8 (Transaction Graph)** A given graph database  $G$  is called a transaction graph database, if it contains a set of moderate sized graphs.

$$G = \{g_1, g_2, g_3, g_4, \dots, g_n\} \text{ where } g_1, g_2 \text{ etc. are individual graphs}$$

**Definition 9 (Frequent Subgraph Structure)** Given a graph database  $D = \{G_1, G_2, G_3, \dots, G_n\}$ , let a subgraph  $g$  is contained in  $|D_g|$  number of graphs. Then support of  $g$  is defined as  $\text{sup}(g) = |D_g|/|D|$ , where  $|D|$  is the total number of graphs in  $D$  and  $|D_g|$  is the number graphs in  $D$  which contain  $g$ . The subgraph  $g$  is said to be frequent if its support is not less than the minimum support threshold provided by the user. The following example figure 2.2 shows a database consisting of 3 chemical compounds which comes under the undirected labeled graph category. If we take support as 2, then we find two subgraphs shown in figure 2.3 as the frequent structures.

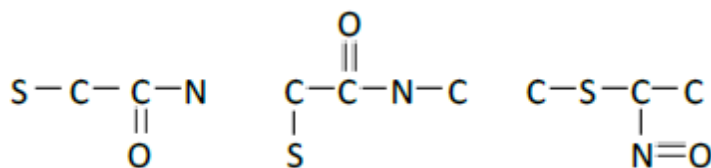
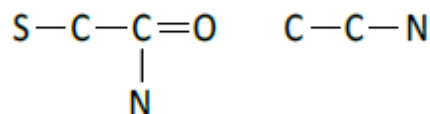


Figure 2.2 A sample graph dataset [41]



*Figure 2.3 Frequent subgraphs (left: support 2, right: support 3)*

Frequent pattern mining became a very popular topic after the invention of several scalable and efficient techniques in the areas of item set mining. To mention a few, the very first association rule mining [1, 14] introduced the area of frequent pattern mining. Subsequently, several item-set mining methods [15, 16, 17, 18, 53], sequential patterns [19, 20, 21], and trees [22, 23, 52] were developed. With the motivation from apriori algorithm [1], Inokuchi et al. [31] proposed AGM which mines the association rules among the frequently occurring subgraphs. Following the apriori model, FSG [91], PATH [26] algorithms were developed. Another group of researchers used a non- apriori-based approach [Mofa, gSpan, FFSM, GASTON] where the subgraphs were extended by adding a single edge each time. With the growing size of databases and availability of larger disk space and cloud-based technologies, some researchers proposed traditional database-based and cloud- based approaches for scalability. The following subsections are grouped into 3 categories. The first category covers memory-based single machine techniques (Apriori-based methods and pattern-growth approaches). The category describes the disk-based techniques (partition-based approach, traditional database approach, and parallel and distributed approach). The third category is the new generation techniques which is based on the highly distributed but computation happens inside memory.

## 2.1 Memory-based Single Machine Techniques

The algorithms developed around early 2000's didn't have much flexibility except running in single machine setting. There are many major algorithms developed around this time. We categorize them into apriori and pattern-growth approaches.

### 2.1.1 Apriori Approach

Most apriori-based approaches follow the breadth-first method of traversal. Figure 2.4 shows the growth pattern of apriori method. P, Q, and R are three  $n$ -edge subgraphs, the apriori algorithm merges two  $n$ -edge subgraphs if they share same  $(n-1)$ -edge core and the resulting  $(n+1)$ -edge subgraphs are  $G_1, G_2, G_3, \dots, G_n$ . The apriori-based frequent subgraph algorithms follow the downward closure property which states that if a graph is frequent then all its subgraphs must be frequent. The "Apriori" algorithm is given in algorithm 2.1 which is adapted from [41].

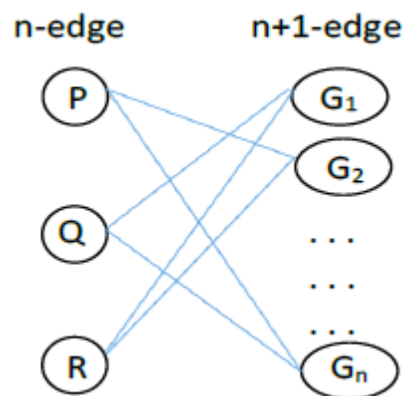


Figure 2.4 Apriori-based extension

**Algorithm 2.1 Apriori***Input: A graph dataset  $G_s$ ,  $min\_sup$* *Output: Frequent subgraphs  $F_k$* 

1. *Populate  $F_1$  by removing all infrequent edges and vertices from  $G_s$*
  2.  *$k = 1$*
  3. *While ( $F_k \neq 0$ )*
  4. *forall frequent  $S_i \in F_k$*
  5. *forall frequent  $S_j \in F_k$*
  6. *forall size( $k+1$ ) subgraph ( $s$ ) generated from merging  $S_i$  and  $S_j$*
  7. *If  $support(s) \geq min\_sup$  and  $s \notin F_{k+1}$*
  8. *add  $s$  to  $F_{k+1}$*
  9.  *$k = k+1$*
  10. *return*
- 

The above algorithm works as follows: in the beginning, all the infrequent edges and vertices are removed from the database. In each iteration, the frequent subgraphs of size  $k$  are merged which have common size  $(k-1)$  cores. The generated size  $(k)$  structure is checked for frequency and added to the frequent subgraph set. Those that do not comply with the frequency are pruned from the input dataset. The algorithm terminates when there are no more newly formed subgraphs.

We will discuss four very well-known apriori-based algorithms AGM [30], FSG [91], PATH [26] and FFSM [33]. AGM [30] takes a vertex-oriented approach, in each iteration of the above apriori algorithm, AGM adds a new node. The newly formed

structure of size  $(k+1)$  contains the core which is  $(k-1)$  vertices and two new vertices from the merged structures. Figure 2.5 shows the candidate generation of AGM.

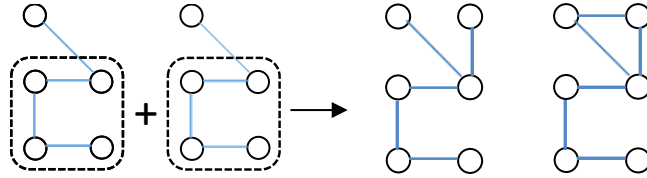


Figure 2.5 AGM [41]

Kuramochi et al. [3, 91] developed the frequent subgraph mining algorithm “FSG” in which they took an edge-based approach where the size of the subgraph represents the number of edges it contains. They followed the same approach as shown in the “Apriori” algorithm. In FSG, a new size  $(k+1)$  structure is formed by merging two size- $k$  structures which share a common core. Here core means both the subgraphs have same size  $(k-1)$  edges. The newly formed subgraph contains the core size  $(k-1)$  and two new edges from the merged subgraphs. Figure 2.6 illustrates the candidate generated when two subgraphs with common cores are merged.

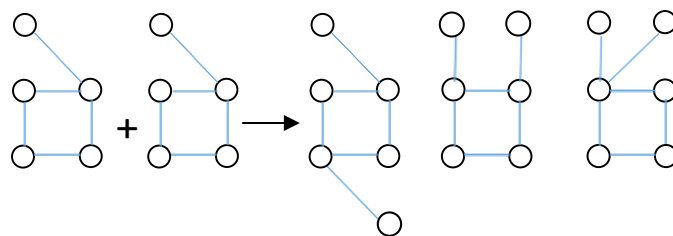


Figure 2.6 FSG [41]

Vanetik et al. [26] proposed a path approach in which candidate generation follows Apriori strategy where the building blocks are edge-disjoint paths. Two paths of length

(k) are joined if they share the same core. The following figure 2.7 shows three paths of graph G to the right.

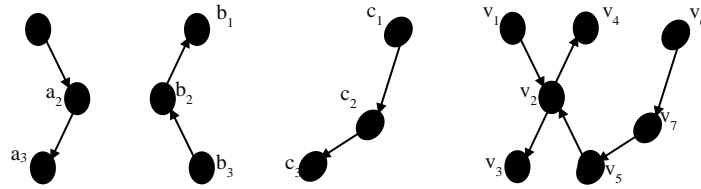


Figure 2.7 Graph and 3 edge-disjoint paths

The pseudocode of PATH [26] is given in algorithm 2.2. Initially, all frequent single edge paths are found. Size-2 edge-disjoint paths are constructed from size-1 edges, they proposed a table structure which stores paths as columns and the vertices as the rows. A few paths together build a composition relation.

**Algorithm 2.2 PATH**

- Find all frequent single edge paths.
- Construct  $k+1$ -th candidate path by joining two  $k$ -th candidates which share the same core.
- Evaluate the frequency of the newly formed path and add that to the candidate set if that satisfy the support threshold.
- Repeat the process until there is no new frequent paths.

An example of composition relation for figure 2.7 is given in table 2.1. Two composition relations are joined if they have  $(n-1)$  paths in common.

*Table 2.1 Composition Relation*

<b>Node</b>	<b>P1</b>	<b>P2</b>	<b>P3</b>
<b>V1</b>	<b>a1</b>	<b>0</b>	<b>0</b>
<b>V2</b>	<b>a2</b>	<b>b2</b>	<b>0</b>
<b>V3</b>	<b>a3</b>	<b>0</b>	<b>0</b>
<b>V4</b>	<b>0</b>	<b>b1</b>	<b>0</b>
<b>V5</b>	<b>0</b>	<b>b3</b>	<b>c3</b>
<b>V6</b>	<b>0</b>	<b>0</b>	<b>c1</b>
<b>V7</b>	<b>0</b>	<b>0</b>	<b>c2</b>

The subgraph extension is described in two different ways. The first approach is a bijective sum on two composition relations having  $k$  paths where both share  $k-1$  paths. The other method is splice method, which is defined as a merger of two nodes belonging to two different paths in a graph into a single node. Let  $C_1$  and  $C_2$  be two composition relations. A splice of two composition relations  $C_1(P_1, P_2, P_3, \dots, P_n)$  and  $C_2(P_i, P_j)$ ,  $1 \leq i, j \leq n$ , is a composition relation that turns every node common to  $P_i$  and  $P_j$  in  $C_2$ , into the node common to  $P_i$  and  $P_j$  in  $C_1$  as well.

Huan et al. proposed [33] a novel data structure called Canonical Adjacency Matrix (CAM) to store the graph. The rows and columns in a CAM represent the vertices in the graph. The diagonal entries represent the node labels, all other entries are the edge entries. Figure 2. 8 represents two graphs and figure 2. 9 represents their canonical adjacency matrices.

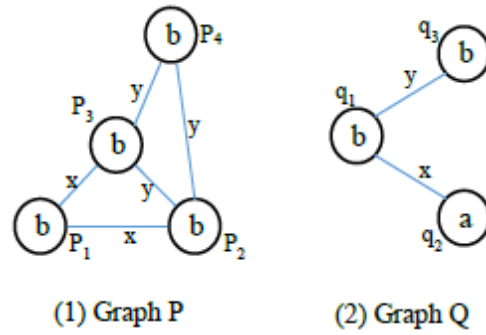


Figure 2.8 Example Graphs [33]

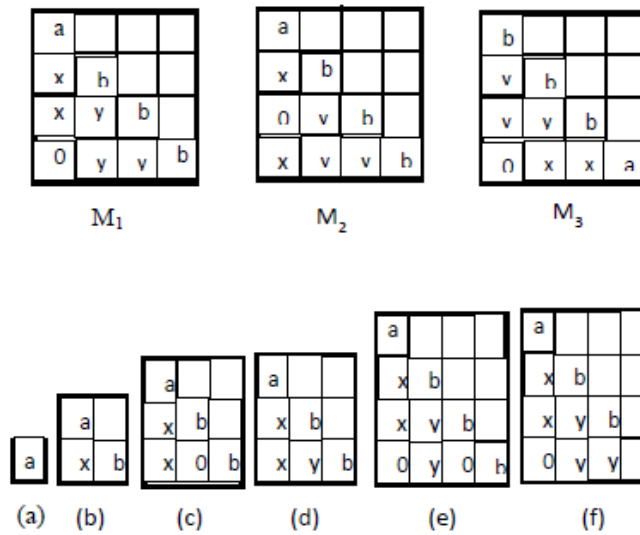


Figure 2.9 Canonical adjacency matrices [33]

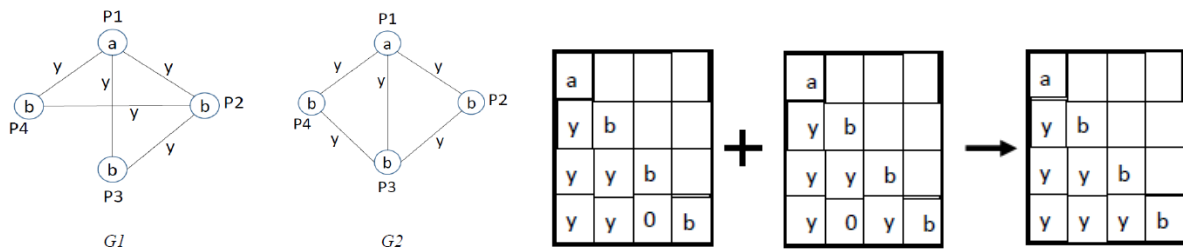


Figure 2.10 Example graphs  $G_1$ ,  $G_2$  and their join [33]

The paper has discussed several cases for joining and extension. Here, we show one case. Figure 2.10 shows joining of two CAMs (corresponding to graphs  $G_1, G_2$ ) both of size  $m \times m$ , all the edge entries are same except the last edge. The resultant matrix shown to right of figure 2.10 is also of size  $m \times m$ . FFSM [33] defines a canonical code for the adjacency matrix as the sequence formed by concatenating lower triangular entries of the matrix. If the matrix  $M$  is of  $m \times m$  size, then the sequence of lower triangular entries will constitute  $m_{1,1}m_{2,1}m_{2,2}\dots m_{n,1}m_{n,2}\dots m_{n,n-1}m_{n,n}$  where  $m_{i,j}$  is the entry of the  $i$ th row and  $j$ th column in  $M$  assuming the rows and columns are numbered 1 through  $n$ .

### 2.1.2 Pattern Growth Approach

We broadly categorized all non-apriori based algorithms as pattern growth-based approach. The general idea in these algorithms are to add an additional edge to the existing frequent subgraph. The newly added edge may or may not add a new vertex.

Figure 2.11 shows the pattern growth graph.

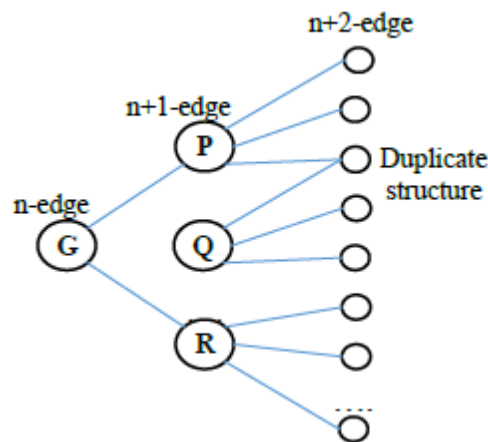


Figure 2.11 Pattern Growth-based Extension

In this category, there are quite a few efficient algorithms, which are nearly comparable to each other w.r.t. efficiency. We will discuss three significant algorithms [32, 34, and 11]. In pattern growth algorithms, the subgraph extension can be both breadth-first and depth-first, whereas the DFS approach is best suited for better memory usage. Algorithm 2.3 gives a general idea of pattern growth approach adapted from Han and Kamber book.

**Algorithm 2.3 Pattern\_Growth( $s, GDB, min\_sup, G$ )**

*Input: A frequent subgraph  $s$ , graph dataset  $GDB$ , Minimum*

*Support ( $min\_sup$ )*

*Output: A frequent subgraph set  $G$*

1. *if  $s \in G$  then return*
  2. *else add  $s$  to  $G$*
  3. *scan  $GDB$  once to find all edges  $e$  where  $s$  can be extended to  $s \langle \rangle e$*
  4. *forall frequent  $s \langle \rangle e$*
  5. *call  $Pattern\_Growth(s \langle \rangle e, GDB, G)$*
  6. *return*
- 

The first algorithm in this category is known as MoFa [11], in which the candidate generation happens by adding a new edge. Extension is restricted to the fragments that actually appear in the database. Embedding are stored for faster support calculation.

Second algorithm in this category is popularly known as “gSpan” [32, 42]. The authors have proposed a DFS lexicographic ordering and minimum DFS code to support DFS search. The figure 2.12 shows three graphs b, c, and d isomorphic to a, but only one of them have the potential to grow.

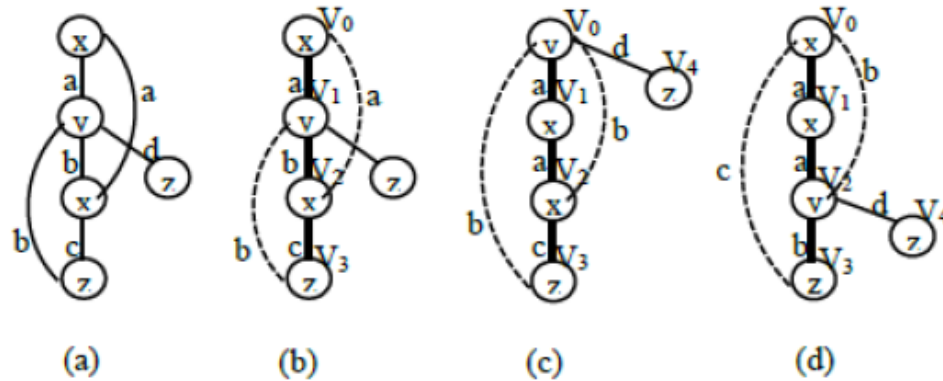


Figure 2.12 DFS code [32]

Given the DFS codes for different DFS trees gSpan algorithm chooses the minimum code. From figure 2.12, following the minimum DFS code rule,  $a < b < c$ . In order to eliminate duplicate generation, gSpan approach adapts a similar methodology like TreeMinerV's equivalence class extension [23] and FREQT's rightmost expansion [22] in frequent tree discovery. Rightmost expansion for the candidates follows a preorder of tree traversal and restricts the expansion to only the nodes in the rightmost path for forward edges and rightmost vertex for the back edges. Forward edges are the edges which add a new vertex to the DFS tree. Back edges only add an edge which connects the rightmost vertex to an existing vertex in the rightmost path. Back edges are not included in the DFS tree [45]. Figure 2.13 shows the rightmost expansion of graphs.

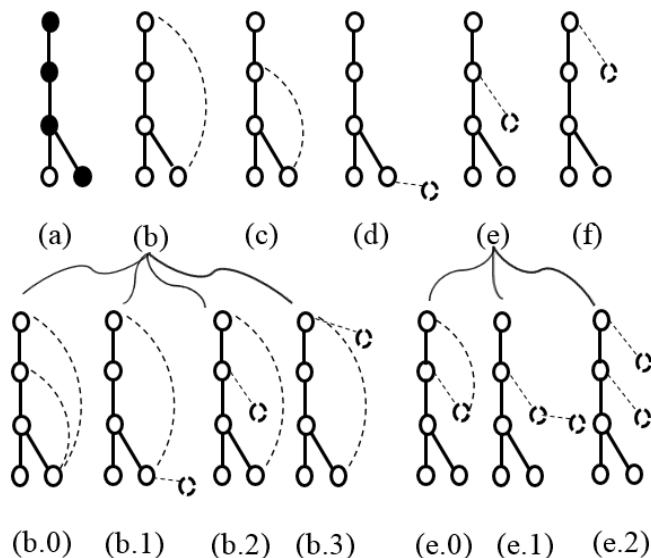


Figure 2.13 Rightmost Expansion [32]

The last algorithm in this category is GASTON [34]. They define a partial order consisting of paths, free trees and cyclic graphs. Path is on top of the partial order in which two nodes have degree 1, while all other nodes have degree 2. A graph without cycles is considered as a free tree. A free tree becomes a cyclic graph when an edge is added between two existing nodes. They propose an efficient data structure to store the embedding of a structure and its ancestors in the partial order. The embedding list consists of all occurrences of a particular label in the database. The embedding tuple consists of (1) a pointer to an embedding tuple of the parent structure, (2) the identifier graph in the graph database and (3) a node in that graph. Figure 2.14 shows two example graphs in the database and figure 2.15 shows the embedding of the ancestors. Individual row in the embedding lists table denotes the embedding list of an ancestor of the database graphs shown in figure 2.15.

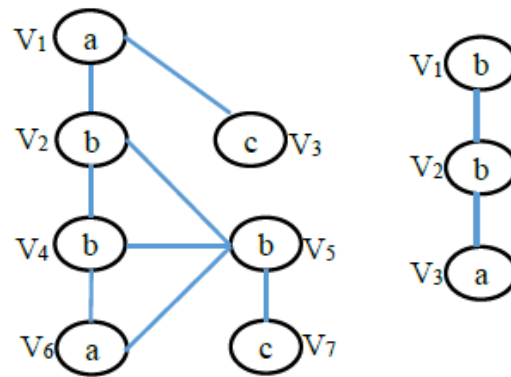


Figure 2.14 Database graphs

	1	2	3	4	5	6	7	8
1	$(\lambda, G_1, v_2)$	$(\lambda, G_1, v_4)$	$(\lambda, G_1, v_5)$	$(\lambda, G_2, v_1)$	$(\lambda, G_2, v_2)$			
2	$(1, G_1, v_4)$	$(1, G_1, v_5)$	$(2, G_1, v_2)$	$(2, G_1, v_5)$	$(3, G_1, v_2)$	$(3, G_1, v_4)$	$(4, G_2, v_2)$	$(5, G_2, v_1)$
3	$(1, G_1, v_6)$	$(3, G_1, v_1)$	$(5, G_1, v_1)$	$(6, G_1, v_6)$	$(7, G_2, v_3)$			
4	$(1, G_1, v_1)$	$(2, G_1, v_6)$						
5	$(1, G_1, v_5)$	$(2, G_1, v_5)$						
6	1	<u>2</u>						
7	<u>2</u>							

Figure 2.15 Embedding

## 2.2 Disk-based Techniques

The major drawback of memory-based technique is that data must be small to fit into main memory. We have reached a time where we have plenty of data available, but we cannot process all of them at one time in main memory. We categorized the disk-based approaches into three categories. The first category belongs to disk-based approach where the data is partitioned such that the chunks will fit in memory, after which the memory-based algorithms are applied on the chunks to find frequent patterns. The second category belongs to the traditional database-based approach where the entire data is stored in databases such as relational databases (DB2, Oracle,

MySQL) and object-oriented databases like db4o [12]. The third approach consists of parallelizing the data mining process. In summary, the idea is to partition the data between the worker nodes and find the frequent subgraphs at each node.

### 2.2.1 Partition-based Approach

A horizontal data partitioning approach on transaction databases was first introduced by Savasere et al [38]. Wang et al. proposed a partition-based approach, ADI-Mine [37], in which they created an index structure ADI (adjacency index). For each edge, they maintain the graph ids in a linked list. A graph id is entered once per edge irrespective of multiple occurrence of same edge. Figure 2.16 shows the example of the graph and its adjacency index. They adapted the famous gSpan [32] algorithm methodology for frequent subgraph-mining.

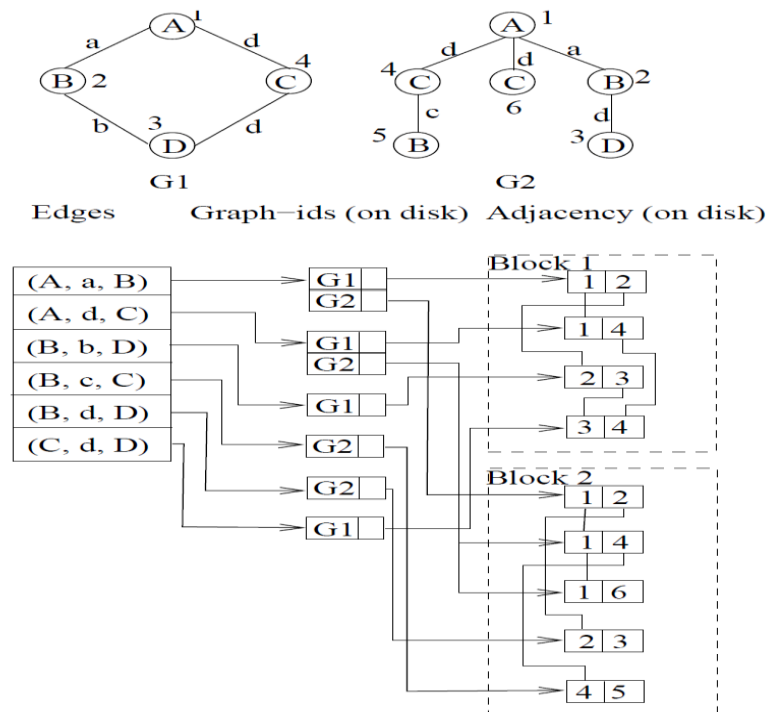


Figure 2.16 An ADI structure [37]

In [35], Wang et al. proposed a partitioning algorithm called PartMiner, which takes the transaction database, the number of partitions  $k$  and minimum support as input. PartMiner works in two phases - in the first phase, the database is divided into  $k$  subunits such that each unit data fits in memory, the memory-based algorithm GASTON [34] is called on all subunits. The minimum support threshold used in their approach is the fraction of user provided support divided by  $k$ . After local mining is complete, a merge-join procedure is called to combine the results. Figure 2.17 shows the phase1 and phase2 of their procedure.

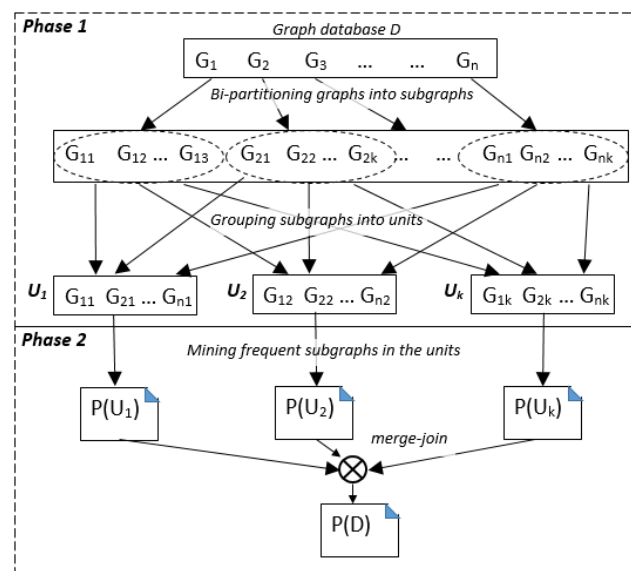


Figure 2.17 PartMiner partition method [35]

Nguyen and Orlowska [36] proposed to use data partition technique on graphs that are an extension of their previous work, which was applied on frequent item sets [39]. In their work [36], K-means algorithm is used to partition the data. Figure 2.18 shows the general idea behind their partitioning approach. Their algorithm is given below in algorithm 2.4.

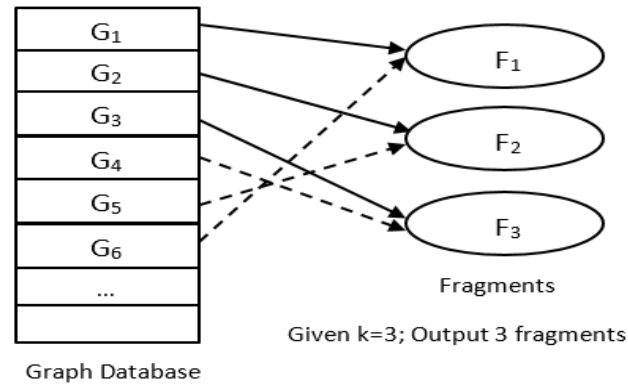


Figure 2.18 Data Partition Scheme for PartGraphMining [36]

### **Algorithm 2.4 PartGraphMining**

*Input: Graph database GDB, Minimum support, Number of partitions ( $k$ )*

*Output: Frequent subgraph set*

1. *Partition the graph database into  $k$  fragments ( $G_1, G_2, G_3 \dots G_n$ ) such that every fragment can be loaded into memory*
2. *Call GASTON or gSpan on each fragment and find the locally frequent subgraphs  $f(G_i)$  where  $i = 1, 2, 3, \dots, k$*
3. *Compute the union of all  $f(G_i)$ , add them to  $LG$*
4. *Compute the intersection for all Globally frequent sets, add to  $G^G$*
5. *Scan the database again to verify if  $(L^G - G^G)$  is frequent or not, output all frequent subgraphs*

### **2.2.2 Traditional Database Approach**

Traditional databases such as relational databases and object-oriented databases became the second choice for large data storages. DB-subdue [47] is the very first attempt using relational database approach for subgraph mining. DB-subdue implements the idea of

SUBDUE [46], which is one of the early frequent subgraph mining algorithms on single graph that detects the best structure using minimum description length principle [48]. The minimum description length principle states that the best theory to describe a set of data is a theory which minimizes the description length of the whole data set. DB- subdue [47] stores graphs as relations in database. Evaluation of best structures are done by counting the frequency of the instances of the substructure within the single graph. It uses standard SQL where subgraph expansion is done by the join operation and counting is performed by the group by operation. Enhanced DB-Subdue [83] and HDB-Subdue [49] is an improvement over DB-Subdue. They handle cycles in graph and multiple edges between vertices. HDB-Subdue allow unconstrained expansion of substructures. The drawback of unconstrained expansion is that it generates duplicates as the same structure is generated from instances in different order. HDB-Subdue keeps track of the duplicates and eliminates them by maintaining an order of vertex numbers and connectivity map. Frequency counting is done by arranging the vertex labels and their connectivity maps. All the above traditional database approaches are based on SUBDUE [46] idea. These implementations surely provided some ideas to apply on transaction graphs.

DB-FSG [84] is the first relational database-based approach which implements frequent subgraph-mining algorithm on a set of transaction graphs. Graphs are represented in relational databases as relations. All the vertices and edges of the individual graphs are stored in the vertex and edge table maintaining their graph id as the identifier. Initially vertex and edge table are constructed with corresponding vertex/edge labels, numbers assigned to them and the graph id that contains them. Figure 2.19 shows

the example graph based on which table 2 is constructed. Table 2.2 shows the vertices, their labels and graph id. Table 2.3 contains the edges, their labels and graph id.

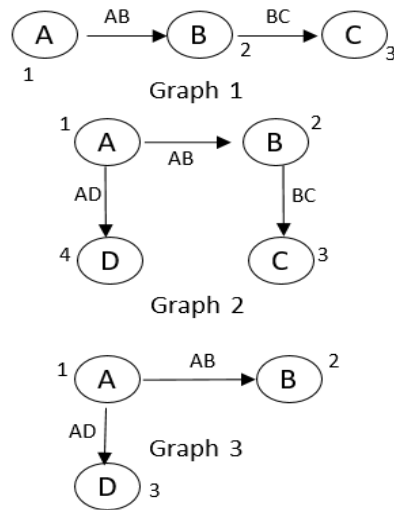


Figure 2.19 DB-FSG Example Graph [84]

Table 2.2 Vertex table

Vertex No.	Vertex Name	Graph ID
1	A	1
2	B	1
3	C	1
1	A	2
2	B	2
3	C	2
4	D	2
1	A	3
2	B	3
3	D	3

Once the vertex and edge tables are formed, an edge table is created by joining both vertex and edge tables at the matching vertex numbers and keeping the graph id the same. Two-edge substructures are formed by joining single edges with itself. Similarly, size-k subgraphs are generated by joining size (k-1) subgraphs with single-edge subgraphs.

*Table 2.3 Edge table*

Vertex 1	Vertex 2	Edge Label	Graph ID
1	2	AB	1
1	3	BC	1
1	2	AB	2
1	4	AD	2
2	3	BC	2
1	2	AB	3
1	3	AD	3

Since the expansion is unconstrained, a particular substructure could be generated multiple times from two different instances joined in different manner. Hence, duplicates are handled carefully. As multiple edges and cycles are considered, DB-FSG [84] imposes that the edge new that is added should not have same edge number as in the instance edges. Frequency counting is done based the node label, edge label, graph id and the connectivity map. DB-FSG encouraged us to implement frequent subgraph mining on object-oriented databases (db4o). Our method [50] is discussed in subsequent section.

### ***2.2.3 Parallel and Distributed Approach***

With the advancement of multi-core technologies, graphic processing units (GPUs) and Google's MapReduce model [51], many researchers tried to apply the parallel and distributed approach to data mining. There are quite a few parallel computing-based approaches in the area of frequent itemset mining. Li et al. [55] used bitmap to represent the itemsets. Each item is represented as '0' or '1' based on the appearance in the

transaction set. To explain it briefly let us consider the table 2.4. Item ‘a’ is represented as {11000} which means ‘a’ appears in transactions T1 and T2. In [54], the items are organized in a tri-based structure which is basically the prefix tree. Li [56] presented an inverse tree structure with bitmap representation to find frequent maximal itemset over stream data.

*Table 2.4 Example transaction/itemsets*

<b>Tid</b>	<b>Itemsets</b>
1	a b c d e
2	a b c d
3	b c d
4	b e
5	c d e

A novel data structure is introduced by Amossen and Pagh [57] called BATMAP, which provides all advantages of bitmap along with space compression for sparse data sets using hash tables. Teodoro et al. [58] use tree-projection based structure. Instead of bitmaps, the authors have proposed to store the transactions in a vector. Cheung et al. proposed FDM [59] to mine association rules using distributed approach. They find locally frequent items on each machine and broadcasts them to all machines. Both local and global pruning are applied to have lesser number of candidates at individual sites. Li et al. [60] proposed a parallel version of FP-Growth [17], a memory-based algorithm on multi-core system. They propose a cache- conscious frequent pattern array and a lock-free dataset tiling parallelization mechanism. A MapReduce based parallel FP- Growth is proposed in [61]. In their approach, data is partitioned, and each machine performs the mining task

independently. This way they reduce the communication cost between machines. Instead of depending on user support, they find top-k frequent patterns. Miliaraki et al. proposed MG-FSM [62] a sequence pattern mining using MapReduce. Their partitioning approach is based on the concept of “projected database”.

After the development of many memory-based algorithms in the area of frequent subgraph mining, the focus is on parallelizing the algorithms to increase the efficiency and handle large-scale graph data. Wu et al. [7] implemented a parallel subgraph mining algorithm using MapReduce framework where motif network diameter and degrees of vertices are taken as standard for motif matching. Liu et al. [63] proposed a MapReduce-based pattern-finding algorithm MRPF for network motif detection from complex networks. Reinhardt and Karypis proposed [6] an algorithm using OpenMP that finds connected edge-disjoint embedding. Wang et al. [66] presented parallel algorithm for their previously developed Motif Miner Toolkit [68] that mines structural motifs in a wide range of bio-molecular datasets. SUBDUE [28] system has been improved a lot since it was developed. The parallel version [65] applies three partitioning schemes such as functional parallel approach (FP-SUBDUE), dynamic partitioning (DP-SUBDUE) and static partitioning (SP-SUBDUE). FP-SUBDUE divides the search for candidates among processors, a second functional parallel approach called dynamic partitioning (DP-SUBDUE) in which each processor evaluates a disjoint set of the input data, and SP-SUBDUE uses a static data partitioning approach. Meinel et al. [67] parallelized the memory-based algorithm MoFa [11] with a substantial speed-up gain. Kang et al. [5] presents “PEGASUS”, an open source graph mining library built using MapReduce framework on Hadoop platform. PEGASUS handles typical mining tasks such as

connected component [71, 72, 73], diameter of the graph [70], and computing the radius of node. Zhao et al. [69] proposed “SAHAD” a MapReduce-based algorithm, which is in fact a Hadoop version of the color-coding algorithm [74, 75]. Afrati et al. [76] proposed a MapReduce-based approach for finding all instances of a given sample graph in a larger graph. They use the techniques from their paper [77] for computing multiway joins to reduce communication cost. Xiang et al. [78] present a MapReduce-based scalable and fault-tolerant solution for the maximum clique problem. They use a graph coloring-based partitioning approach which recursively partition the data into smaller units while maintaining load balance. The maximum cliques of different partitions are computed independently.

Fatta et al. [44] use a search tree partitioning strategy, along with dynamic load balancing and a peer-to-peer communication framework for efficient mining. Luo et al. [79] proposed a MapReduce-based subgraph query search method. The idea is: given a subgraph find all graphs containing that particular query graph. Buehrer et al. [64] proposed parallelizing FSG algorithms on CMP architecture. We proposed [80] a MapReduce-based FSG which is covered in chapter 4. A few more works are published following our implementation on MapReduce. Aridhi et al. [43] proposed a density-based data partitioning approach on MapReduce framework. Bhuiyan and Hasan [81] proposed MIRAGE, a MapReduce-based approach in which they have adopted idea from gSpan [32] for right-most extension to prevent duplicate generation and a gSpan style dfs code for counting and isomorphism checks. Lin et al. [82] makes use of a memory-based algorithm GASTON [34] for their mining task. Data is partitioned between the machines and GASTON is applied to find locally frequent substructures. Then they perform a final

scan to find all globally frequent subgraphs. Next two chapters will describe our disk-based methods towards frequent subgraph mining in transaction databases.

### 2.3 Distributed In-memory Techniques

MapReduce model had a few drawbacks like disk I/O, and especially due to the iterative style requirement for subgraph mining, it proved to be inefficient. Spark evolved based on the shortcomings of MapReduce model (though MR model is still one of the best models for huge batch processing). Over the past years, Spark [87] has become the major industry standard for its in-memory processing of big data. As per our knowledge and findings, there are not many publications utilizing the power of Spark. Authors in [88] have used Spark to find the frequent subgraphs from single large graphs, which is not the major focus of the paper. In this study, our focus is on the transactional setting. Authors in DIMSpan [85] have used Apache Flink, which is similar to Spark but mostly used for real-time processing. In their paper, their focus is on directed multi-graphs. To the best of our knowledge for the first time, we have introduced the ability of Spark engine on undirected transactional graphs. Leveraging the same utility, we could see tremendous improvement on our previous MapReduce-based [80] approach on directed graphs. Algorithm 2.5 describes DIM Span’s distributed dataflow.

#### Algorithm 2.5 Distributed FSM Dataflow

**Require:**  $\mathcal{G} = \{\langle G, \mu^1 \rangle_i\}_{i \in \mathbb{N}}, f_{min}$

- 1:  $\mathcal{F} \leftarrow \emptyset$
- 2:  $\mathcal{F}^k \leftarrow \emptyset$
- 3: **repeat**
- 4:    $\mathcal{P}^k \leftarrow \mathcal{G}.\text{flatmap}(\text{report})$
- 5:    $\phi_w^k \leftarrow \mathcal{P}^k.\text{combine}(\text{count})$
- 6:    $\phi^k \leftarrow \phi_w^k.\text{reduce}(\text{sum})$
- 7:    $\mathcal{F}^k = \mathcal{P}^k.\text{filter}(\phi^k(P) \geq f_{min})$
- 8:   **broadcast**( $\mathcal{F}^k$ )
- 9:    $\mathcal{G} \leftarrow \mathcal{G}.\text{map}(\text{patternGrowth})$
- 10:    $\mathcal{G} \leftarrow \mathcal{G}.\text{filter}(\exists P : |\mu^{k+1}(G, P)| > 0)$
- 11:    $\mathcal{F} \leftarrow \mathcal{F} \cup \mathcal{F}^k$
- 12: **until**  $\mathcal{F}^k \neq \emptyset$
- 13: **return**  $\mathcal{F}$

### **3 OBJECT-ORIENTED APPROACH TO FREQUENT SUBGRAPH MINING (OO-FSG)**

This chapter covers our object-oriented database approach towards FSG. We chose the db4o [12], an open-source object database for java and .NET applications. The interesting aspect of db4o is that the user does not need to create a separate data model, the applications class model defines the structure of the data in db4o database. Db4o database provides persistence to objects automatically. Object persistence is the capability of the system to hold objects even after the system stops running unlike main memory applications which dies when the program stops. There are a few other options exist to make persistence objects. Serialization is one among them where the object is converted into a sequence of bits that is written to a file. However, the drawback of serialization is you need to retrieve the entire information to a file even if all of them are not needed. That's the reason databases are given preference as they are independent of any application that use them.

#### **3.1 Background**

Frequent subgraph mining (FSG) has always been an important issue in data mining. Several frequent subgraph mining have been developed for mining graph data. However, most of these are main memory algorithms in which scalability is a bigger issue. A few algorithms have opted for a relational approach that stores the graph data in relational tables. However, relational databases have their own style of storing data in table format and we need multiple tables to store different aspects of the data. Additionally, multi-valued objects such as collections objects are not easy to map into relational tables. Inheritance, which is a key aspect of object-oriented approach is not supported by relational model. Db4o stores objects directly as is without splitting the components, which is a flexibility to store semantic information.

### 3.2 Related Work

There are a few works which deals with relational database management system (RDBMS). Chakravarthy et al. [47] first introduced the FSG using DB2 database. In this paper, they implemented the idea of SUBDUE [28] on databases. In subsequent efforts, they improved upon [47] and implemented Enhanced DB-Subdue [83] and HDB-Subdue [49]. Both [83] and [49] use oracle DBMS. Our method is largely related to DB-FSG [84]. In their work, they have used relational tables to store graph data and subgraphs. Their approach is briefly as follows: the method has two tables to begin with: one for the vertices and other for the edges which contain individual vertices and edges. The individual tables are joined to obtain size-1 subgraphs. Each time the candidates are generated, the columns in the table grow depending on the size of the graphs. This will eventually place a limit on the size of the maximum substructure that can be detected, as there is a limit on the number of columns a relation can have in a relational database. The algorithm described in DB-FSG can discover substructures of size 165 at the most. After implementing their algorithm, we figured out that it poses many difficulties for larger datasets, and efficiency is the major drawback when the dataset size is large. Another issue related to relational database is storing semantic information. As graph databases, like chemical compounds and protein-protein interactions, have explicit relations between elements, semantic information must be taken care of by the data model.

We implemented the algorithm by Chakravarthy et al. [84] using the same datasets to analyze the pros and cons of both approaches. The algorithm is as follows:

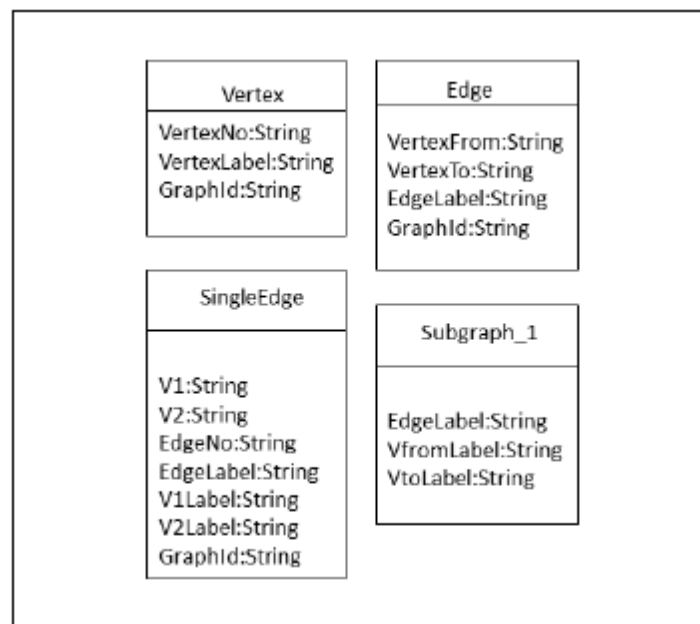
**Algorithm 3.1 DB-FSG [84]**

1. *Create oneedge (instance 1) table by joining vertex table and edge table*
  2. *Remove the edges with instance count less than support from the oneedge table*
  3. *for n=2 to MaxSize do*
    - a. *Join instance (n-1) with oneedge table to generate instance n*
    - b. *Eliminate pseudo duplicates from instance n table*
    - c. *Canonically order instance n table on vertex labels*
    - d. *Project distinct vertex label, edge label and gid to obtain one instance per substructure for each graph and store in dist n table.*
    - e. *Group dist n table by vertex label and edge label to obtain substructures and its count*
    - f. *Retain only the instances of substructure satisfying support and store it in instance n table*
    - g. *If there are no instances of substructure satisfying support then stop*
  4. *End for*
- 

**3.3 An OO-approach to Mine FSGs**

We propose to use db4o to store the graph dataset. The advantage of using db4o as the data storage is because it's highly scalable and do not put burden on memory. To begin with, our approach includes the following basic classes: Vertex, Edge, SingleEdge and Subgraph-1 shown in figure 3.1. The classes are extended as the size of subgraphs increase. For example, for size-2 subgraphs, TwoEdge and Subgraph-2 classes are used. Note that the paper focuses on directed labeled graphs where the direction is assumed to be from a smaller vertex number to the larger

vertex number. For example, if the vertices are given the numbers as 0, 1, 2, 3 etc., then the direction of the edges are considered to be from 0 to 1, 1 to 2 or 2 to 3 but not 3 to 1. Hence our method does not need any specific field to keep track of direction between the vertices. The Vertex class represents nodes in the graph. In the Vertex class, each object has a unique object identity which is 'VertexNo' and label as 'VertexLabel'. Figure 3.2 shows a simple subgraph where the numbers 1 and 2 represent vertex numbers which are allocated for ease of use, but these do not have any significance for subgraph mining. A and B represent labels of the vertices and C is the edge label. Each vertex object represents a node of the given graph; the Edge class is similar to the Vertex class. Each edge object represents an edge of the given graph. The SingleEdge class is the combination of the Vertex and Edge classes. It contains all the single-edged subgraphs. The Vertex and Edge classes are constructed separately as the Edge class does not contain the label details of the vertices.



*Figure 3.1 All Major Classes*

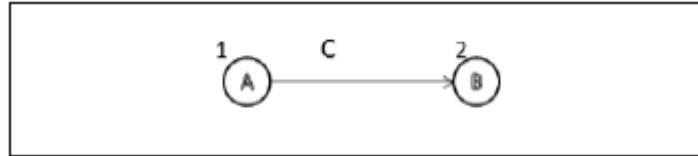


Figure 3.2 An Example Subgraph

The Subgraph-1 class includes all the subgraphs satisfying minimum support which is described in the subsequent sections. The Subscript '1' in Subgraph-1 represents the size-1 subgraphs. As the sizes of the subgraphs increase the subscript changes.

### 3.3.1 Subgraph Construction and FSG Determination

This section describes on subgraph construction starting from a single edge. Our focus is on labeled directed graph. In order to make each chapter self-contained, we provide some definitions as per necessity.

**Definition (labeled graph):** A labeled graph  $G$  is represented by a 4-tuple,

$G = (V, E, L, I)$  where

$V$  is a set of vertices (or nodes)

$E \subseteq V \times V$  is a set of edges, they can be directed or undirected

$L$  is a set of labels

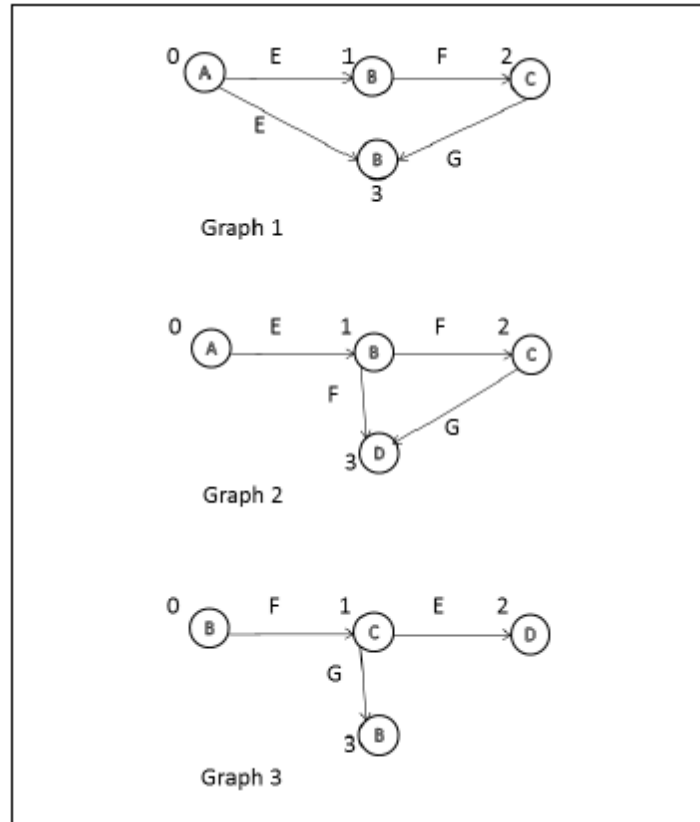
$I: V \cup E \rightarrow L$ ,  $I$  is a function assigning labels to the vertices and the edges

**Definition (Subgraph Support):** Let  $n_{\text{Graph}}$  be the total number of graphs in the dataset and  $n_{\text{SubGraph}}$  be the number of times a particular sub-graph appears in the dataset.

Then, the support 'Sup' of a particular subgraph is defined as:  $\text{Sup} = n_{\text{SubGraph}} \div n_{\text{Graph}}$

In figure 3.3, three transaction graphs are shown. The numbers 0, 1, 2 and 3 are the numbers assigned for programming purpose. The labels (names) A, B, C, D, E, F and

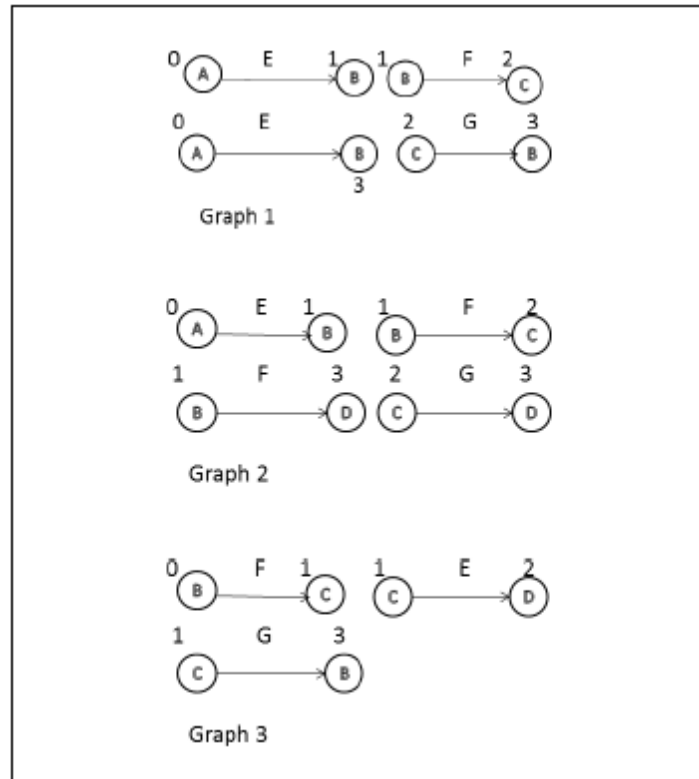
G are important to the algorithm. Graph isomorphism problem needs to be tackled while counting the support of subgraphs in the dataset. Two instances are isomorphic if the vertex and edge labels are same and directions are same. In our experiment, the direction is assumed to be from the lower numbered vertex to the higher numbered vertex. For example, if we count the number of occurrences of subgraph A-E-B in the three graphs, the count is 3, but in reality it is 2. Graph 1 contains the subgraph A-E-B twice. That must be counted once. This problem is eliminated by finding the distinct subgraphs per graph. Note that though we count only one instance of the subgraph per graph, we do not discard the other instances before pruning. The problem could be the instance omitted might have significance in the discovery of the subgraph of size 2. If we remove the pair 0-1 (vertex numbers) from graph 1 instead of 0-3, then in the next level, construction of subgraph-2 would not generate the subgraph A-B-C (0-1-2). So we store the other instances too.



*Figure 3.3 Representation of graphs in the dataset*

### **3.3.1.1 Subgraph Construction**

This section elaborates the process of subgraph construction. To begin with, we save the vertices and edges in different classes named Vertex and Edge classes. Since Edge class does not have information on the labels of the vertices, we join the Vertex and Edge classes based on the vertex numbers and the graph id to create the size-1 subgraphs stored in SingleEdge class. The graph id must be same during joining as the expansion happens in the same graph. SingleEdge class contains all the information on the vertices and edge labels. Considering the graphs shown in figure 3.4, the size-1 edges are shown in the diagrams.



*Figure 3.4 Objects of SingleEdge Class*

In order to provide a detailed view of the relational method DB-FSG [84] and object-oriented method, we have provided the tables along with the graph structures. Subsequent stages of construction are also shown in the figures. The edges are assigned a number to keep track of the edges joined during candidate generation. Actually, they are stored as objects in the db4o database. Table 3.1 contains all the objects in the Vertex class where each row represents the individual objects of the Vertex class. Vertex and Edge classes are not shown in graphical format.

Table 3.1 Vertex table

VertexNo	VertexLabel	GraphId
0	A	1
1	B	1
2	C	1
3	B	1
0	A	2
1	B	2
2	C	2
3	D	2
0	B	3
1	C	3
2	D	3
3	B	3

Table 3.2 Edge Table

V1	V2	EdgeLabel	GraphId
0	1	E	1
0	3	E	1
1	2	F	1
2	3	G	1
0	1	E	2
1	2	F	2
1	3	F	2
2	3	G	2
0	1	F	3
1	2	E	3
1	3	G	3

Similarly, table 3.2 has all the objects which are individual edge objects. After joining the Vertex and Edge classes, we obtained the SingleEdge class shown in figure 3.4 and in table 3.3. In order to generate size-2 subgraphs, each object of SingleEdge class is joined with itself. Similarly, size-3 subgraphs are constructed by joining size-2 subgraphs with size-1 subgraphs. In all cases, joining happens within the same graph. Unlike, relational databases where there is a defined join query using SQL; db4o does not have such join queries. Instead, it supports a

query called 'Native Query' which constrains the class to be joined and has a keyword called 'descend' which goes down to the field level to query the data.

We provide the details of construction of subgraphs of sizes more than one under fsg determination though they belong here.

*Table 3.3 Single-edge Table*

V1	V2	ENo	ELabel	V1L	V2L	GId
0	1	1	E	A	B	1
0	3	2	E	A	B	1
1	2	3	F	B	C	1
2	3	4	G	C	B	1
0	1	5	E	A	B	2
1	2	6	F	B	C	2
1	3	7	F	B	D	2
2	3	8	G	C	D	2
0	1	9	F	B	C	3
1	2	10	E	C	D	3
1	3	11	G	C	B	3

### **3.3.1.2 FSG Determination**

Frequent subgraphs are determined based on the number of times it appears in the whole dataset. If we consider the single-edge subgraphs shown in figure 3.4, there are eleven of them, but AEB (Graphs 1 and 2), CGB (Graphs 1 and 3) and BFC (Graph 1, 2 and 3) appear more than once. Hence it is obvious that the other subgraphs except AEB, CGB and BFC are insignificant. Our purpose is to find the subgraphs which occur more than a specific number of times (min-sup provided by the user) in the dataset. Let's consider the minimum support as 2, which mean a subgraph must be appearing in at least two graphs. Subgraph-1 class contains the following size-1 subgraphs shown in the figure 3.5. Notice that the Subgraph\_1 class does not have the numbers of the vertices. Only significant

details, the labels are stored. Though while counting, we do not consider the duplicate substructures, we do preserve them in order to ensure that we do not miss out any new substructure generation. Figure 3.6 shows the size-1 subgraphs which preserves the duplicates. After determining the frequent subgraphs in the first round, we move to generate size-2 substructures with the help of frequent size-1 subgraphs. Note that, we use substructures and subgraphs interchangeably, but they mean the same. Figures 3.7 shows the size-2 subgraphs after unconstrained expansion of size-1 subgraphs. Due to unconstrained expansion, we obtain many duplicate structures. Those duplicate structures are handled by keeping track of edge numbers. Figure 3.8 shows the subgraphs with only labels after pruning which are required for counting purposes.

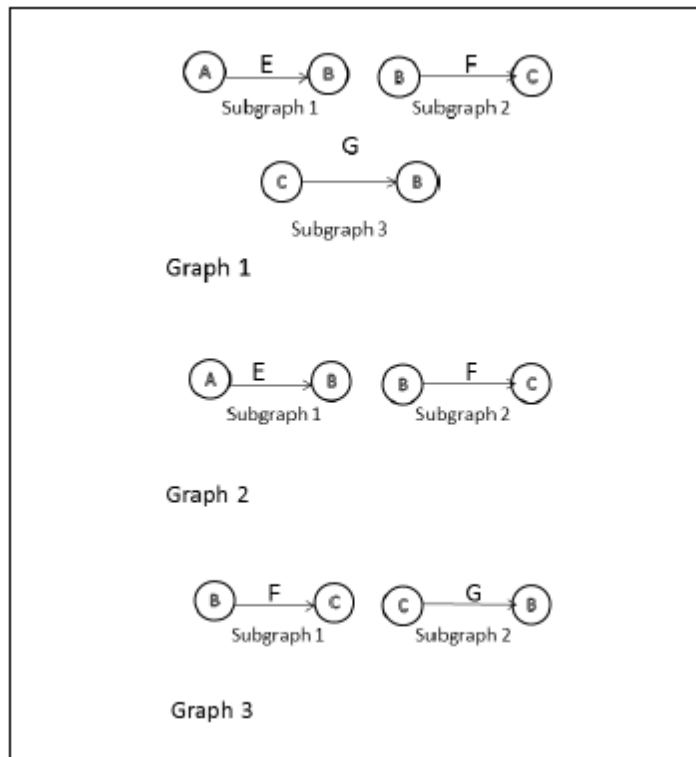


Figure 3.5 Objects of Subgraph\_1 class (satisfying min\_sup)

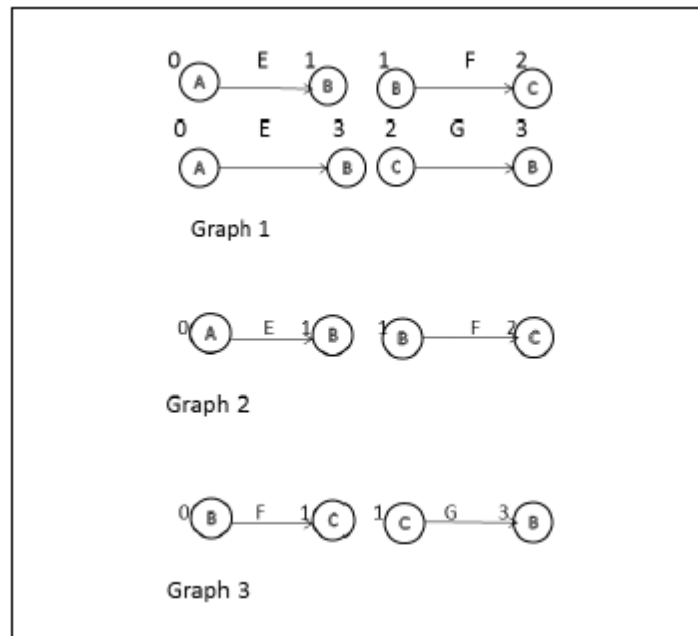


Figure 3.6 Objects of Single-edge After Pruning

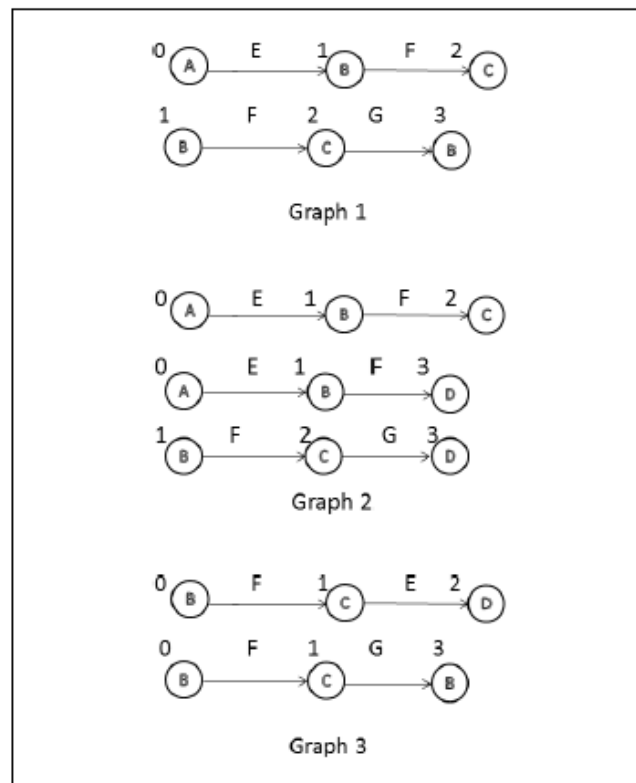


Figure 3.7 Objects of TwoEdge class (Before pruning)

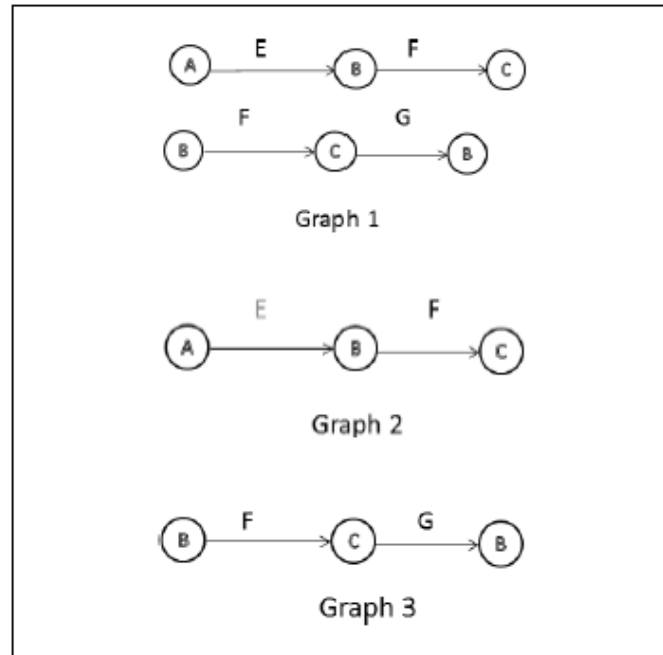


Figure 3.8 Objects of Subgraph\_2 class (satisfying min\_sup)

### 3.3.2 Optimization Techniques

This section discusses various optimization techniques used in object-oriented approach. We have used available data structures across the application to avoid frequent querying of the object database and hence increasing efficiency. Though data structures are used to make the processes faster, the applications are independent of each other, in other words, the graph dataset is always in db4o store. In order to retrieve the distinct instances (to tackle graph isomorphism), we used the data structure “hash sets”. In many places, common Java data structures are used to make application process faster. Subgraph counting time has been dramatically improved by using “MultiKey” and “MultiValueMap” common collections data structure available from apache.org. MultiKey can store the same sub-structure instances more than once; in other words, the

keys do not need to be unique. For example, considering figure 3.5 we can save the sub-structure A-E-B from both the graphs 1 and 2.

### ***3.3.3 DB-FSG vs OO-FSG Implementation***

The coding of Algorithm 1 was done in Java using Oracle 11g. The tables have the same name as the classes in db4o. We tried to optimize the relational method as much as possible by using indexes, and prepared statements for the insert statements. We noticed a significant time delay while inserting millions of records, whereas in db4o database it takes significantly less time. Initial data loading was quite time consuming, so we used Perl script to minimize the time by separating raw input data to Vertex, Edge and SingleEdge files to load into the relational database. For small sized datasets, the efficiency of relational and db4o approach are nearly same, but as the dataset size increases, the performance of db4o over relational increases dramatically. The only problem with db4o approach is it needs strong programming skills whereas relational approach solves things with simple queries. But, at the same time manipulating millions of database records through queries has a huge drag on efficiency. Also the join queries of SQL get messy when we join more than 2 tables. The queries of db4o database are quite simple. A comparison of both queries is given below.

The following is an example of a query used to join the matching vertices of the SingleEdge class/table with itself in order to obtain TwoEdge class/table objects/rows. In the SODA (db4o query) query the “vertexFrom” from the SingleEdge class is joined with the matching “vertexTo” of the same SingleEdge class. In the second statement of the code snippet the keyword “constrain” constrains the SingleEdge class. In the third statement the “descend” keyword means starting from the class level the query goes

down to one level to the “VertexFrom” field and ”constrain” keyword is used to match the “VertexTo” of the SingleEdge class. VertexFrom and VertexTo are the fields in the SingleEdge class and they are named so to indicate the direction of the edges. The last statement executes the query which retrieves all the matching objects in the class. We have also shown the SQL version of the query.

#### db4o version of a join query

```
query = db.query();
query.constrain(DB4OSingleEdge.class);
query.descend("VertexFrom")
      .constrain(VertexTo)
      .and(GraphId.constrain(GraphId));
query.execute();
```

#### SQL version of the same query

```
select distinct
  I1.VertexFrom as V1,
  I1.VertexTo as V2,
  I2.VertexTo as V3,
  I1.EdgeNo as E1No,
  I2.EdgeNo as E2No,
  I1.EdgeLabel as E1Label ,
  I2.EdgeLabel as E2Label,
  I1.VfromLabel as V1Label,
  I1.VtoLabel as V2Label,
  I2.VtoLabel as V3Label,
  I1.GraphId
from
  oneedge I1, oneedge I2
where
  I1.VertexTo = I2.VertexFrom and
  I1.GraphId = I2.GraphId;
```

### 3.4 Details of OO-FSG Algorithm

OO-FSG algorithm has two major aspects. One is generating candidates and another one is pruning the insignificant edges from the graphs. Each step of the algorithm is discussed in detail. In the algorithm, first step is for the construction of SingleEdge class from Vertex and Edge classes. In the second step, the distinct single edges are separated to get rid of isomorphic structures and stored in Subgraph-1 class.

**Algorithm 3.2 OO-FSG**

*Input: A graph dataset  $G_s$  and  $\text{min-sup}$*

*Output: The frequent subgraph set  $S$*

*Method:*

1. *construct SingleEdge class by joining Vertex and Edge class.*
  2. *select distinct single edges and store the subgraphs which satisfies  $\text{min-sup}$  in Subgraph-1 class.*
  3. *remove the edges with count less than the  $\text{min-sup}$  from SingleEdge.*
  4. *repeat steps a through e until a candidate subgraph of size- $N$  with  $\text{min-sup}$  is generated.*
    - (a) *join  $(N-1)$  Edge class with SingleEdge class to generate  $*(N)$ Edge.*
    - (b) *eliminate the redundant subgraphs from  $(N)$ Edge and store the size- $N$  subgraphs in Subgraph-Distinct- $N$  class.*
    - (c) *count the unique vertex and edge labels in the Subgraph-Distinct- $N$  class.*
    - (d) *eliminate the subgraphs from Subgraph-Distinct- $N$  with count less than  $\text{min-sup}$  and store it in Subgraph- $N$  class.*
    - (e) *remove the edges with count less than  $\text{min-sup}$  from  $(N)$ Edge class.*
  5. *end loop.*
- \* $(N)$  Edge: represents the TwoEdge, ThreeEdge, FourEdge and FiveEdge classes etc.*
-

Counting of the distinct edges is done using MultiKey and MultiValueMap on the whole dataset with the user provided minimum support (min-sup). In the third step, we remove the edges which fail to satisfy the minimum support value from the SingleEdge class. Step 4 is the looping condition, looping occurs from steps 4 (a) through 4 (e) until size-n which is 5 for our experiment. Step 4 (a) combines the SingleEdge class with itself based on the matching vertices and graph id. Step 4 (b) removes the redundant subgraphs to find the distinct instances and stores in the temporary class Subgraph-Distinct-2 class. In Step 4 (c), we count the subgraphs. When we say subgraphs, means only the edge labels and vertex labels not the numbers given to the nodes and edges. Steps 4 (d) and 4 (e) are self-explanatory. In the second iteration of the loop, we combine TwoEdge class with SingleEdge class and follow the steps accordingly. We keep repeating the loop until we get a subgraph of size-5.

**Candidate Generation:** this process is same as the subgraph construction described in section 3.3.1. First time the SingleEdge class is combined with itself. In subsequent iterations it is combined with TwoEdge, ThreeEdge and FourEdge classes as we are running the loop until size-5 subgraphs are generated.

**Frequency Counting and Pruning:** The subgraphs from the Subgraph-Distinct-1 class are searched for frequency counting on the vertex labels, edge labels. The subgraphs which meet the support value (user defined) are stored in a class called Subgraph-1. Edges are retained in the SingleEdge class where there is a matching; all other edges are pruned from the SingleEdge class. Similarly, for subsequent edge sizes the pruning is done based on minimum support threshold.

### 3.5 Experimental Details

The experiments were conducted on a Linux machine with 2 GB memory. The OO-FSG algorithm was coded in Java. The experimental results are shown in table 3.4 as well as in graphical format. The graphs in the figures 3.9 and 3.10 show the efficiency comparison DB-FSG and OO-FSG w.r.t minimum supports 1, 3, 5 and 7 respectively. We observed that using db4o database, efficiency is much higher than relational database. Also, scalability of db4o is higher than relational database.

*Table 3.4 DB-FSG [84] vs OO-FSG Performance*

Dataset size	Min_sup	DB-FSG	OO-FSG
50K	1%	357	353
100K	1%	1349	731
100K	3%	1220	656
100K	5%	1061	563
100K	7%	827	484
200K	1%	2439	1331
200K	3%	2002	1206
200K	5%	1717	1117
200K	7%	1622	1030
300K	1%	5887	2221
300K	3%	5394	2141
300K	5%	5137	2019
300K	7%	4164	1863
400K	1%	9502	2879
400K	3%	8228	2457
400K	5%	7156	2426
400K	7%	6962	2313

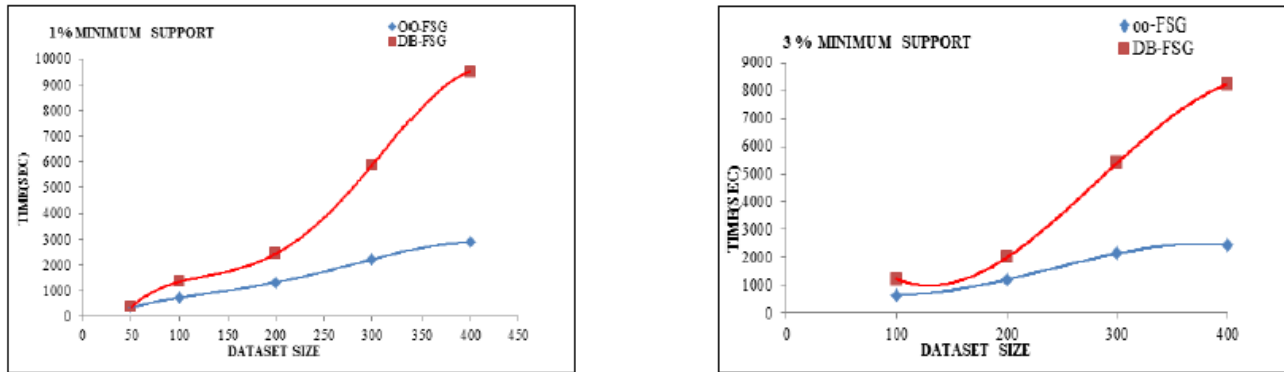


Figure 3.9 Comparison with 1% and 3% minimum support

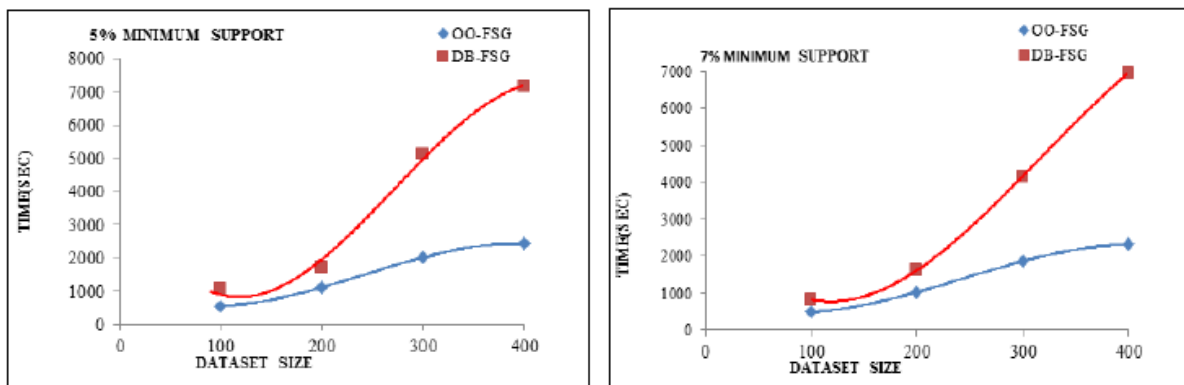


Figure 3.10 Comparison with 5% and 7% minimum support

#### 4 A MAPREDUCE-BASED FREQUENT SUBGRAPH MINING (MRFSM)

This chapter focuses on our approach to FSG based on MapReduce-based technique on transaction graphs. Parallel and distributed computing has taken a center stage as large-scale data processing has become almost impossible with main memory. Though traditional database-based approaches have provided some relief on processing large-scale databases, but we still reach at a bottleneck when we need to handle very large data. To solve this purpose, we need to make use of available hardware and software resources in an effective manner so that we can divide the

work between machines for independent processing. MapReduce framework by Google [51] motivated us to implement the frequent subgraph mining method on graph databases. There are a few research that have applied MapReduce for graph mining, which provided us with some motivation that we can apply the framework on frequent subgraph mining area.

#### **4.1 Background**

Finding frequent substructures from transaction databases in particular has a typical pattern, in the first step, we find all frequent subgraphs of size-1 and then step into subsequent iterations. While analyzing the compatibility of MapReduce model with this particular mining method, we figured out that the process of counting the frequency of isomorphic structures can be easily done with the help of key-value pairs. With respect to one key, which is a particular subgraph in our case, the respective values are the graph ids that contain the subgraph. Since we have so many machines available for our use, we can easily handle large amount of data in each step which used to be a bottleneck in our previous traditional database approach.

#### **4.2 Related Work**

There are several work aiming graph data mining using MapReduce model. We provide some of the related articles for reference before going into the detail of our approach. Luo et al. [79] proposed a subgraph query search method using MapReduce. Afrati et al. [76] proposed a MapReduce-based approach for finding all instances of a given sample graph in a large graph. Wu et al. [7] proposed a parallel subgraph mining algorithm using MapReduce where they took motif diameter and degrees of vertices are taken into consideration for motif matching. Liu et al. [63] proposed a MapReduce-based pattern finding algorithm MRPF for network motif detection from complex networks. Zhao et al. [69] proposed “SAHAD” a MapReduce-based color coding algorithm. Xiang et al. [78] present a MapReduce-based solution to the maximum clique

problem. PEGASUS [5] is an open source graph mining library developed by Kang et al. using MapReduce model. Aridhi et al. [43] proposed a density-based data partitioning approach on MapReduce framework. Bhuiyan and Hasan [81] proposed MIRAGE, a MapReduce-based approach in which they have adopted idea from gSpan [32] for right-most extension to prevent duplicate generation and a gSpan like dfs code for counting and isomorphism checks. Lin et al. [82] makes use of a memory-based algorithm GASTON [34] for their mining task. Data is partitioned between the machines and GASTON is applied to find locally frequent substructures. Then they perform a final scan to find all globally frequent subgraphs.

### **4.3 MapReduce Overview**

MapReduce, proposed by Google, is a distributed model for processing large-scale data. Users specify a map function and a reduce function. MapReduce takes in a list of key-value pairs, splits them among the possible map tasks, and then each map function produces any number of intermediate key-value pairs. Pairs with similar keys are gathered together at the reduce tasks, and then each reduce function performs computations before outputting values, which are either the final results, or possibly input for the next iteration. Ideally, MapReduce frameworks consist of several computers, usually referred to nodes, on the scale of tens to thousands. Processing occurs on data stored in the filesystem. Computation should be parallelized across the cluster, fault tolerant, and scheduled efficiently. We now go into some of specifics of the map and reduce functions. Figure 4.1 shows the MapReduce model.

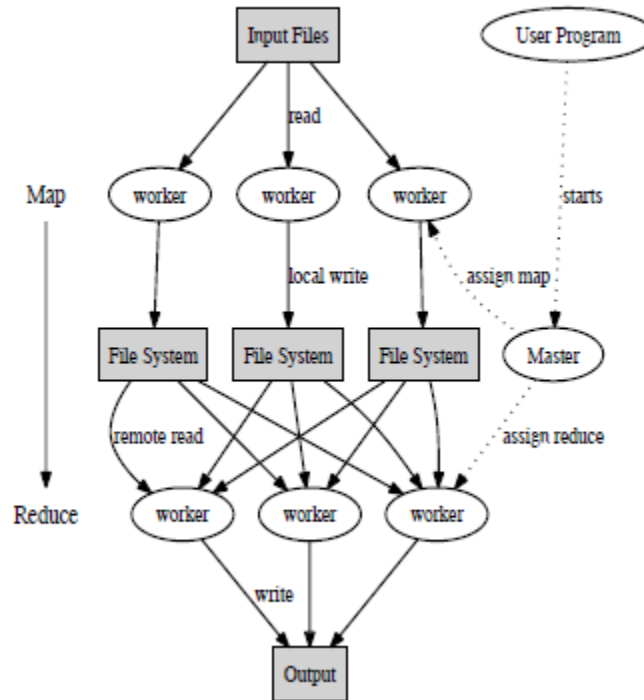


Figure 4.1 A MapReduce Model

### 4.3.1 Map Function

The mapper's job is to take in a key-value pair. This key-value pair often comes from a partition of data specified by the MapReduce architecture. After processing, the map function will emit another key-value pair. An added bonus comes in the form of an in-mapper combiner, which can do local computations to lessen the burden on the filesystem by acting as a mini-reducer. After all mappers have finished, all of the results are shuffled, sorted, and sent to the reducers.

### 4.3.2 Reduce Function

The reducer takes in a list of values corresponding to a specific key. Here, the reduce function can perform many operations, such as aggregations and summations. Since all

the values we need have been grouped, bulk computations on those values becomes trivial.

#### 4.4 Frequent Subgraph Mining Using MapReduce

We use Apache Hadoop [13], an open source framework derived from Google's MapReduce and Google File System, to generate the frequent subgraphs. Hadoop has become a popular approach for distributed and parallel computing due its top-level status within Apache, as well as being widely supported by the community. Computations through Hadoop are highly scalable and reliable, making Hadoop a very powerful tool for processing large datasets, or in the context of this chapter, large graph datasets. Using Hadoop iteratively, we can construct all isomorphic subgraphs that exceed a user defined support. We have two heterogeneous MapReduce jobs per iteration: one for gathering subgraphs for the construction of the next generation of subgraphs, and the other for counting these structures to remove irrelevant data. Figure 4.2 shows this workflow. We describe the process in more detail.

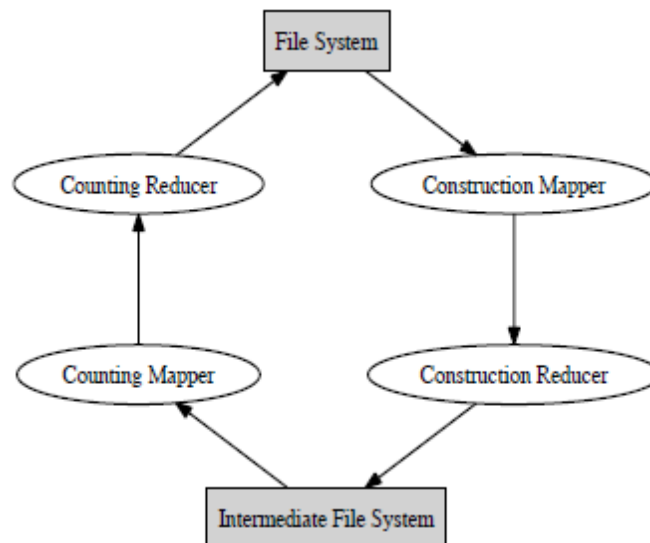


Figure 4.2 Frequent Subgraph mining using MapReduce

#### 4.4.1 FSG Determination

This section provides the frequent subgraph determination with respect to our example graph database. In figure 4.3, three transaction graphs are shown. The numbers 1, 2, 3, 4, and 5 are numbers assigned for programming purposes. The labels A, B, C, D, E, F, G, H, and J are important to the algorithm. Let  $n_{\text{Graph}}$  be the total number of graphs in the dataset and  $n_{\text{SubGraph}}$  be the number of times a particular subgraph appears in the dataset. Then the support  $\text{Sup}$  of a particular subgraph is defined as:

$$\text{Sup} = n_{\text{SubGraph}}/n_{\text{Graph}}$$

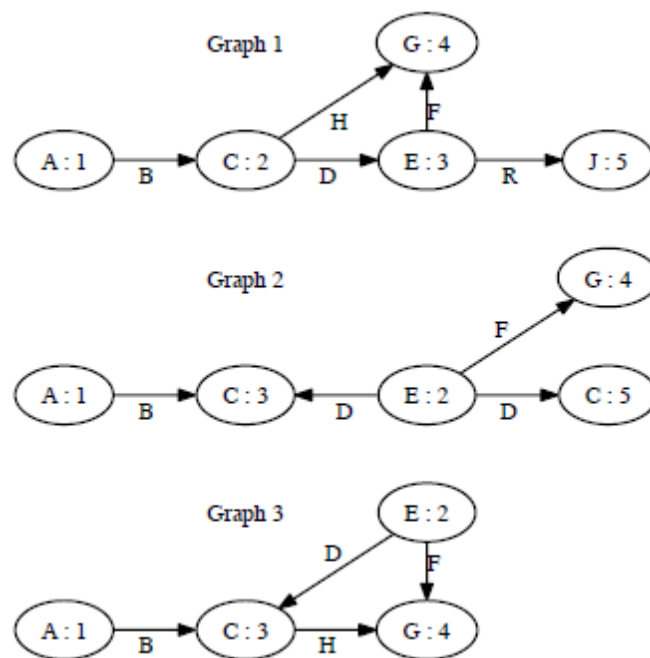


Figure 4.3 Example graphs in the Dataset

The graph isomorphism problem needs to be tackled while counting the support of subgraphs in the dataset. Two instances are isomorphic if the vertex and edge labels are same and the directions are the same. In our method, graphs can be directed and undirected, and the node numbers help identify cases of repeated labels. For example, if we count the number of occurrences of the subgraph E-D-C in the three graphs, the count is 4, but we only take the unique counts, so it's actually 3. Graph 2 contains the subgraph E-D-C twice. Do note that although we count it as one instance of the subgraph, we do not discard the other instance before pruning. Omitting that instance could be a potential problem when we construct the next generation of subgraphs. If the user support is taken as 2, then E-D-C/A-B-C are frequent subgraphs whereas E-R-J is not.

## 4.5 Subgraph Construction

This section elaborates on the process of subgraph construction. We explain in detail the process of map functions and reducer functions within each job of each iteration.

### 4.5.1 Map Function for Gathering Subgraphs with Similar Graph ID

Hadoop sends single lines from the input file to the mappers, to which each applies a map function to those lines. This initial map function will have the responsibility of sending the subgraph encoded in the input string to the correct reducer using the graph id. For the first iteration, the encoded input string will represent a single edge of the graph. For all other iterations, we have an encoded input string representing a subgraph of size  $k - 1$ .

*input key : offset of the input file for the string*

*input value : string representing a subgraph of size-( $k - 1$ ) and graph id*

*output key : graph id*

*output value : string representing the input subgraph*

### **4.5.2 Reducer for Constructing Subgraphs**

All of the subgraphs of size  $k-1$  with the same graph id are gathered for the reducer function. We note all of the single edges in these subgraphs and use that information to generate the next generation of possible subgraphs of size  $k$ . We encode this subgraph as a string just as was outputted from the previous map function. We keep all labels alphabetized and use special markers to designate differing nodes with the same labels. The results of this step are written out to the Hadoop File System.

*input key : graph id*  
*input values : list of subgraphs of size-( $k - 1$ ) encoded with graph id*  
*output key : encoded subgraph of size- $k$  and graph id*  
*output value : none*

### **4.5.3 Map Function for Gathering Subgraph Structures**

Similar to the process involving the first map function, Hadoop sends lines of input to the mappers. This second map function will have the responsibility of outputting the label-only subgraph encodings as a key and the node identification numbers and graph ids as values.

*input key : offset of the input file for the string*  
*input value : encoded string representing subgraph of size- $k$  and graph id*  
*output key : label-only string encoding subgraph*  
*output value : corresponding node ids and graph id*

### **4.5.4 Reducer for Determining Frequent Subgraphs**

The last reducer function per iteration will gather on label-only subgraph structures. The main task is to count the unique instances of the specific subgraph, which is done by iterating through the input values, incrementing a count, and ignoring subgraphs with previously seen graph ids. The label markers are removed at this point. At the end, if the count agrees with the given user defined support, it is written out to the Hadoop File

System for the next iteration, and otherwise it is ignored, effectively pruned. The output of iteration  $k$  is all subgraphs of size  $k$  that meet the support.

*input key* : label-only string encoding subgraph of size- $k$   
*input values* : list of corresponding node ids and graph ids  
*output key* : the encoded subgraph and graph id  
*output value* : none

#### 4.6 Details of MapReduce-FSG

MapReduce-FSG is an iterative algorithm that relies on two heterogeneous MapReduce Jobs.

The first job (denoted as  $A_k$ ) constructs size- $k$  subgraphs from size- $(k-1)$  subgraphs, while the second job (denoted as  $B_k$ ) will check whether or not a subgraph meets the user defined support.

The algorithm starts with single edges, and runs until there are no longer any frequent subgraphs constructed. Algorithms 4.1 and 4.2 highlight the tasks of  $A_k$ . Algorithms 4.3 and 4.4 outline the important steps of  $B_k$ . These algorithms are essential for pruning unnecessary subgraphs for the next iteration. Without them, we would quickly weigh down the disk and network.

##### **Algorithm 4.1 Map $A_k$**

*Input:* (offset, subgraph)

*parse subgraph for graph id*

*EMIT:* (graph id, subgraph)

##### **Algorithm 4.2 Reduce $A_k$**

*Input:* (graph id, subgraphs  $s_1, s_2, s_3, \dots$ )

*Edges*  $\leftarrow \varnothing$

*new Subgraphs*  $\leftarrow \varnothing$

*for all*  $s \in$  subgraphs *do*

*Retrieve all edges from*  $s$  *and add to* *Edges* *end for*

*for all  $s \in \text{subgraphs}$  do*

*Construct  $k$ -sized subgraphs from  $(k - 1)$ -sized  $s$  using edges from  $E$  dges that are eligible and add the new subgraph to  $\text{newS}$  ubgraphs*

*end for*

*for all  $s \in \text{newS}$  ubgraphs do*

*EMIT: (encoding for subgraph, empty text)*

*end for*

---

**Algorithm 4.3 Map  $B_k$**

*Input: (offset, encoded subgraph)*

*parse encoded subgraph for label-only subgraph*

*EMIT: (label-only subgraph, subgraph)*

---

**Algorithm 4.4 Reduce  $B_k$**

*Input: (label-only subgraph, subgraphs  $s_1, s_2, s_3, \dots$ )*

*GraphI Ds  $\leftarrow \varnothing$*

*count  $\leftarrow 0$*

*for all  $s \in \text{subgraphs}$  do*

*if  $s.\text{graphid} \in (\text{GraphI Ds})$  then*

*count  $\leftarrow \text{count} + 1$*

*GraphI Ds  $\leftarrow \text{GraphI Ds} \cup s.\text{graphid}$*

*end if*

*end for*

```

if count  $\geq$  user support then
  for all s  $\in$  subgraphs do
    EMIT: (subgraph, empty text)
  end for
end if

```

---

#### **4.6.1 Canonical Ordering of Elements**

As we are using Hadoop's Text to encapsulate a string object representing a subgraph, it is important to be able to differentiate between repetitive labels. We sort the outgoing nodes lexicographically based on label, and then use the unique id numbers if there still remains ambiguity. The sorting will help us with key matching, which is essential for our MapReduce approach. Reducer A will dynamically mark all node labels in the encoding Text so that we may distinguish between identical labels that belong to different nodes during Reducer B.

#### **4.6.2 Illustrative Example**

Here we illustrate our implementation of the MapReduce-FSG algorithm by showing outputs generated in various steps. We use the three sample graphs of figure 4.3. We will assume user-support is 2, meaning that we want all subgraphs that appear in at least 2 different graphs. The strings generated by the both the  $A_i$  and  $B_i$  steps are coded as three-part strings separated by "-". The first part represents the graph id, the second part represents a label-only subgraph, such as (A:B-C) standing for "node A has an edge B to node C", and the third part represents the subgraph using node id numbers, such as (1:3) standing for "node with id 1 has an edge to node with id 3."

#### 4.6.2.1 Step B1

As we are using single edges as the initial input, we do not need an A1, and can proceed directly to B1. We show the output below, represented in Figure 4.4 and the subgraph strings.

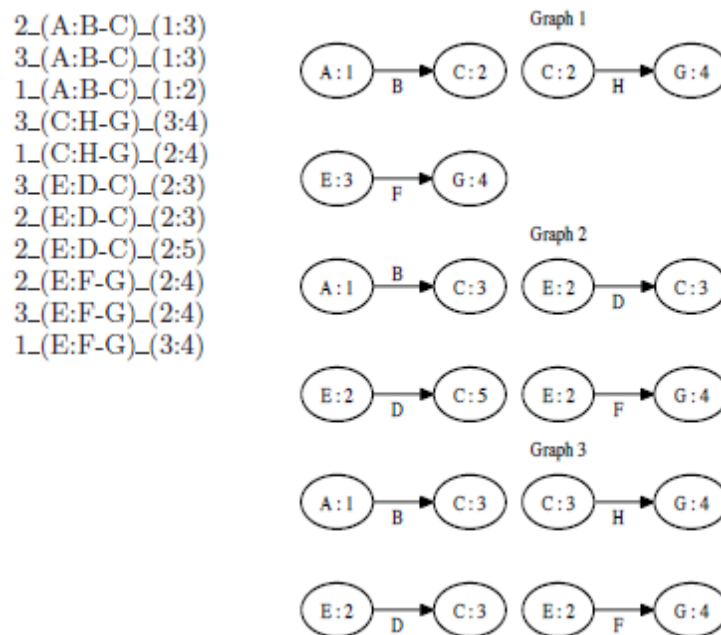


Figure 4.4 Single edge subgraphs that meet support

#### 4.6.2.2 Step A2

The worker for A2 will read input from the filesystem corresponding to the job of B1. The output strings are follows:

1\_(A^1:B-C^1)(C^1:H-G^1)\_(1:2)(2:4)

1\_(C^1:H-G^1)(E^1:F-G^1)\_(2:4)(3:4)

2\_(A^1:B-C^1)(E^1:D-C^1)\_(1:3)(2:3)

2\_(E^1:D-C^1,D-C^2)\_(2:3,5)

$2_{(E^1:D-C^1,F-G^1)}(2:3,4)$

$2_{(E^1:D-C^1,F-G^1)}(2:5,4)$

$3_{(A^1:B-C^1)(C^1:H-G^1)}(1:3)(3:4)$

$3_{(A^1:B-C^1)(E^1:D-C^1)}(1:3)(2:3)$

$3_{(C^1:H-G^1)(E^1:D-C^1)}(3:4)(2:3)$

$3_{(C^1:H-G^1)(E^1:F-G^1)}(3:4)(2:4)$

$3_{(E^1:D-C^1,F-G^1)}(2:3,4)$

Notice the "" used above. These are markers for the correct placement of labels.

Dealing with repetitive labels and subgraphs, we have to deal with a lot of ambiguity. In graph 2, we have  $2_{(E^1:D-C^1,D-C^2)}(2:3,5)$ . Without the marker, we would have  $(E:D-C,D-C)$ . To make sure we are following the substructure through multiple graph ids, we need those markers to remove confusion.

#### **4.6.2.3 Step B2**

The worker for B2 will read input from the filesystem corresponding to the job of A2. This input is an unfiltered group of size-2 subgraphs, and B2 will filter out results that do not agree with the user-support, as well as remove special markers. As a result, we obtain the subgraphs shown in figure 4.5 along with the subgraph strings.

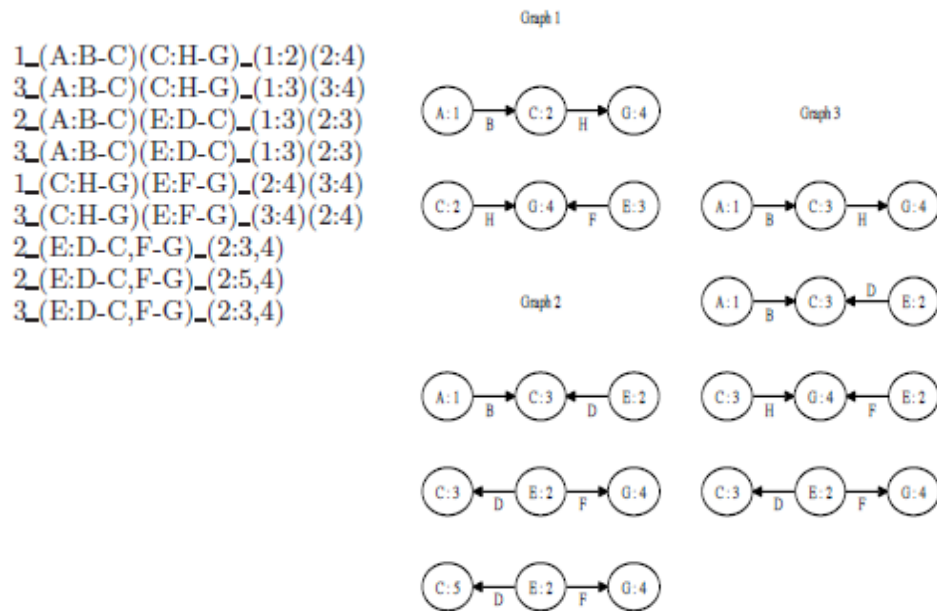


Figure 4.5 Double edge subgraphs that meet support

#### 4.6.2.4 Step A3 and B3

Similar to A2, we read from the results from the preceding B2 step. We arrive at the final result (represented in figure 4.6).

1\_(A^1:B-C^1)(C^1:H-G^1)(E^1:F-G^1)\_(1:2)(2:4)(3:4)  
 2\_(A^1:B-C^1)(E^1:D-C^1,D-C^2)\_(1:3)(2:3,5)  
 2\_(A^1:B-C^1)(E^1:D-C^1,F-G^1)\_(1:3)(2:3,4)  
 2\_(E^1:D-C^1,D-C^2,F-G^1)\_(2:3,5,4)  
 3\_(A^1:B-C^1)(C^1:H-G^1)(E^1:D-C^1)\_(1:3)(3:4)(2:3)  
 3\_(A^1:B-C^1)(C^1:H-G^1)(E^1:F-G^1)\_(1:3)(3:4)(2:4)  
 3\_(A^1:B-C^1)(E^1:D-C^1,F-G^1)\_(1:3)(2:3,4)  
 3\_(C^1:H-G^1)(E^1:D-C^1,F-G^1)\_(3:4)(2:3,4)

1\_(A:B-C)(C:H-G)(E:F-G)\_(1:2)(2:4)(3:4)  
 3\_(A:B-C)(C:H-G)(E:F-G)\_(1:3)(3:4)(2:4)  
 2\_(A:B-C)(E:D-C,F-G)\_(1:3)(2:3,4)  
 3\_(A:B-C)(E:D-C,F-G)\_(1:3)(2:3,4)

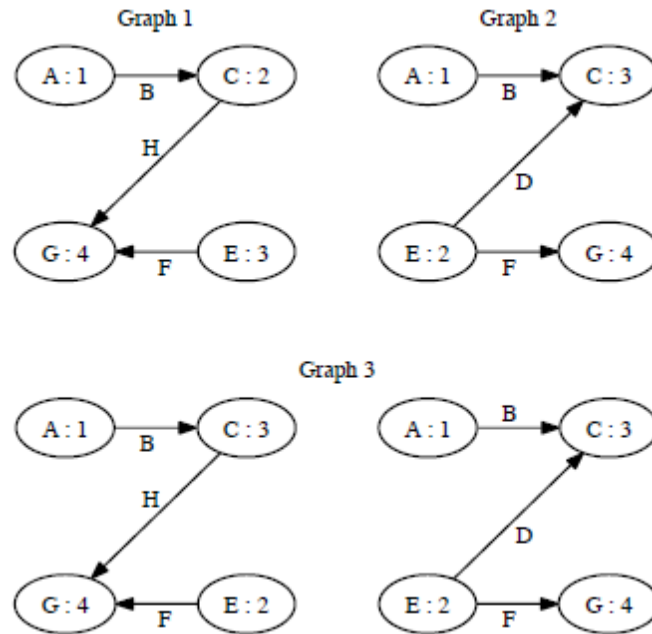


Figure 4.6 Triple edge subgraphs that meet support (the subgraph strings show on the top)

The left and right set of strings represent before and after minimum support calculation.

## 4.7 Experimental Details

The experiments were conducted on 4 Linux machines, each with 16 GB of memory and 2-4 quad core processors. The MapReduce-FSG algorithm was coded in Java as to work with Hadoop.

### 4.7.1 Synthetic Datasets

The experimental results are shown in Table 4.1 as well as in graphical format. The graphs in figures 4.7 shows the scalability of our method by comparing 2 and 4 sized clusters with varying supports. Even with our minimal setup, we managed to make

substantial gains. For our method, we performed experiments on datasets ranging from 100,000 to 1,000,000 transaction graphs. Each graph contains 30-50 edges and 30-50 vertices. The synthetic datasets were generated using a graph generator provided by the authors<sup>1</sup>. Tests were conducted with varying minimum support values 1%, 4%, and 7%. The maximum substructures is taken as four, and so we only iterate four times. Jumping from 2 nodes to 4 scaled very well for both increases in datasets, as well as number of nodes. Although we only have access to a modest cluster, it is easy to see the potential gains from large-scale clusters.

*Table 4.1 Performance of MapReduce-FSG  
(time in seconds)*

Dataset size	support	2 Nodes	4 Nodes
100K	1%	2471	1332
100K	4%	1718	1002
100K	7%	1203	721
400K	1%	2704	1559
400K	4%	2134	1217
400K	7%	1778	1018
1000K	1%	3702	2021
1000K	4%	3282	1809
1000K	7%	2786	1559

<sup>1</sup><http://www.cse.ust.hk/graphgen/>

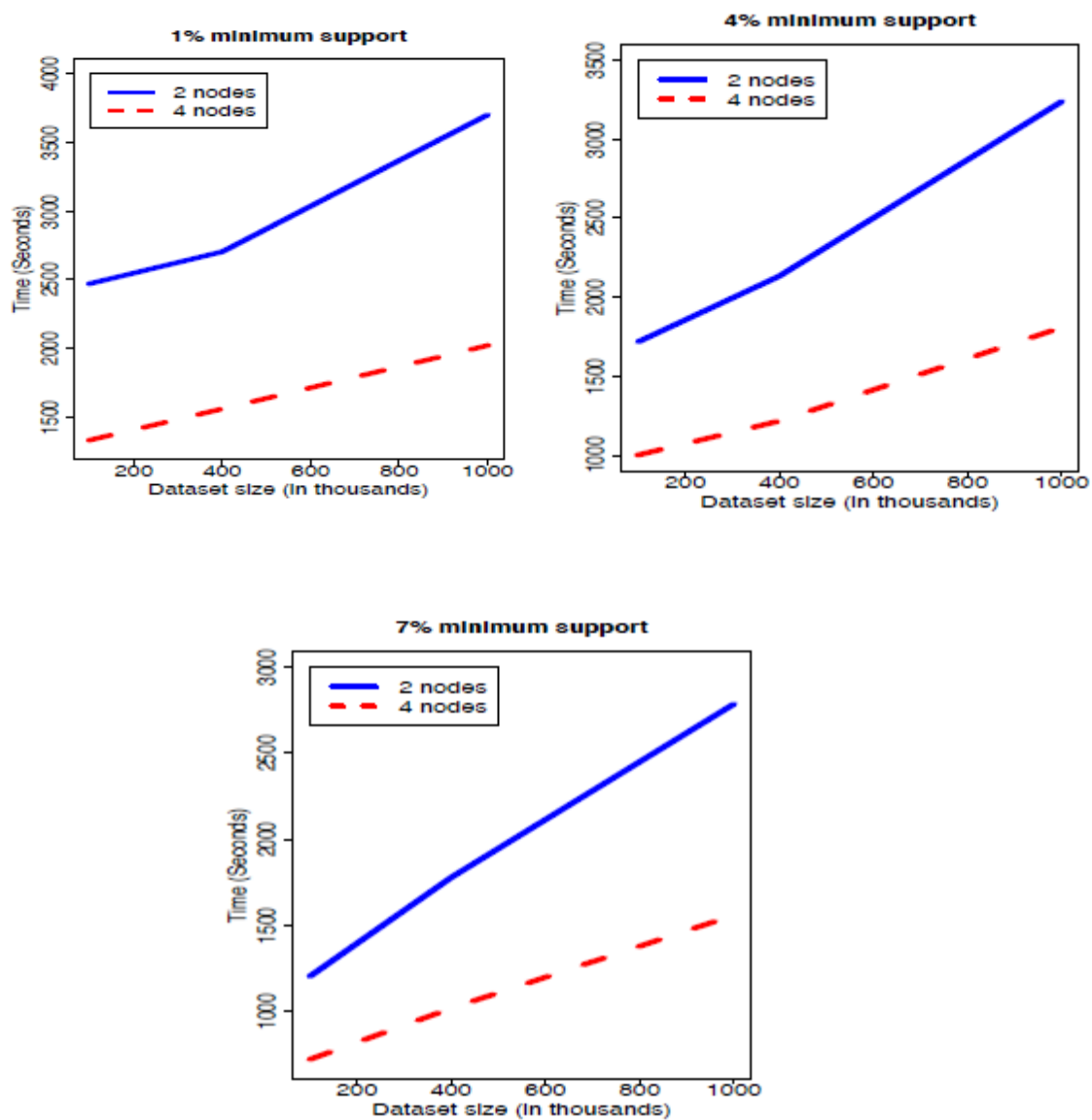


Figure 4.7 Comparison with 1%, 4% and 7% Support

#### 4.7.2 Biological Datasets

The real datasets are taken from an online source<sup>2</sup>, which contains data extracted from the PubChem website<sup>3</sup>. The dataset essentially contains the bioassay records for anti-cancer screen tests with different cancer cell lines, the outcome of which was either active or

inactive. We first ran our method on a cluster of size 2, and then again on a cluster of 4 to show the scalability. Results are shown in table 4.2, and graphically in figure 4.8.

---

<sup>2</sup><http://www.cs.ucsb.edu/~xyan/dataset.htm>

<sup>3</sup><http://pubchem.ncbi.nlm.nih.gov>

*Table 4.2 Performance on Biological datasets using a support of 50% and clusters of size 2 and 4 (in seconds)*

Dataset	active: 2	active: 4	inactive: 2	inactive: 4
MCF-7	33	587	1092	683
MOLT-4	22	556	1279	815
NCI-H23	15	516	1537	889
OVCAR-8	61	552	1257	844
P388	43	483	976	683
PC-3	57	546	1150	752
SF-295	36	528	1217	817
SN12C	13	502	1474	883
SW-620	59	568	1454	898
UACC257	36	536	1333	883
Yeast	10	607	1282	812

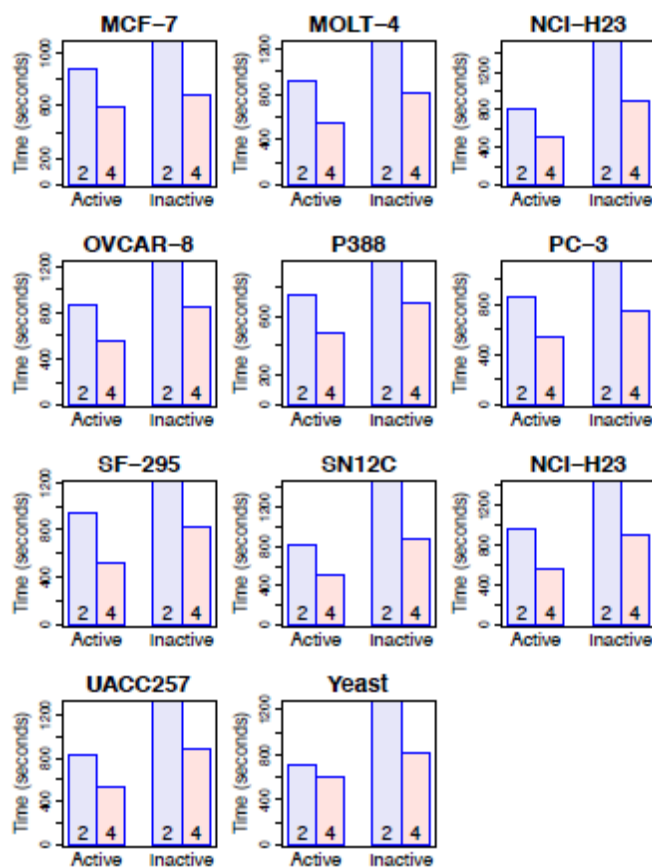


Figure 4.8 Results of Biological datasets. Each graph shows the runtimes for active and inactive outcomes on both clusters of size 2 and 4.

## 5 A HIGHLY SCALABLE FREQUENT SUBGRAPH MINING APPROACH USING APACHE SPARK (SPARKFSM)

Our major focus in this chapter is on mining frequent subgraphs from undirected transaction graphs using Apache Spark. A major part of our preliminary research focused on the directed graphs in chapter 3 and 4. Directed and undirected graphs are very different semantically. When we consider airline flight information graphs, those are directed and isomorphism detection is different in them than the chemical compound structures. Isomorphism plays a little different role here, for example, water (H<sub>2</sub>O), two hydrogen atoms share one electron each with the

oxygen atom forming the single covalent bond structure, and if we remove one H-O structure, then essence will be lost and we may lose many expected subgraphs. This is the reason we preserve the isomorphic structure during the first iteration while creating the single-edge structures, but do not count while determining frequency in undirected biological graphs. We provide the analysis in detail in the next sections.

## 5.1 Background

Based on our previous experiments using MapReduce, we noticed a few drawbacks specific to FSG mining algorithms. FSG process is normally an iterative style as subsequent steps use the result of the previous result. MapReduce does not offer this flexibility and two-steps of disk I/O are involved to read and write the same set. Spark with its distributed in-memory capability, we could achieve the iterative requirement on the fly. The Resilient Distributed Dataset (RDD) of Spark has the capability for lazy evaluation, so it helps in evaluation of a particular transformation at a later stage. Every step result RDD can be easily passed to the next step and based on the requirement the action is performed.

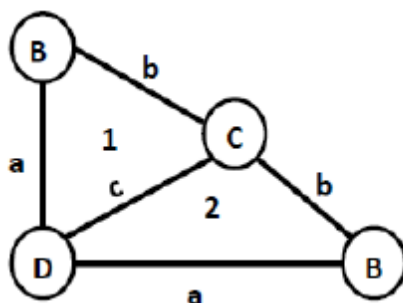
## 5.2 Related Work

Over the past years, Spark [87] has become the major industry standard for its distributed but in-memory processing of big data. As per our knowledge and findings, there are not many publications utilizing the power of Spark. Authors in [88] have used Spark to find the frequent subgraphs from single large graphs, which is not the major focus of our work. In this study, our focus is on the transactional setting. Authors in DIMSpan [85] have used Apache Flink, which is similar to Spark but mostly used for real-time processing. In their paper, their focus is on directed multi-graphs. To the best of our knowledge for the first time, we have introduced the ability of Spark engine on undirected transactional graphs. Leveraging the same utility, we could

see tremendous improvement on our previous MapReduce-based [80] approach on directed graphs.

### 5.3 FSM on Undirected Transaction Graphs

Frequent subgraph mining on undirected transaction graphs has a little different approach than the directed ones. While considering isomorphic structures, in the directed graphs, direction makes a subgraph different than the other even though it has same labels, but for undirected graph it is not the same. Since there is no direction, both the subgraphs indicated as 1 and 2 in figure 5.1 are identical and hence isomorphic.



*Figure 5.1 Isomorphic Structures*

We provide three example graphs in figure 5.2 to explain the undirected subgraph features and the pruning methodologies used. For the three example graphs, we have set the frequency threshold as 50%, so a subgraph needs to appear in at least two graphs. The subgraph B-b-C-b-B from figure 5.3 is retained in undirected graphs as it satisfies the support. Note here that even though B-b-C and C-b-B are isomorphic structures, we still keep it for undirected graphs, but count the frequency as one occurrence. The reason for this approach is very specific to the nature of chemical compound structures. We are providing a small comparison here to guide the readers through the process, in chemical compounds, ex. water (H<sub>2</sub>O), two hydrogen atoms share one

electron each with the oxygen atom forming the single covalent bond structure, and this is preserved in our experiment in undirected biological graphs. Similar is the case with  $\text{NH}_3$ , a compound consisting of Nitrogen and three Hydrogen atoms.

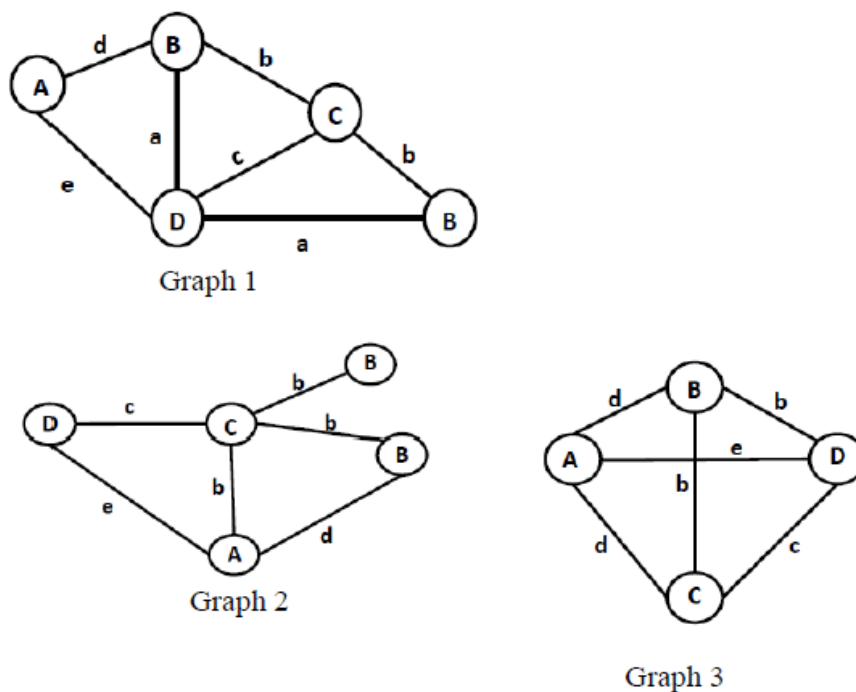


Figure 5.2 Undirected Graphs



Figure 5.3 Retained Structures

Similarly, pruning is done if a subgraph does not satisfy minimum threshold frequency level. Consider the 3-edge subgraphs shown in figure 5.4, both the subgraphs are pruned before we

reach the four-edge structure. They don't satisfy the minimum threshold criteria and appear only in one graph.

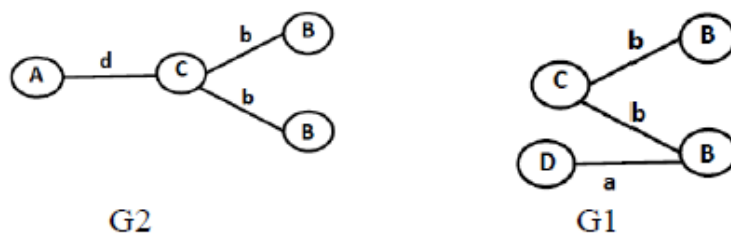


Figure 5.4 Pruned subgraphs

As we noticed from figure 5.3, the structures are retained during single and double edge formations due to the nature of chemical compounds. For isomorphic structure determination, instead of pruning subgraph 1 or 2 from figure 5.1, it is retained. At this level it becomes a little tricky to decide which one to keep and which one to eliminate from next computation. If 1 is pruned, the entire next generation will be lost. Instead of pruning, we decided to keep both the structures and their unique codes. While considering for the number of occurrences, we counted this as one. This way the next generation structures are not impacted. The unique code is the key factor for the undirected graphs. Each node and edge has been assigned a specific weight for programming purposes. In the algorithm 1, step 5 explains on the core components of the undirected algorithm.

### **Algorithm 5.1 Undirected Graphs**

*Input: Graph (G1), Frequency (f)*

*Output: Qualified Subgraph Edge list*

*Process:*

1) *G1.map => Load RDD1*

- 2)  $RDD_1.filter(count \geq f) \Rightarrow RDD_1$
- 3)  $RDD_1.map \Rightarrow SingleEdgeRDD$  (For each single edge in  $RDD_1$ , append  $reverse\_single\_edge$  to  $RDD_1$ )
- 4) Assign unique code to each unique node label
- 5)  $k\ EdgeRDD.join(SingleEdgeRDD) \Rightarrow k+1\_EdgeRDD$ 
  - Unidirection – join  $RDD_A.secondNode === RDD_B.firstNode$
  - Filter ( $RDD_A.graphID === RDD_B.graphID$ )
  - Generate unique code for each edge
  - Filter isomorphic structures
- 6)  $k+1\_EdgeRDD.groupby(code).count()$
- 7)  $k+1\_EdgeRDD.filter(count \geq f) \Rightarrow k+1\_EdgeRDD$
- 8) Repeat steps 5 – 7 for  $k+1\_EdgeRDD$
- 9) Repeat step 8 for 1 to  $n$  edge subgraphs

---

*\* $RDD_A$  and  $RDD_B$  represent the alias for  $SingleEdgeRDD$  for initial round, and it represents the future  $n$ -edge RDDs as  $RDD_A$  and  $SingleEdgeRDD$  as  $RDD_B$  for subsequent steps.*

---

As per our observation, the intermediate subgraphs meeting frequency threshold are very high. It is because of the isomorphic structure retention in the early stages like stage 1 and 2. We are not showing all the intermediate frequent subgraphs. There are many four-edge subgraphs, but we show only the important subgraphs that are unique for the undirected structures. From our observation, there are many subgraphs common across undirected and directed, but the directed

graphs do not produce the four-edge substructure shown in figure 5.5. Five edge frequent subgraphs are shown in figure 5.6.

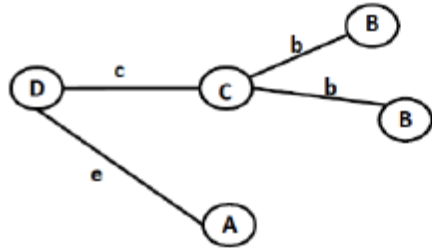


Figure 5.5 a G1, G2

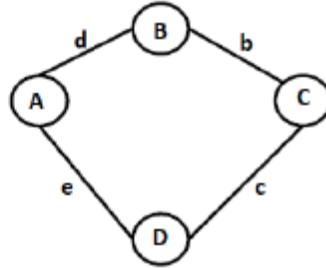


Figure 5.5 b G1, G2, and G3

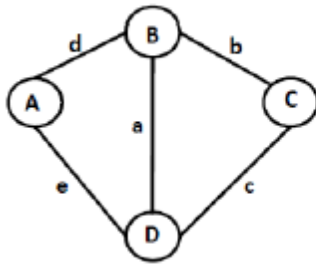


Figure 5.6 a G1, G3

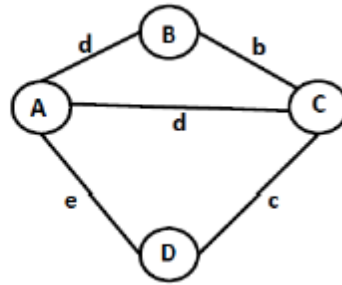
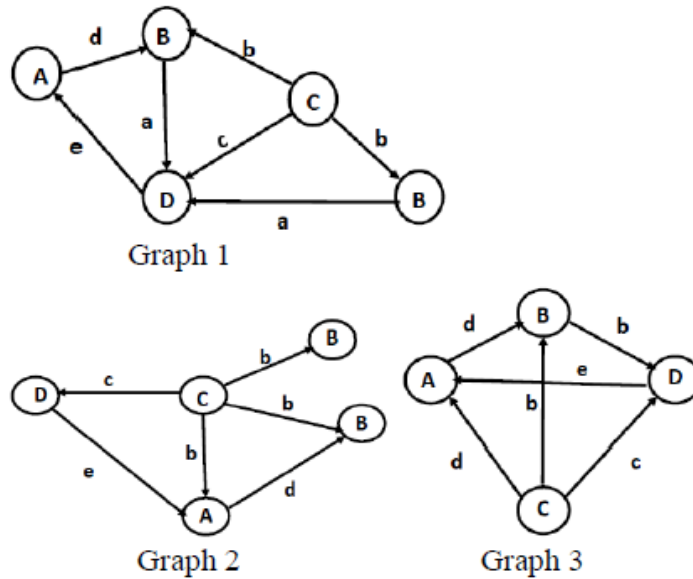


Figure 5.6 b G2, G3

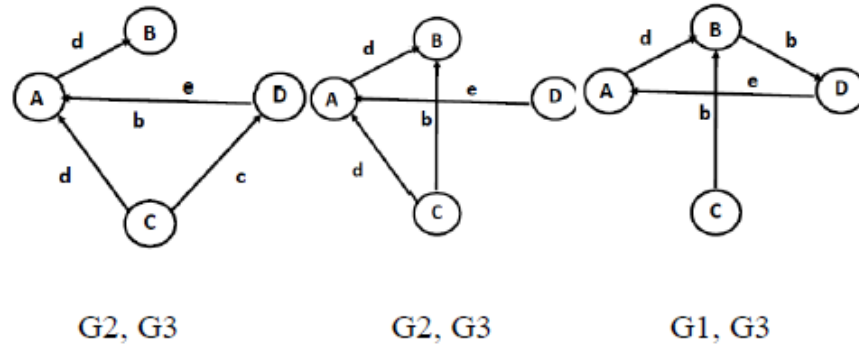
#### 5.4 FSM on Directed Transaction Graphs

This section describes the algorithm for directed graphs. The isomorphic structure determination for directed graphs are a little straight forward as it is based on the direction. As per our observation, many subgraphs that appear in the undirected results, don't appear in the directed structure. Many structures are pruned at very early stage. Figure 5.7 shows the directed graphs with the same three graphs used in undirected section, having direction attached to the nodes and edges.

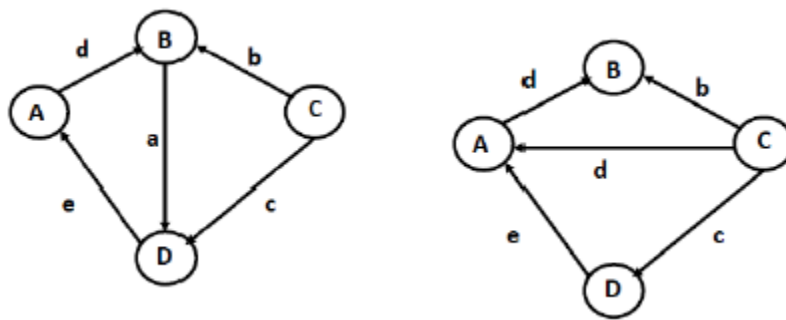


*Figure 5.7 Directed graphs*

One important thing to note here is, in the very early stage, single-edge filter prunes the C-b-B directed edge as part of isomorphic structure elimination. When we go to the next level subgraphs, the structure generated in figure 5.3 for undirected graphs does not exist. We observed that many substructures get pruned in the directed graphs. The graphs shown in figure 5.5 don't appear in directed graphs. Figure 5.8 shows the four-edge subgraphs from the directed structures (these graphs also appear in undirected structures as well, but it is not shown in figure 5.5). We found a very interesting pattern from the three example graphs; both directed and undirected graphs yield the same five-edge subgraphs shown in figure 5.9.



*Figure 5.8 Four-edge directed subgraphs*



*Parent graphs:  $G_1, G_3$*

*Parent graphs:  $G_2, G_3$*

*Figure 5.9 Five-edge directed subgraphs*

Algorithm 5.2 describes the directed graph subgraph finding process. Step 4 provides the helper methods used in the process. The converge keyword represents the structures where both subgraphs meet at one node w.r.t. their direction pointing to the node. Diverge keyword is used to represent the subgraphs whose edges depart from a node and point in opposite directions. Unidirection is used for the subgraphs whose edges form a path.

**Algorithm 5.2 Directed Graphs**

Input: Graph (G1), Frequency (f)

Output: Qualified Subgraph Edge list

Process:

- 1) G1.map => Load RDD1
- 2) RDD1.filter(count >= f) => RDD\_1
- 3) RDD\_1.filter(duplicate edges) => SingleEdgeRDD
- 4) kEdgeRDD.join(SingleEdgeRDD) => k+1\_EdgeRDD
  - Unidirection - join RDD<sub>A</sub>.secondNode === RDD<sub>B</sub>.firstNode
  - Converge – join RDD<sub>A</sub>.secondNode === RDD<sub>B</sub>.secondNode
  - Diverge – join RDD<sub>A</sub>.firstNode === RDD<sub>B</sub>.firstNode
  - Filter(RDD<sub>A</sub>.graphID === RDD<sub>B</sub>.graphID)
  - Eliminate isomorphic structures
  - Eliminate duplicates within same graphID
  - Assign Node labels according to the orientation of the join to maintain directional pattern
- 5) k+1\_EdgeRDD.groupBy(NodeLabel and edge pattern).count()
- 6) k+1\_EdgeRDD.filter(count >= f) => k+1EdgeRDD
- 7) Repeat steps 4 – 7 for k+1EdgeRDD
- 8) Repeat step 7 for 1 to n edge subgraphs

---

\*RDD<sub>A</sub> and RDD<sub>B</sub> represent the alias for SingleEdgeRDD for initial round, and it represents the future n-Edge RDDs as RDD<sub>A</sub> and SingleEdgeRDD as RDD<sub>B</sub> for subsequent steps.

---

## 5.5 Experimental Details

All the tests were conducted on AWS EMR [92] with 1 master node and 2 slave nodes with m4 large configuration. Spark 2.3 was used for all the experiments. Both directed and undirected jobs ran in parallel on the same cluster and this was an evaluation criterion to have parallel processing for all the experiments.

**Dataset Preparation:** We used the chemical compound dataset retrieved from the repository [89]. The dataset contains the bioassay records for anti-cancer screen tests with different cancer cell lines; they are categorized as active and inactive. Our initial round of experiments is conducted on the graphs as they appear on the site. Later, the data preparation was the most important criteria to test the scalability. A few authors concatenated the graphs from biological set to produce the larger sizes. After our analysis, we found that the graph sizes would not help much for proper evaluation if concatenated as it reads the last graph number and generates the next generation single edges and produce equal number of graphs. This way we can make sure that the evaluation is accurate for frequency determination. In addition, as the biological graphs contain only vertices, edge numbers and labels, we have written a Perl script that helps with the preprocessing steps to create the single edges. After the initial load, the data load is not required for the several runs, so the time taken by the initial load is ignored (approx. 15-20 seconds).

**Comparison:** Exact comparison with DIMSpan [85] would not be appropriate, as we have covered the undirected graphs in this research. The graphs generated for our evaluation are very complex due to the way they are created. It is not mere concatenation, rather every graph has millions of unique edges and the frequencies of new undirected graphs are massive. We did one level comparison with the biological directed graphs that shows somewhat comparable results. However, we see improvements over DIMSpan. Since the original biological graph sizes are not

very large, the time between DIMSpan and SparkFSM [90] would not differ much. Matching the MRFSM [80] computation time with the SparkFSM would not be fair as the technologies are different and Spark is in-memory computation. The table 5.2 below provides the computation time in seconds, size of graphs, number of approximate edges present. As observed, the original graphs take a few seconds for frequencies 10%, 20%, 25% and 50%. The comparison is based on both the undirected and directed implementations.

*Table 5.1 SparkFSM [90] performance analysis on biological graphs (time in seconds, threshold frequency: 50%)*

Graph	Size	Graph Nos.	Edge	Time(s) undirected	Time(s) Directed
MCF7A	1.3M	2293	18	5	30
MCF7HA	2.3M	2293	31	34	49
MCF7I	12M	25475	36	40	74
MCF7HI	20M	25475	59	91	77
MOLT4A	1.7M	3139	43	33	55
MOLT4HA	3M	3139	60	76	37
MOLT4I	17M	36624	36	84	63
MOLT4HI	29M	36624	59	74	75
NCIH23I	18M	38295	36	78	65
NCIH23HI	31M	38295	59	73	86
OVCAR8I	18M	38436	36	56	56
OVCAR8HI	20M	38436	48	45	33

MRFSM vs SparkFSM: We skipped the synthetic graphs' performance evaluation for this work, as those graph generators do not produce proper transaction after a certain point. As we observed, beyond 1000K limit, the number of new edge and vertices combination was very limited. The minimum support level was not able to match beyond 7%, which is not very practical in real life graph scenarios. Table 5.2 indicates our previous evaluation MRFSM [80] on the biological graphs with 2/4 node cluster using MapReduce model. It is evident from table

7, with similar number of nodes (3 nodes) in SparkFSM, the time has reduced to 5 seconds compared to the 587 seconds in the MRFSM approach.

*Table 5.2 MRFSM [80] performance analysis on biological graphs (time in seconds, threshold frequency: 50%)*

Dataset	Active: 2	Active: 4	Inactive: 2	Inactive: 4
MCF-7	833	587	1092	683
MOLT-4	922	556	1279	815
NCI-H23	815	516	1537	889
OVCAR-8	861	552	1257	844

*DIMSpan vs SparkFSM*: We used DIMSpan [85] as one of our evaluation standards, but DIMSpan has focused on the multi directed graphs as opposed to our SparkFSM [90], which is more focused on undirected graphs. From their Data Sets section 5.2, we noticed that they are simply copying the graphs several times to create the larger volume. For this reason, the comparison between DIMSpan and SparkFSM will not provide any valuable insight.

The table 5.3 shows the evaluation on undirected graphs. As described in the dataset preparation section, the graphs span from 50-100 edges. It became more complex after the graphs were duplicated with a new number assigned to each graph. We created graphs up to 4 million and captured the time in minutes. Graph sizes range from 124MB to 2.1GB.

Table 5.3 SparkFSM [90] performance analysis on large undirected datasets (time in minutes)

Graph	Support	Graph Nos.	Time (min)
OVCAR8HI	75%	153,180	2.2
OVCAR8HI	90%	153,180	0.7
OVCAR8HI	75%	306,366	4.0
OVCAR8HI	90%	306,366	0.96
OVCAR8HI	75%	1,225,465	13
OVCAR8HI	90%	1,225,465	2
OVCAR8HI	75%	2,450,931	26
OVCAR8HI	90%	2,450,931	4

Figure 5.10 and 5.11 shows the graph plots for undirected and directed performance comparisons.

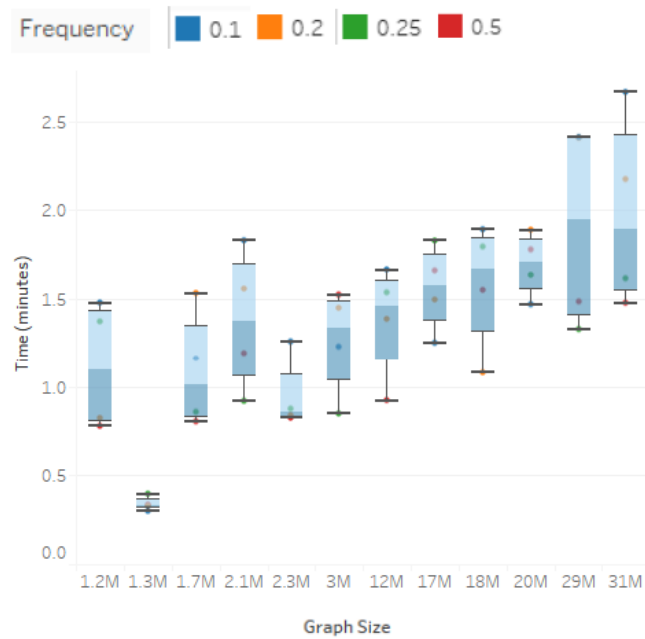
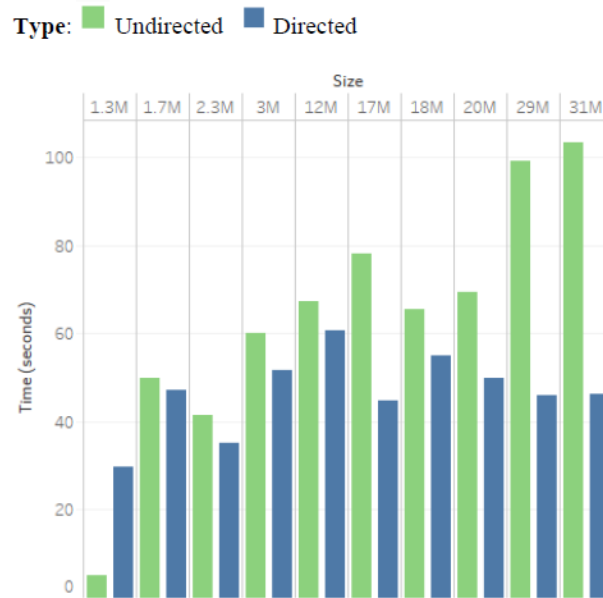


Figure 5.10 Box and Whisker plot showing time required to compute each undirected graph size at the varying frequencies



*Figure 5.11 Performance Comparison between Directed and Undirected on Biological Graph Dataset*

## 6 CONCLUDING REMARKS

Graphs are everywhere. Knowledge mining from graph data has been a major focus with the evolution of computer technology, social networking sites, and web logs as these generate a lot of graphs. Only proper mining methods can dig deep into the abundance of knowledge hidden inside these graphs. Since the graphs are huge in size, there is a need for high performance technology to find the frequent substructures. During my research journey, the goal was to develop high performance mining methods to find useful frequent patterns from the transactional graphs. A very good review is also provided for the readers starting with FSM's inception and on the status until writing this work. With the rapid progress in big data technologies many issues are easily handled. Some analysis are provided based on the experience while experimenting

different approaches on transactional FSM. Recently the entire research work is compiled into a journal and getting ready for publication [93].

Single machine memory-based vs RDBMS: the major difference between these two are that RDBMS can contain more data during processing making it more scalable. Memory based approaches are very efficient if the dataset size is small enough to fit to the data structure in use. Certain built-in functions such as groupBy and distinct can help to a greater extent, the problem can be solved via SQL query and can potentially reduce the programming. Intermediate results can be available even after the job is no longer active which not the case for memory-based approach is where if the job is complete, the results will be removed from memory.

RDBMS vs Object Oriented Approach: being motivated by the RDBMS based paper [84], we used Db4o while experimenting on FSM, and it is an open source object db. The interesting aspect of db4o is that the user does not need to create a separate data model, the applications class model defines the structure of the data in db4o database. Db4o database provides persistence to objects automatically. Object persistence is the capability of the system to hold objects even after the system stops running. We observed improvement with our Db4o approach over the RDBMS based approach DB-FSG [84].

Object Oriented Approach vs Hadoop MapReduce: Our second experiment on FSM was motivated by Hadoop/MapReduce which came as a savior for very big data processing with its additional benefit of the reducer concept in MapReduce model. The reduce function has in-built capability of accumulating all the key-value pairs and summing it on the go, and this helped us with the frequency counting. Since then cluster computing has become a normal standard and comparing the database-oriented approach with the MapReduce model felt like comparing apples with oranges. We could work on the real life complicated anti-cancer datasets and tremendous

improvement gain was observed.

*Hadoop MapReduce vs Spark/Scala:* during the experiment with MapReduce model, we faced some drawbacks of disk I/O due to the intermediate results being written to disk and then read again, which added two extra layers of I/O. All our issues are easily resolved with the Spark engine using Scala language. Many benefits are achieved by this: 1) it is distributed computing which happens in-memory, 2) the need for iterative style of algorithm for FSM comes as a well-built functionality with the concept of Spark's RDD (Resilient Distributed Dataset). 3) Scala, being a functional style language, it has many advantages over any verbose programming and being the language base for Spark, it comes with many compatible functions that make several lines of code to a few lines. Performance improvements are multifold as observed from our experiments. The same graph with 3 nodes with MapReduce took ~500 seconds, but the Spark/Scala implementation took about 5 seconds.

## BIBLIOGRAPHY

- [1] Agrawal, R., & Srikant, R. (1994, September). Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB* (Vol. 1215, pp. 487-499).
- [2] Jiang, X., Xiong, H., Wang, C., & Tan, A. H. (2009). Mining globally distributed frequent subgraphs in a single labeled graph. *Data & Knowledge Engineering*, 68(10), 1034-1058.
- [3] Kuramochi, M., & Karypis, G. (2005). Finding frequent patterns in a large sparse graph\*. *Data mining and knowledge discovery*, 11(3), 243-271.
- [4] Kuramochi, M., & Karypis, G. (2004, November). Grew-a scalable frequent subgraph discovery algorithm. In *Data Mining, 2004. ICDM'04. Fourth IEEE International Conference on* (pp. 439-442). IEEE.
- [5] Kang, U., Tsourakakis, C. E., & Faloutsos, C. (2009, December). Pegasus: A peta-scale graph mining system implementation and observations. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on* (pp. 229-238). IEEE.
- [6] Reinhardt, S. P., & Karypis, G. (2007, March). A Multi-Level Parallel Implementation of a Program for Finding Frequent Patterns in a Large Sparse Graph. In *IPDPS* (pp. 1-8).
- [7] Wu, B., & Bai, Y. (2010). An efficient distributed subgraph mining algorithm in extreme large graphs. In *Artificial Intelligence and Computational Intelligence* (pp. 107-115). Springer Berlin Heidelberg.
- [8] Liu, C., Yan, X., Yu, H., Han, J., & Philip, S. Y. (2005). Mining Behavior Graphs for "Backtrace" of Noncrashing Bugs. In *SDM* (pp. 286-297).
- [9] Milo, R., Shen-Orr, S., Itzkovitz, S., Kashtan, N., Chklovskii, D., & Alon, U. (2002). Network motifs: simple building blocks of complex networks. *Science*, 298(5594), 824-827.
- [10] Milo, R., Kashtan, N., Itzkovitz, S., Newman, M. E. J., & Alon, U. (2003). On the uniform generation of random graphs with prescribed degree sequences. *arXiv preprint cond-mat/0312028*.
- [11] Borgelt, C., & Berthold, M. R. (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.
- [12] <http://www.db4o.com/>
- [13] [http://hadoop.apache.org/docs/r1.2.1/mapred\\_tutorial.html](http://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html)

- [14] Agrawal, R., Imieliński, T., & Swami, A. (1993, June). Mining association rules between sets of items in large databases. In *ACM SIGMOD Record* (Vol. 22, No. 2, pp. 207-216). ACM.
- [15] Burdick, D., Calimlim, M., & Gehrke, J. (2001). MAFIA: A maximal frequent itemset algorithm for transactional databases. In *Data Engineering, 2001. Proceedings. 17th International Conference on* (pp. 443-452). IEEE.
- [16] Bayardo Jr, R. J. (1998, June). Efficiently mining long patterns from databases. In *ACM Sigmod Record* (Vol. 27, No. 2, pp. 85-93). ACM.
- [17] Han, J., Pei, J., & Yin, Y. (2000, May). Mining frequent patterns without candidate generation. In *ACM SIGMOD Record* (Vol. 29, No. 2, pp. 1-12). ACM.
- [18] Zaki, M. J., & Hsiao, C. J. (2002, April). CHARM: An Efficient Algorithm for Closed Itemset Mining. In *SDM* (Vol. 2, pp. 457-473).
- [19] Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., & Hsu, M. C. (2001, April). Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)* (pp. 0215-0215). IEEE Computer Society.
- [20] Mannila, H., Toivonen, H., & Verkamo, A. I. (1997). Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3), 259-289.
- [21] Agrawal, R., & Srikant, R. (1995, March). Mining sequential patterns. In *Data Engineering, 1995. Proceedings of the Eleventh International Conference on* (pp. 3-14). IEEE.
- [22] Asai, T., Abe, K., Kawasoe, S., Sakamoto, H., & Arikawa, S. (2001). Efficient Substructure Discovery from Large Semi-structured Data.
- [23] Zaki, M. J. (2002, July). Efficiently mining frequent trees in a forest. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 71-80). ACM.
- [24] Fortin, S. (1996). *The graph isomorphism problem*. Technical Report 96-20, University of Alberta, Edmonton, Alberta, Canada.
- [25] Ghazizadeh, S., & Chawathe, S. S. (2002, January). SEuS: Structure extraction using summaries. In *Discovery science* (pp. 71-85). Springer Berlin Heidelberg.
- [26] Vanetik, N., Gudes, E., & Shimony, S. E. (2002). Computing frequent graph patterns from semistructured data. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on* (pp. 458-465). IEEE.
- [27] Yoshida, K., Motoda, H., & Indurkha, N. (1994). Graph-based induction as a unified learning framework. *Applied Intelligence*, 4(3), 297-316.

- [28] Holder, L. B., Cook, D. J., & Djoko, S. (1994, July). Substructure Discovery in the SUBDUE System. In *KDD workshop* (pp. 169-180).
- [29] Dehaspe, L., Toivonen, H., & King, R. D. (1998, August). Finding Frequent Substructures in Chemical Compounds. In *KDD* (Vol. 98, p. 1998).
- [30] Inokuchi, A., Washio, T., & Motoda, H. (2000). An apriori-based algorithm for mining frequent substructures from graph data. In *Principles of Data Mining and Knowledge Discovery* (pp. 13-23). Springer Berlin Heidelberg.
- [31] Inokuchi, A., Washio, T., Nishimura, K., & Motoda, H. (2002). A fast algorithm for mining frequent connected subgraphs. *IBM Research, Tokyo Research Laboratory, Tech. Rep.*
- [32] Yan, X., & Han, J. (2002). gspan: Graph-based substructure pattern mining. In *Data Mining, 2002. ICDM 2003. Proceedings. 2002 IEEE International Conference on* (pp. 721-724). IEEE.
- [33] Huan, J., Wang, W., & Prins, J. (2003, November). 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.
- [34] Nijssen, S., & Kok, J. N. (2004, August). A quickstart in frequent structure mining can make a difference. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 647-652). ACM.
- [35] Wang, J., Hsu, W., Lee, M. L., & Sheng, C. (2006, April). A partition-based approach to graph mining. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on* (pp. 74-74). IEEE.
- [36] Nguyen, S. N., Orłowska, M. E., & Li, X. (2008, January). Graph mining based on a data partitioning approach. In *Proceedings of the nineteenth conference on Australasian database-Volume 75* (pp. 31-37). Australian Computer Society, Inc..
- [37] Wang, C., Wang, W., Pei, J., Zhu, Y., & Shi, B. (2004, August). Scalable mining of large disk-based graph databases. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 316-325). ACM.
- [38] Savasere, A., Omiecinski, E., & Navathe, S. B. (1995, September). An Efficient Algorithm for Mining Association Rules in Large Databases. In *Proceedings of the 21th International Conference on Very Large Data Bases* (pp. 432-444). Morgan Kaufmann Publishers Inc..
- [39] Nguyen, S. N., & Orłowska, M. E. (2005). Improvements in the data partitioning approach for frequent itemsets mining. In *Knowledge Discovery in Databases: PKDD 2005* (pp. 625-633). Springer Berlin Heidelberg.

- [40] Liu, C., Yan, X., Fei, L., Han, J., & Midkiff, S. P. (2005). SOBER: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 30(5), 286-295.
- [41] Jiawei, H., & Kamber, M. (2001). Data mining: concepts and techniques. *San Francisco, CA, itd: Morgan Kaufmann*, 5.
- [42] Yan, X. (2006). Mining, indexing and similarity search in large graph data sets, *Comp sci dissertation*.
- [43] Aridhi, S., d'Orazio, L., Maddouri, M., & Mephu Nguifo, E. (2013). Density-based data partitioning strategy to approximate large-scale subgraph mining. *Information Systems*.
- [44] Di Fatta, G., & Berthold, M. R. (2006). Dynamic load balancing for the distributed mining of molecular structures. *Parallel and Distributed Systems, IEEE Transactions on*, 17(8), 773-785.
- [45] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to algorithms* (Vol. 2, pp. 531-549). Cambridge: MIT press.
- [46] Cook, D. J., & Holder L. B. (2000). Graph-based data mining. *IEEE Intelligent Systems*, vol. 15, no. 2, pp. 32-41.
- [47] Chakravarthy, S., Beera, R., & Balachandran, R. (2004). DB-Subdue: Database approach to graph mining. In *Advances in Knowledge Discovery and Data Mining* (pp. 341-350). Springer Berlin Heidelberg.
- [48] Rissanen, J. (1989). *Stochastic complexity in statistical inquiry theory*. World Scientific Publishing Co., Inc..
- [49] Padmanabhan, S., & Chakravarthy, S. (2009). HDB-Subdue: A Scalable Approach to Graph Mining. *Data Warehousing and Knowledge Discovery*, 325-338.
- [50] Srichandan, B., & Sunderraman, R. (2011, December). Oo-fsg: An object-oriented approach to mine frequent subgraphs. In *Proceedings of the Ninth Australasian Data Mining Conference-Volume 121* (pp. 221-228). Australian Computer Society, Inc..
- [51] Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 107-113.
- [52] Agarwal, R. C., Aggarwal, C. C., & Prasad, V. V. V. (2001). A tree projection algorithm for generation of frequent item sets. *Journal of parallel and Distributed Computing*, 61(3), 350-371.
- [53] Zaki, M. J., & Gouda, K. (2003, August). Fast vertical mining using diffsets. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 326-335). ACM.

- [54] Fang, W., Lu, M., Xiao, X., He, B., & Luo, Q. (2009, June). Frequent itemset mining on graphics processors. In *Proceedings of the fifth international workshop on data management on new hardware* (pp. 34-42). ACM.
- [55] Li, H., & Zhang, N. (2010, August). Mining maximal frequent itemsets on graphics processors. In *Fuzzy Systems and Knowledge Discovery (FSKD), 2010 Seventh International Conference on* (Vol. 3, pp. 1461-1464). IEEE.
- [56] Li, H. (2010, June). A GPU-based maximal frequent itemsets mining algorithm over stream. In *Computer and Communication Technologies in Agriculture Engineering (CCTAE), 2010 International Conference On* (Vol. 1, pp. 289-292). IEEE.
- [57] Amossen, R. R., & Pagh, R. (2011, May). A new data layout for set intersection on gpus. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International* (pp. 698-708). IEEE.
- [58] Teodoro, G., Mariano, N., Meira, W., & Ferreira, R. (2010, October). Tree projection-based frequent itemset mining on multicore cpus and gpus. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on* (pp. 47-54). IEEE.
- [59] Cheung, D. W., Han, J., Ng, V. T., Fu, A. W., & Fu, Y. (1996, December). A fast distributed algorithm for mining association rules. In *Parallel and Distributed Information Systems, 1996., Fourth International Conference on* (pp. 31-42). IEEE.
- [60] Liu, L., Li, E., Zhang, Y., & Tang, Z. (2007, September). Optimization of frequent itemset mining on multiple-core processor. In *Proceedings of the 33rd international conference on Very large data bases* (pp. 1275-1285). VLDB Endowment.
- [61] Li, H., Wang, Y., Zhang, D., Zhang, M., & Chang, E. Y. (2008, October). Pfp: parallel fp-growth for query recommendation. In *Proceedings of the 2008 ACM conference on Recommender systems* (pp. 107-114). ACM.
- [62] Miliaraki, I., Berberich, K., Gemulla, R., & Zoupanos, S. (2013, June). Mind the gap: large-scale frequent sequence mining. In *Proceedings of the 2013 international conference on Management of data* (pp. 797-808). ACM.
- [63] Liu, Y., Jiang, X., Chen, H., Ma, J., & Zhang, X. (2009). Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network. In *Advanced Parallel Processing Technologies* (pp. 341-355). Springer Berlin Heidelberg.
- [64] Buehrer, G., Parthasarathy, S., & Chen, Y. K. (2006, December). Adaptive parallel graph mining for cmp architectures. In *Data Mining, 2006. ICDM'06. Sixth International Conference on* (pp. 97-106). IEEE.

- [65] Cook, D. J., Holder, L. B., Galal, G., & Maglothin, R. (2001). Approaches to parallel graph-based knowledge discovery. *Journal of Parallel and Distributed Computing*, 61(3), 427-446.
- [66] Wang, C., & Parthasarathy, S. (2004, June). Parallel algorithms for mining frequent structural motifs in scientific data. In *Proceedings of the 18th annual international conference on Supercomputing* (pp. 31-40). ACM.
- [67] Meinl, T., Fischer, I., & Philippsen, M. (2005). Parallel Mining for Frequent Fragments on a Shared-Memory Multiprocessor-Results and Java-Obstacles. In *In LWA 2005-Beiträge zur GIWorkshopwoche Lernen*.
- [68] Coatney, M., & Parthasarathy, S. (2003, March). Motifminer: A general toolkit for efficiently identifying common substructures in molecules. In *Bioinformatics and Bioengineering, 2003. Proceedings. Third IEEE Symposium on* (pp. 336-340). IEEE.
- [69] Zhao, Z., Wang, G., Butt, A. R., Khan, M., Kumar, V. A., & Marathe, M. V. (2012, May). Sahad: Subgraph analysis in massive networks using hadoop. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International* (pp. 390-401). IEEE.
- [70] Kang, U., Tsourakakis, C. E., Appel, A. P., Faloutsos, C., & Leskovec, J. (2011). Hadi: Mining radii of large graphs. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 5(2), 8.
- [71] Shiloach, Y., & Vishkin, U. (1982). An  $O(n \log n)$  parallel connectivity algorithm. *Journal of Algorithms*, 3(1), 57-67.
- [72] Awerbuch, B., & Singh, T. (1983, August). New Connectivity and MSF Algorithms for Ultracomputer and PRAM. In *ICPP* (Vol. 83, pp. 175-179).
- [73] Hirschberg, D. S., Chandra, A. K., & Sarwate, D. V. (1979). Computing connected components on parallel computers. *Communications of the ACM*, 22(8), 461-464.
- [74] Alon, N., Dao, P., Hajirasouliha, I., Hormozdiari, F., & Sahinalp, S. C. (2008). Biomolecular network motif counting and discovery by color coding. *Bioinformatics*, 24(13), i241-i249.
- [75] Alon, N., Yuster, R., & Zwick, U. (1995). Color-coding. *Journal of the ACM (JACM)*, 42(4), 844-856.
- [76] Afrati, F. N., Fotakis, D., & Ullman, J. D. (2013, April). Enumerating subgraph instances using map-reduce. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on* (pp. 62-73). IEEE.
- [77] Afrati, F. N., & Ullman, J. D. (2011). Optimizing multiway joins in a map-reduce environment. *Knowledge and Data Engineering, IEEE Transactions on*, 23(9), 1282-1298.

- [78] Xiang, J., Guo, C., & Abounaga, A. (2013, April). Scalable maximum clique computation using mapreduce. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on* (pp. 74-85). IEEE.
- [79] Luo, Y., Guan, J., & Zhou, S. (2011). Towards efficient subgraph search in cloud computing environments. In *Database Systems for Adanced Applications* (pp. 2-13). Springer Berlin Heidelberg.
- [80] Hill, S., Srichandan, B., & Sunderraman, R. (2012, October). An iterative MapReduce approach to frequent subgraph mining in biological datasets. In *Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine* (pp. 661-666). ACM.
- [81] Bhuiyan, M. A., & Hasan, M. A. (2013). MIRAGE: An Iterative MapReduce based FrequentSubgraph Mining Algorithm. *arXiv preprint arXiv:1307.5894*.
- [82] Lin, W., Xiao, X., & Ghinita, G. (2014, March). Large-scale frequent subgraph mining in MapReduce. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on* (pp. 844-855).
- [83] Balachandran, R., Padmanabhan, S., & Chakravarthy, S. (2006). Enhanced DB-subdue: Supporting subtle aspects of graph mining using a relational approach. In *Advances in Knowledge Discovery and Data Mining* (pp. 673-678). Springer Berlin Heidelberg.
- [84] Chakravarthy, S., & Pradhan, S. (2008, January). Db-fsg: An sql-based approach for frequent subgraph mining. In *Database and Expert Systems Applications* (pp. 684-692). Springer Berlin Heidelberg.
- [85] A. Petermann, M. Junghanns, and E. Rahm. DIMSpan - transactional frequent subgraph mining with distributed in-memory dataflow systems. BDCAT '17 Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, 2017.
- [86] C. H. Teixeira et al. Arabesque: a system for distributed graph mining. In Proc. of the 25th Symposium on Operating Systems Principles, pages 425–440. ACM, 2015.
- [87] <https://databricks.com/spark>
- [88] Qiao, F.; Zhang, X.; Li, O.; Ding, Z.; Jia, S.; Wang, H. A parallel approach for frequent subgraph mining in a single large graph using Spark. *Appl. Sci.* 2018, 8, 230.
- [89] <http://www.cs.ucsb.edu/~xyan/dataset.htm>
- [90] Bismita S. Jena, Cynthia Khan, Rajshekhar Sunderraman. SparkFSM: A Highly Scalable Frequent Subgraph Mining Approach using Apache Spark, ICDM 2018, Singapore

[91] Kuramochi, M., & Karypis, G. (2001). Frequent subgraph discovery. In Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on (pp. 313-320).

[92] <https://aws.amazon.com/emr/>

[93] Bismita S. Jena, Cynthia Khan, Rajshekhar Sunderraman. High Performance Frequent Subgraph Mining on Transactional Datasets: A Survey and Performance Comparison (Accepted at Tsinghua University Journals - <https://mc03.manuscriptcentral.com/bdma>)