BIG DATA MINING AND ANALYTICS
ISSN 2096-0654 02/05 pp159–180
Volume 2, Number 3, September 2019
DOI: 10.26599/BDMA.2019.9020006

High Performance Frequent Subgraph Mining on Transaction Datasets: A Survey and Performance Comparison

Bismita S. Jena*, Cynthia Khan, and Rajshekhar Sunderraman

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, 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 survey, 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.

Key words: frequent subgraphs; isomorphism; Spark

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. Due to the rapid growth of the social networking sites and web logs, graphs became very abundant and drew a lot of research attention. 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 or wireless interconnection networks, and program traces from software engineering[2]. Graphs are chosen as a common structure in all these domains as modelling complicated structures via graphs is easy. Mining these graphs to extract knowledge has become the real challenge and Frequent Subgraph Mining (FSM) is one such solution. FSM is divided into two major categories, one category belongs to a dataset consisting of moderate

• Bismita S. Jena, Cynthia Khan, and Rajshekhar Sunderraman are with the Department of Computer Science, Georgia State University, Atlanta, GA 30302, USA. E-mail: [bsrichandan1, ckhan3]@student.gsu.edu; raj@cs.gsu.edu.
∗ To whom correspondence should be addressed.
Manuscript received: 2018-11-13; accepted: 2019-02-15

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[3–9] or parallel computing[10–12] 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[13–18].

Problem Statement: The problem is defined as follows: given a dataset (D) consisting of a set of graphs $G_1, G_2, G_3, G_4, \ldots, G_n$, and a minimum support threshold 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 amount. Support of a subgraph is defined as the number of graphs that contain the subgraph. When we discuss about graphs, the graph isomorphism and subgraph isomorphism are the major aspects that need to be discussed which is known to be an NP-complete problem[19].

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[20]. In bioinformatics domain, frequently occurring patterns are introduced as functional building blocks in transcriptional regulatory networks[21,22]. 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[23]. 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 Object-Oriented approach to Frequent SubGraph Mining (OO-FSG)[24] and MRFSM[25] were the major drawbacks and this provided us insight to apply the distributed in-memory Spark engine.

Our Contribution: The following are our contributions:

(1) We have provided an extensive survey on FSM in this paper;

(2) Since our research is on the same line, we have conducted several different experiments on real life datasets, and

(3) Provided performance comparisons between them using different types of high-performance computing methods.

We categorize our research into two types, the first category[24,25] is disk-based where we used the object-oriented database db4o (http://www.db40.com/) and the Hadoop's MapReduce model[26] (http://hadoop.apache.org/docs/r1.2.1/mapredtutorial.html). The second category[27] 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.

Paper Organization: The paper is organized as follows: Section 2 presents definitions related to FSM and surveys pioneering works in the area of FSM for transactional graphs. It covers memory-based single machine techniques (Apriori-based methods and pattern growth approaches), disk-based techniques (partition-based approach, traditional database approach, and parallel and distributed approach), and distributed in-memory approaches. Section 3 introduces our contribution to FSM utilizing high performance techniques. Section 4 presents the ongoing work and provides concluding remarks.

2 Review of Frequent Subgraph Mining Techniques and Related Work

In this section 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); $E \subseteq V\times$ is a set of edges, where edges 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.

Examples of labeled directed and undirected graphs are shown in Fig. 1. A, B, C, and D are the node labels, and a, b, c, d, and e are the edge labels. 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 $V_g \subseteq g_V$ 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 $G_a = G_b$.

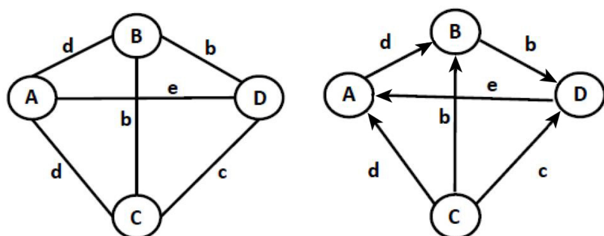


Fig. 1 Undirected (left) and directed (right) labeled graphs.

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. $G = g_1, g_2, g_3, g_4, \ldots, g_n$, where $g_1, g_2 \ldots$ are individual graphs.

Definition 9 (Frequent Subgraph Structure) Given a graph database $D = \{G_1, G_2, G_3, \ldots, G_n\}$, let a subgraph g be contained in $|D_g|$ number of graphs. Then support of g is defined as $\sup(g) = |D_g|/|D|$, 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 in Fig. 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 Fig. 3 as the frequent structures.

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[1,29] introduced the area of frequent pattern mining. Subsequently, several item-set mining methods[30–34], sequential patterns[35–37], and trees[38–40] were developed. With the motivation from apriori algorithm[1], Inokuchi et al.[15] proposed AGM which mines the association rules among the frequently occurring subgraphs. Following the apriori model, PATH[7] and FSG[41] 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

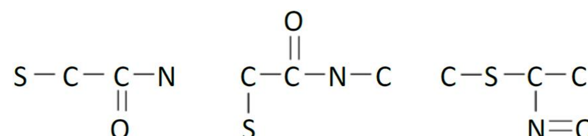


Fig. 2 A sample chemical compound dataset[28].

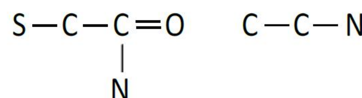


Fig. 3 Frequent subgraphs (left: support 2, right: support 3).

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 describe each category in detail.

2.1 Memory-based single machine techniques

The algorithms developed around early 2000's did not 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 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, ..., 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 1, which is adapted from Ref. [28].

Algorithm 1 works as follows: in the beginning, all

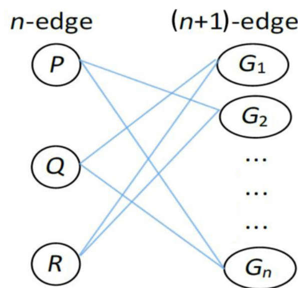


Fig. 4 Apriori-based extension.

Algorithm 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 $(F_k != 0)$
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) $\geqslant$ min_sup and $s \notin F_k + 1$
8: add s to $F_k + 1$
9: $k = K + 1$
10: return

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, PATH[7], AGM[14], FFSM[17], and FSG[41]. AGM[14] 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 has $(k-1)$ vertices and two new vertices from the merged structures. In AGM size, Fig. 5 shows the candidate generation of AGM.

Kuramochi and Karypis[4,41] 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 6 illustrates the candidate generated when two subgraphs with common cores are merged.

Vanetik et al.[7] 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. Figure 7 shows three paths of graph G to the right. The pseudocode of PATH[7] is given in Algorithm 2. Initially, all frequent single edge paths are found. Size-2

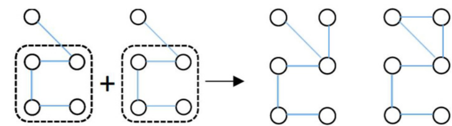


Fig. 5 AGM[28].

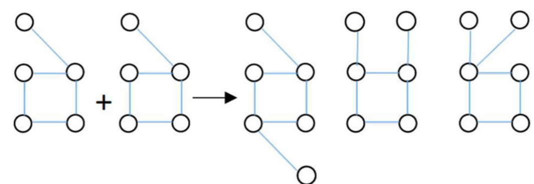


Fig. 6 FSG[28].

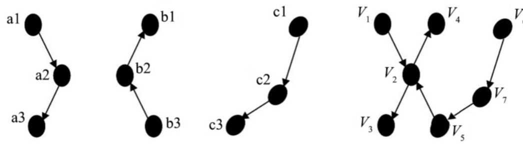


Fig. 7 Graph and 3 edge-disjoint paths.

Algorithm 2 PATH

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

edge-disjoint paths are constructed from size-1 edges, Vanetik et al. proposed a table structure which stores paths as columns and the vertices as the rows. A few paths together build a composition relation.

An example of composition relation for Fig. 7 is given in Table 1. Two composition relations are joined if they have $(n-1)$ paths in common.

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 $C_1(P_1, P_2, P_3, \ldots, P_n)$ and $C_2(P_i, P_j)$, $1 \leqslant i, j \leqslant n$, 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.[17] proposed 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 8 represents two graphs and Fig. 9 represents their canonical adjacency matrices.

Table 1 Composition relation.

Node	P_1	P_2	P_3
V_1	a1	0	0
V_2	a2	b2	0
V_3	a3	0	0
V_4	0	b1	0
V_5	0	b3	c3
V_6	0	0	c1
V_7	0	0	c2

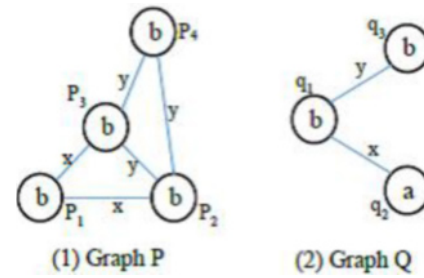


Fig. 8 Example graphs[17].

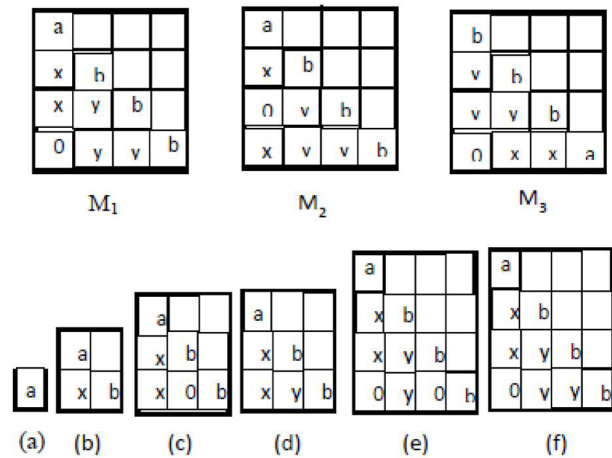


Fig. 9 Canonical adjacency matrices[17].

The paper has discussed several cases for joining and extension. Here, we show one case. Figure 10 shows joining of two CAMs (corresponding 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 Fig. 10 is also of size $m \times m$. FFSM[17] 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

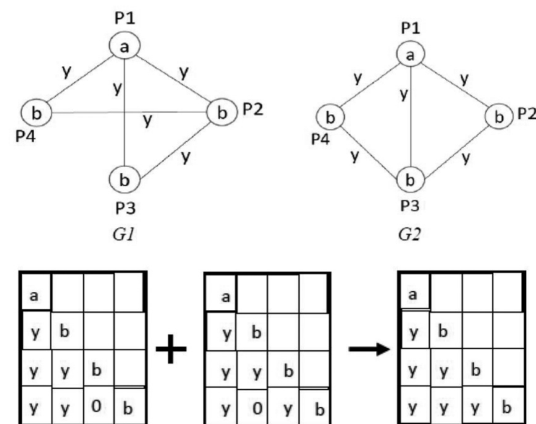


Fig. 10 Example of join[17].

$m_{(1,1)}, m_{(2,1)}, m_{(2,2)}, \ldots, m_{(n,1)}, m_{(n,2)}, \ldots, m_{(n,n)}$ 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 11 shows the pattern growth graph.

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[16,18,23]. 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 3 gives a general idea of pattern growth approach adapted from *Data Mining Concepts and Techniques*[28].

The first algorithm in this category is known as MoFa[23], in which the candidate generation happens by adding a new edge. Extension is restricted to the fragments that actually appear in the database. Embedding is stored for faster support calculation. Second algorithm in this category is popularly known as "gSpan"[16,42]. The authors proposed a DFS lexicographic ordering and minimum DFS code to support DFS search. Figure 12 shows three graphs b, c, and d isomorphic to a, but only one of them have the potential to grow.

Given the DFS codes for different DFS trees, gSpan algorithm chooses the minimum code. From Fig. 12, following the minimum DFS code rule, $a < b < c$. In order to eliminate duplicate generation, gSpan approach adapts a similar methodology like FREQT's rightmost expansion[38] and TreeMinerV's equivalence class extension[39] in frequent tree discovery. Rightmost extension 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[43]. Figure 13 shows the rightmost expansion of graphs.

The last algorithm in this category is GASTON[18]. Nijssen and Kok[18] defined 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

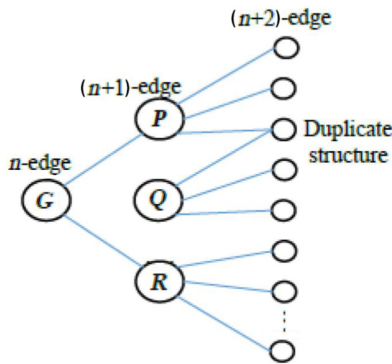


Fig. 11 Pattern growth-based extension.

Algorithm 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 $s \neq e$
4: forall frequent $s \neq e$
5: call Pattern_Growth($s \neq e$, GDB, G)
6: return

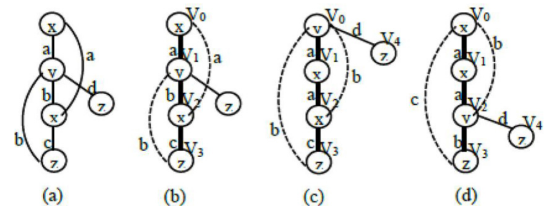


Fig. 12 DFS code[16].

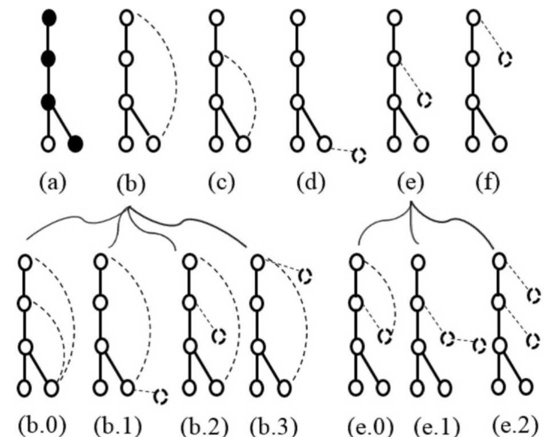


Fig. 13 Rightmost expansion[16].

becomes a cyclic graph when an edge is added between two existing nodes. They proposed 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 14 shows two example graphs in the database and Fig. 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 Fig. 15.

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. 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.

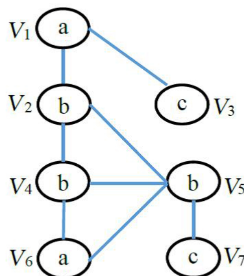


Fig. 14 Database graphs.

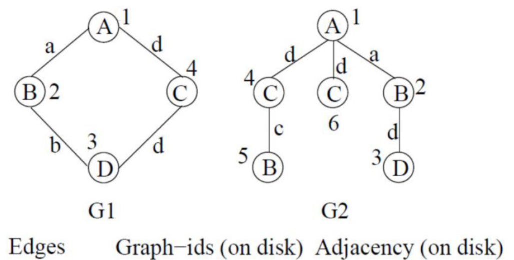
	1	2	3	4	5	6	7	8
1	$(\lambda, G_1, V_2$	$\underline{(\lambda, G_1, V_4)}$	(λ, G_1, V_5)	(λ, G_2, V_1)	(λ, G_2, V_2)			
2	$(1, G_1, V_4)$	$(1, G_1, V_5)$	$\underline{(2, G_1, V_2)}$	$(2, G_1, V_5)$	$(3, G_1, V_2)$	$(3, G_1, V_4)$	$(4, G_2, V_2)$	$(5, G_2, V_1)$
3	$(1, G_1, V_6)$	$\underline{(3, G_1, V_1)}$	$(5, G_1, V_1)$	$(6, G_1, V_6)$	$(7, G_2, V_3)$		G	G
4	$(1, G_1, V_1)$	$\underline{(2, G_1, V_6)}$					G	G
5	$(1, G_1, V_5)$	$\underline{(2, G_1, V_5)}$					G	G
6	1	$\underline{2}$					G	G
7	$\underline{2}$							

Fig. 15 Embedding.

2.2.1 Partition-based approach

A horizontal data partitioning approach on transaction databases was first introduced by Savasere et al.[44] Wang et al.[45] proposed a partition-based approach, ADI-Mine, in which they created an index structure ADI (adjacency index). For each edge, they maintained the graph ids in a linked list. A graph id is entered once per edge irrespective of multiple occurrence of same edge. Figure 16 shows the example of the graph and its adjacency index. They adapted the famous gSpan[16] algorithm methodology for frequent subgraph-mining.

In Ref. [46], 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[18] 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 17 shows the phase1 and phase2 of their procedure. Nguyen et al.[47] proposed to use data partition technique on graphs that



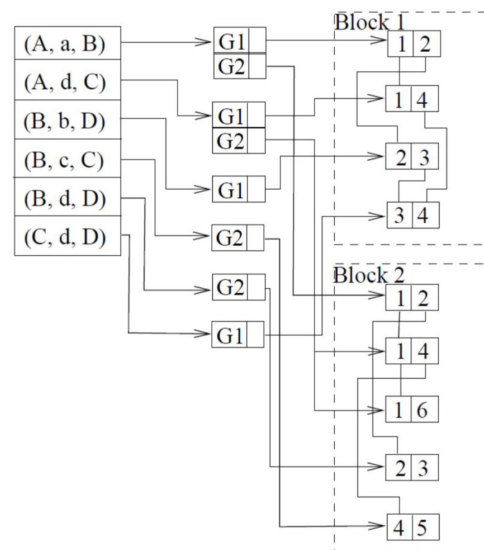


Fig. 16 An ADI structure[45].

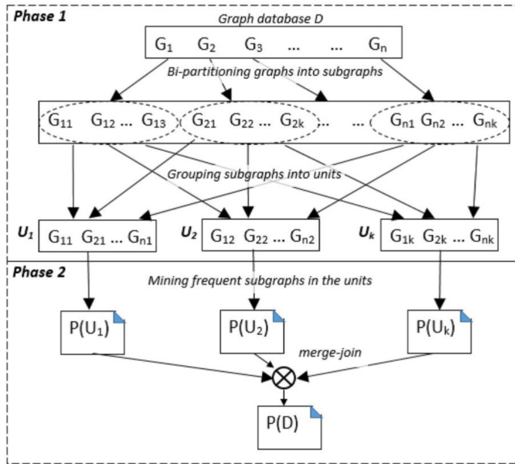


Fig. 17 PartMiner partition method[46].

are an extension of their previous work, which was applied on frequent item sets[48]. In their work[47], K-means algorithm is used to partition the data. Figure 18 shows the general idea behind their partitioning approach. Their algorithm is given below in Algorithm 4.

2.2.2 Traditional database approach

Traditional databases such as relational databases and object-oriented databases became the second choice

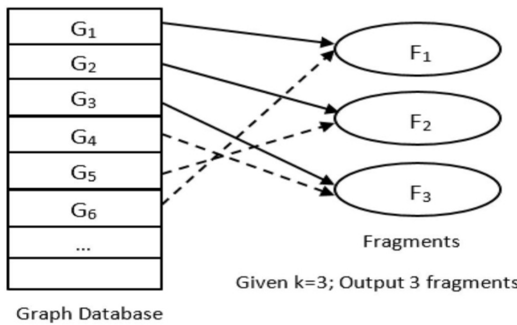


Fig. 18 Data Partition scheme for PartGraphMining[47].

Algorithm 4 PartGraphMining

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

Output: Frequent subgraph set

1: Partition the graph database into k fragments ($G_1, G_2, G_3, \ldots, G_n$) 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 $i = 1, 2, 3, \ldots, k$

3: Compute the union of all $f(G_i)$, add them to L^G

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

for large data storages. DB-subdue[49] is the very first attempt using relational database approach for subgraph mining. DB-subdue implements the idea of SUBDUE[50], which is one of the early frequent subgraph mining algorithms on single graph that detects the best structure using minimum description length principle[51]. 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[49] stores graphs as relations in database. Evaluation of best structures is 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[52] and HDB-Subdue[53] is an improvement over DB-Subdue. They handle cycles in graph and multiple edges between vertices. HDB-Subdue allows unconstrained expansion of substructures. The drawback of unconstrained expansion is that it generates duplicates as the same structure is generated from instances in different orders. 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[50] idea. These implementations surely provided some ideas to apply on transaction graphs.

DB-FSG[54] 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 tables are constructed with corresponding vertex/edge labels, numbers assigned to them, and the graph id that contains them. Figure 19 shows the example graph based on which Table 2 is constructed. Table 2 shows the vertices, their labels, and graph id. Table 3 contains the edges, their labels, and graph id.

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 ($k-1$)

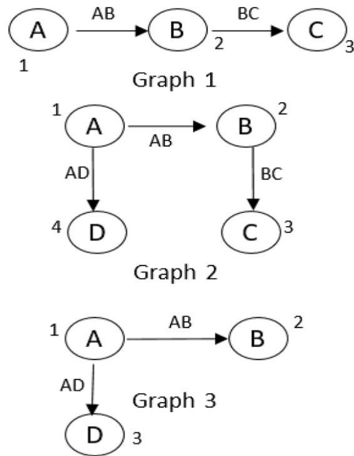


Fig. 19 DB-FSG example graph.

Table 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

Table 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

subgraphs with single-edge subgraphs. 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[54] imposes that the new edge that is added should not have same edge number as in the instance edges. Frequency counting is done based on 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[24] is discussed in Section 3.1.

2.2.3 Parallel and distributed approach

With the advancement of multi-core technologies, Graphic Processing Units (GPUs), and Google's MapReduce model[26], 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 and Zhang[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 Table 4. Item 'a' is represented as 11000 which means 'a' appears in transactions T1 and T2. In Ref. [56], the items are organized in a tri-based structure which is basically the prefix tree. Li[57] presented an inverse tree structure with bitmap representation to find frequent maximal itemset over stream data.

A novel data structure is introduced by Amossen and Pagh[58] called BATMAP, which provides all advantages of bitmap along with space compression for sparse data sets using hash tables. Teodoro et al.[59] used tree-projection based structure. Instead of bitmaps, the authors proposed to store the transactions in a vector. Cheung et al.[60] proposed FDM to mine association rules using distributed approach. They found locally frequent items on each machine and broadcasted them to all machines. Both local and global pruning are applied to have lesser number of candidates at individual sites. Liu et al.[61] proposed a parallel version of FP-Growth[32], a memory-based algorithm on multi-core system. They proposed a cache-conscious frequent pattern array and a lock-free dataset tiling parallelization mechanism. A MapReduce based parallel FP-Growth is proposed in Ref. [62]. 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.[63] proposed MG-FSM, a sequence pattern mining using MapReduce. Their partitioning approach is based on the concept of "projected database".

Table 4 Example transaction/itemsets.

Tid	Item sets
1	a b c d e
2	a b c d
3	b c d
4	b e
5	c d e

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 and Bai[12] 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.[64] proposed a MapReduce-based pattern-finding algorithm, MRPF, for network motif detection from complex networks. Reinhardt and Karypis[11] proposed an algorithm using OpenMP that finds connected edge-disjoint embedding. Wang and Parthasarathy[65] presented parallel algorithm for their previously developed Motif Miner Toolkit[66] that mines structural motifs in a wide range of bio-molecular datasets. SUBDUE[9] system has been improved a lot since it was developed. The parallel version[67] 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, in DP-SUBDUE, each processor evaluates a disjoint set of the input data, and SP-SUBDUE uses a static data partitioning approach. Meinl et al.[68] parallelized the memory-based algorithm, MoFa[23], with a substantial speed-up gain. Kang et al.[10] presents "PEGASUS", an open source graph mining library built using MapReduce framework on Hadoop platform. PEGASUS handles typical mining tasks such as connected component[69–71], diameter of the graph[72], and computing the radius of node. Zhao et al.[73] 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 used the techniques from their paper[77] for computing multiway joins to reduce communication cost. Xiang et al.[78] presented a MapReduce-based scalable and fault-tolerant solution for the maximum clique problem. They used a graph coloring-based partitioning approach which recursively partitions the data into smaller units while maintaining load balance. The maximum cliques of different partitions are computed independently.

Fatta and Berthold[79] used a search tree partitioning strategy, along with dynamic load balancing and a peer-to-peer communication framework for efficient mining. Luo et al.[80] proposed a MapReduce-based "subgraph query search method". The idea is that, given a subgraph, find all graphs containing that particular query sub-graph. Buehrer et al.[81] proposed parallelizing FSG algorithms on CMP architecture. We proposed a MapReduce-based FSG[25] which is covered in our contribution section of the paper. A few more works are published following our implementation on MapReduce. Aridhi et al.[82] proposed a density-based data partitioning approach on MapReduce framework. Bhuiyan and Hasan[83] proposed MIRAGE, a MapReduce-based approach in which they adopted idea from gSpan[16] for right-most extension to prevent duplicate generation and a gSpan style dfs code for counting and isomorphism checks. In Ref. [84], the authors introduced a novel technique to make the distributed embedding exploration more scalable. Lin et al.[85] made use of a memory-based algorithm, GASTON[18], 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. Later, two subsequent sections 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 (http://databricks.com/spark/) 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 Ref. [86] 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 Ref. [87] used Apache Flink, which is similar to Spark but mostly used for real-time processing. In their paper, the 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 approach[25] on directed graphs. Algorithm 5 describes DIM Span's

Algorithm 5 Distributed FSM Dataflow

Input: $G = \{\langle G,\mu^1 \rangle_i\}_{i\subset}, f_{\min}$
1: $F \longleftarrow \phi$
2: $F^k \longleftarrow \phi$
3: repeat
4: $P^k \longleftarrow$ *G.flatmap(report)*
5: $\phi^k \longleftarrow P^k$*.combine(count)*
6: $\phi^k \longleftarrow \phi^k_w$*.reduce(sum)*
7: $F^k \longleftarrow P^k$*.filter*$(\phi^k(P) \geqslant f_{\min})$
8: *broadcast*(F^k)
9: $G \longleftarrow$ *G.map(patternGrowth)*
10: $G \longleftarrow$ *G.filter*$(\exists P : | \, | \geqslant 0)$
11: $F^k \longleftarrow F \cup F^k$
12: until $F^k \neq \phi$
13: return F

distributed dataflow.

3 Our Contribution

We started our research journey in the FSM area following a research work[54] where authors utilized the RDBMS potential to overcome the single machine main memory bottleneck. In the following sub-sections, we introduce our complete work based on several types of high-performance computing techniques. As per our findings, all three categories are first ever introduction to this area of research. We would like to categorize our work in Refs. [25, 27] under the high-performance category as per the paper's title. We would like to begin with our initial work[24].

3.1 OO-FSG[24]

We chose the db4o (http://www.db4o.com/), 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 die when the program stops.

3.1.1 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. The OO-FSG algorithm is given in Algorithm 6.

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 that fail to satisfy the minimum support value from the SingleEdge class. Step 4 is the looping condition, looping occurs from Step 4(a) through Step 4(e) until size n, which is Step 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. In this context, subgraphs mean only the edge labels and vertex labels not the numbers given to the nodes and edges. Step 4(d) and Step 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.

3.1.2 Performance comparison of DB-FSG vs. OO-FSG

The experiments were conducted on a Linux machine with 2 GB memory. The OO-FSG algorithm used

Algorithm 6 OO-FSG

Input: Graph database GDB, Minimum support, Number of partitions (k)
Output: Frequent subgraph set
1: construct SingleEdge class by joining Vertex and Edge class.
2: select distinct single edges and store the subgraphs which satisfies min_sup in Subgraph−1 class.
3: remove the edges with count less than the min_sup from SingleEdge.
4: repeat steps a through e until a candidate subgraph of size−N with 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 min_sup and store it in Subgraph−N class.
(e) remove the edges with count less than min_sup from (N)Edge class.
5: end loop.

*(N)Edge: represents the TwoEdge, ThreeEdge, FourEdge and FiveEdge classes etc.

Java. Table 5 shows the performance metrics for both approaches.

3.2 An iterative MapReduce-based approach to frequent subgraph mining[25]

MapReduce framework by Google[26] motivated us to implement the frequent subgraph mining method on graph databases. There are a few researches that have applied MapReduce for graph mining, which provided us with some motivation that we can apply the framework on frequent subgraph mining for transactional graphs. Finding frequent substructures from transaction databases in particular has a typical pattern such that, 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 could be done easily 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 that used to be a bottleneck in our previous traditional database approach. The next subsections define our approach.

3.2.1 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.

Table 5 Example transaction/itemsets.

Dataset size ($\times 10^3$)	Min_sup (%)	DB−FGS	OO−FSG
50	1	357	353
100	1	1349	731
100	3	1220	656
100	5	1061	563
100	7	827	484
200	1	2439	1331
200	3	2002	1206
200	5	1717	1117
200	7	1622	1030
300	1	5887	2221
300	3	5394	2141
300	5	5137	2019
300	7	4164	1863
400	1	9502	2879
400	3	8228	2457
400	5	7156	2426
400	7	6962	2313

3.2.1.1 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.

3.2.1.2 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 different nodes with the same labels. The results of this step are written out to the Hadoop File System.

Input key: graph id;

Input value: list of subgraphs of size $(k-1)$ encoded with graph id;

Output key: encoded subgraph of size k and graph id;

Output value: none.

3.2.1.3 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.

3.2.1.4 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 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. In the end, if the count agrees with the given user defined support, it is written to the Hadoop File System for the next iteration, and otherwise it is ignored or 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 value: list of corresponding node ids and graph ids;

Output key: encoded subgraph and graph id;

Output value: corresponding node ids and graph id.

3.2.2 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 a subgraph meets the use-defined support. The algorithm starts with single edges and runs until there are no longer any frequent subgraphs constructed. Algorithms 7 and 8 highlight the tasks of A_k. Algorithms 9 and 10 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 7 Map A_k

```
Input: (offset, subgraph)
  parse subgraph for graph id
  EMIT: (graph id, subgraph)
```

Algorithm 8 Reduce A_k

```
Input: (graph id, subgraphs s1, s2, s3, ...)
  Edges ← ϕ
  newSubgraphs ← ϕ
  for all s ∈ subgraphs do
    Retrieve all edges from s and add to edges
  end for
  for all s ∈ subgraphs do
    Construct k-sized subgraphs using Edges and add to newSubgraphs
  end for
  for all s ∈ newSubgraphs do
    EMIT: (encoding for subgraph, empty text)
  end for
```

Algorithm 9 Map B_k

```
Input: (offset, encoded subgraph)
  parse encoded subgraph for label-only subgraph
  EMIT: (label-only subgraph, subgraph)
```

Algorithm 10 Reduce B_k

```
Input: (label-only subgraph, subgraphs s1, s2, s3, ...)
  GraphIDs ← ϕ
  count ← 0
  for all s ∈ subgraphs do
    if s.graphid ∉ GraphIDs
      count ← count + 1
      GraphIDs ← GraphIDs ∪ s.graphid
    end if
  end for
  if count ⩾ user support then
    for all s ∈ subgraphs do
      EMIT: (subgraph, empty text)
    end for
```

3.2.2.1 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 it is still ambiguous. 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.

3.2.2.2 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 Fig. 20. 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 both Ai and Bi 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."

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 Fig. 21.

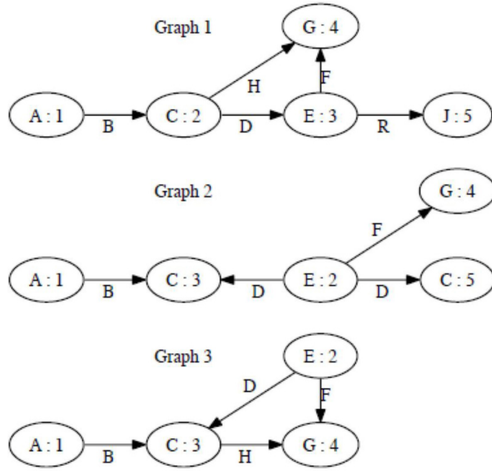


Fig. 20 Example graphs.

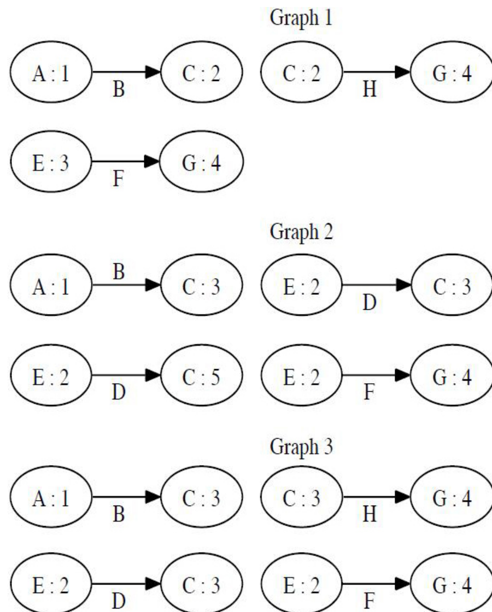


Fig. 21 Single-edge subgraphs that meet support.

Step A2: The worker for A2 will read input from the file system corresponding to the job of B1. The output strings are as 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 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 of Fig. 20, we have (E^1:D-C^1,D-C^2)_(2:3,5)_2. 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.

Step B2: The worker for B2 will read input from the file system 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 Fig. 22.

Here we are showing only subgraph generation until three edges. The final output from Fig. 20 with support threshold 50% is displayed in Fig. 23.

3.2.3 Experimental details

Experiments were conducted on synthetic datasets, obtained from http://www.cse.ust.hk/graphgen/ and on the biological datasets obtained from http://www.cs.ucsb.edu/xyan/dataset.htm. The real-life datasets contain data extracted from PubChem website which contains the bioassay records for anti-cancer screen tests with different cancer cell lines. Table 6 shows the performance analysis.

3.3 SPARKFSM: A highly scalable frequent subgraph mining approach using Apache Spark[27]

In our recent work, we have handled the undirected transaction graphs utilizing the power of Spark engine. After our MapReduce implementation[25], many

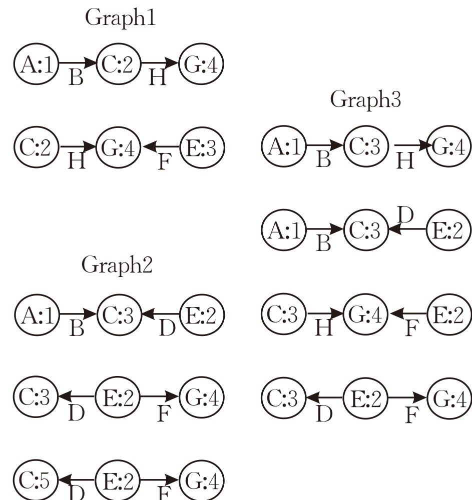


Fig. 22 Double edge subgraphs that meet support.

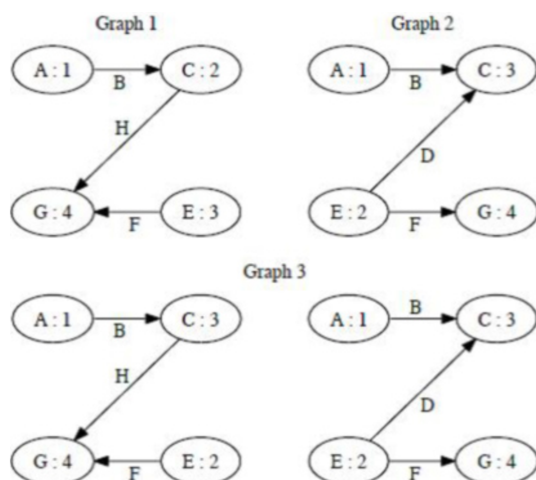


Fig. 23 Triple-edge subgraphs that meet support.

Table 6 MRFSM[25] 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	833	587	1092	683
MOLT-4	922	556	1279	815
NCI-H23	815	516	1537	889
OVCAR-8	861	552	1257	844
P388	743	483	976	683
PC-3	857	546	1150	752
SF-295	936	528	1217	817
SN12C	813	502	1474	883
SW-620	959	568	1454	898
UACC257	836	536	1333	883
Yeast	710	607	1282	812

authors tried to handle directed graphs differently, but none experimented on the undirected graphs. There is a big semantic difference between directed and undirected graphs. When consider airline flight information graphs, those are directed and isomorphism detection is different in them than the chemical compound structures. The biological datasets (http://www.cs.ucsb.edu/xyan/dataset.htm) we experimented on are chemical compound structures. Isomorphism plays a little different role here, for example, water (H_2O), 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 subgraps. 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. Similar is the case with NH_3, a compound consisting of nitrogen and three hydrogen atoms. We do prune the structures in the subsequent iterations. We provide here a comparison of both types of graphs with three sample graphs and show how the structure retention is essential in the chemical compounds. Figure 24 shows the undirected sample graphs. Figures 25 and 26 show the retained and pruned structures for undirected graphs from Fig. 24.

The algorithms are given below for undirected and directed graphs. In Algorithm 11 for undirected, the major difference is the unique code captured for each subgraph. The difference between the two is Step 4 of directed graphs in Algorithm 12 where multiple scenarios are caught due to the direction constraint.

The sample directed graphs are shown in Fig. 27. We observed a very interesting pattern from both the directed and undirected graphs shown in Fig. 24 and Fig. 27, even though the undirected graphs resulted in multiple intermediate subgraphs due to the isomorphic structure retention, the final 5-edge FSGs are same for both as shown in Fig. 28. The left sugraphs "a" belong to graphs G_1 and G_3, and the right ones "b" belong to G_2, G_3.

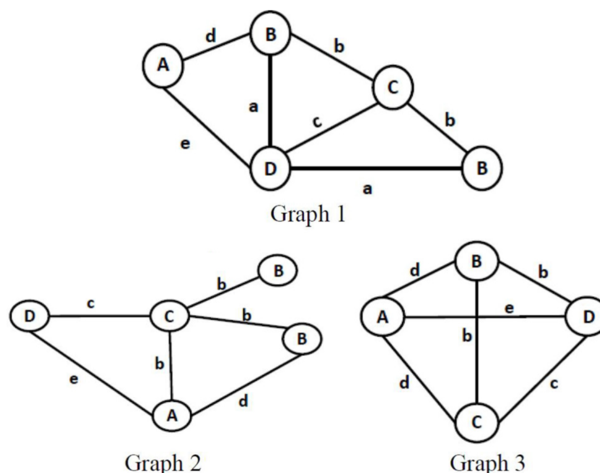


Fig. 24 Undirected graphs.

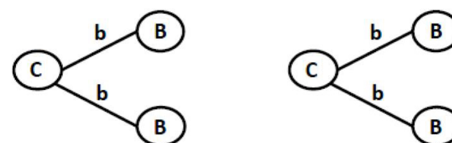


Fig. 25 Structures retained (G_1, G_2).

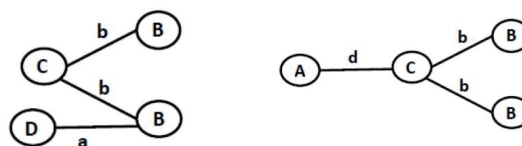


Fig. 26 Pruned subgraphs (G_1, G_2).

Algorithm 11 Undirected Graphs

Input: Graph (G_1), Frequency (f)
Output: Qualified Subgraph Edge list

Process:
1) G_1*.map* ⇒ *Load RDD_1*
2) *RDD1.filter(count* ⩾ f*)* ⇒ *RDD_1*
3) *RDD_1.map* ⇒ *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)* ⇒ $k+1$*_EdgeRDD*
• Unidirection − join *RDDA.secondNode === RDDB.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* ⩾ f*)* ⇒ $k+1$*EdgeEDD*
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-egde RDDs as RDD_A and SingleEdgeRDD as RDD_B for subsequent steps.

Algorithm 12 Directed Graphs

Input: *Graph (*G_1*), Frequency (f)*
Output: Qualified Subgraph Edge list

Process:
1) G_1*.map* ⇒ *Load RDD_1*
2) *RDD_1.filter(count* ⩾ *f)* ⇒ *RDD_1*
3) *RDD_1.filter(duplicate edges)* ⇒ *SingleEdgeRDD*
4) *kEdgeRDD.join(SingleEdgeRDD)* ⇒ *k+1_EdgeRDD*
• *Unidirection – join* RDD_A*.secondNode ===* RDD_B*.firstNode*
• *Converge – join* RDD_A*.secondNode ===* RDD_B*.secondNode*
• *Diverge – join* RDD_A*.secondNode ===* RDD_B*.firstNode*
• *Filter (*RDD_A*.graphID ===* RDD_B*.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 edgepatttern).count()*
6) *k+1_EdgeRDD.filter(count* ⩾ *f)* ⇒ *k+1EdgeEDD*
7) Repeat steps 4 – 6 for $k+1$ *EdgeRDD*
8) Repeat step 7 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-egde RDDs as RDD_A and SingleEdgeRDD as RDD_B for subsequent steps.

All our tests were conducted on AWS EMR with 1 master node and 2 slave nodes with m4

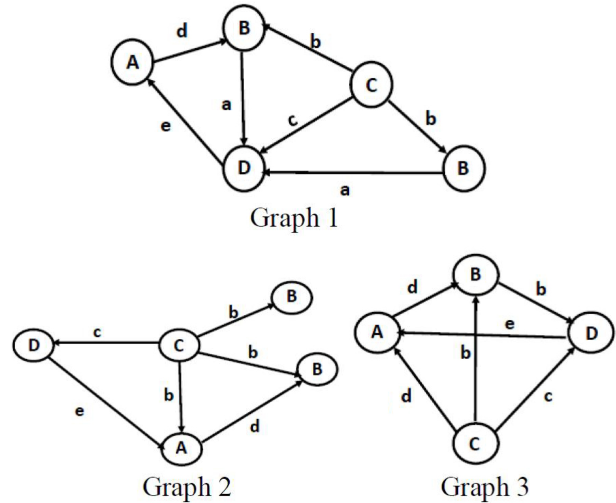


Fig. 27 Directed graphs.

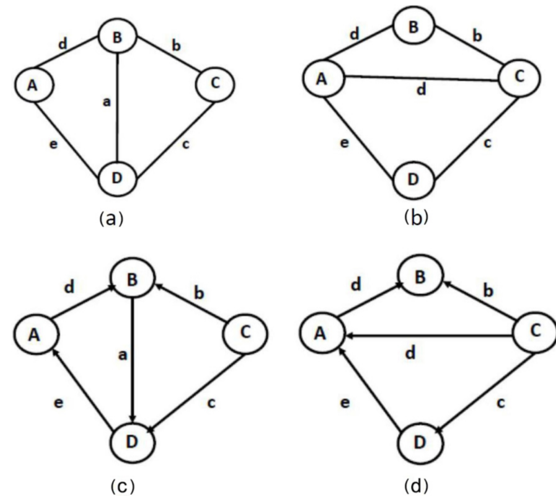


Fig. 28 Five-edge subgraphs from undirected and directed graphs.

large configuration. We used Spark 2.3 for all our experiments. Both directed and undirected jobs ran in parallel on the same cluster and this was an evaluation criterion for the experiments.

Dataset Preparation: We used the chemical compound dataset retrieved from the repository (http://www.cs.ucsb.edu/xyan/dataset.htm). 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 is. The isomorphic

subgraphs are eliminated during the very first step, as the graph numbers remain same across the larger set. We wrote a script to generate the larger graphs like OVCAR8I and OVCAR8HI. The script 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[87] 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[27] would not differ much. Matching the MRFSM[25] computation time with the SparkFSM would not be fair as the technologies are different and Spark is in-memory computation. Table 7 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.

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 1×10^6 limit, the number of new edges 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 8 indicates our previous evaluation MRFSM[25] 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.

DIMSpan vs. SparkFSM: We used DIMSpan[87] as one of our evaluation standards, but DIMSpan has focused on the multi directed graphs as opposed to our SparkFSM[27], which is more focused on undirected graphs. From their Data Sets in 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.

Table 9 shows our 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

Table 7 SparkFSM[27] performance analysis on biological graphs (time in seconds, threshold frequency: 50%).

Graph	Size (MB)	Number of graphs	Number of edges	Undir	Directed
MCF7A	1.3	2293	18	5	30
MCF7HA	2.3	2293	31	34	49
MCF7I	12.0	25 475	36	40	74
MCF7HI	20.0	25 475	59	91	77
MOLT4A	1.7	3139	43	33	55
MOLT4HA	3.0	3139	60	76	37
MOLT4I	17.0	36 624	36	84	63
MOLT4HI	29.0	36 624	59	74	75
NCIH23I	18.0	38 295	36	78	65
NCIH23HI	31.0	38 295	59	73	86
OVCAR8I	18.0	38 436	36	56	56
OVCAR8HI	20.0	38 436	48	45	33

Table 8 MRFSM[25] 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

Table 9 SparkFSM[27] performance analysis on large undirected datasets (time in minutes).

Graph	Support (%)	Number of graphs	Time (min)
OVCAR8HI	75	153 180	2.20
OVCAR8HI	90	153 180	0.70
OVCAR8HI	75	306 366	4.00
OVCAR8HI	90	306 366	0.96
OVCAR8HI	75	1 225 465	13.00
OVCAR8HI	90	1 225 465	2.00
OVCAR8HI	75	2 450 931	26.00
OVCAR8HI	90	2 450 931	4.00

sizes range from 124 MB to 2.1 GB.

4 Concluding Remarks

In this paper, we have tried our best to provide extensive survey on the frequent subgraph mining on transactional graphs. We hope the readers will get a good idea on the concept starting with its inception and on the status as of now. Also, for the first time we have introduced the undirected transaction graphs mining using the high-performance technology Spark. With the rapid progress in big data technologies, many issues are easily handled. We provide some analysis based on our experience while experimenting different approaches on transactional FSM.

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 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[54], 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[54].

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, has many advantages over any verbose programming and being the language base for Spark, and 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.

As part of our ongoing research on FSM, we are exploring on utilizing the high-performance computing on the single large graphs such as social network, protein-protein interaction graphs, and neural network graphs.

References

[1] R. Agrawal and R. Srikant, Fast algorithms for mining association rules, in *Proc. 20th Int. Conf. Very Large Data Bases*, Santiago, Chile, 1994, pp. 487–499.

[2] C. Liu, X. F. Yan, L. Fei, J. W. Han, and S. P. Midkiff, SOBER: Statistical model-based bug localization, *ACM SIGSOFT Soft. Eng. Notes*, vol. 30, no. 5, pp. 286–295, 2005.

[3] X. Jiang, H. Xiong, C. Wang, and A. H. Tan, Mining globally distributed frequent subgraphs in a single labeled graph, *Data Know. Eng.*, vol. 68, no. 10, pp. 1034–1058, 2009.

[4] M. Kuramochi and G. Karypis, Finding frequent patterns in a large sparse graph, *Data Min. Know. Dis.*, vol. 11, no. 3, pp. 243–271, 2005.

[5] M. Kuramochi and G. Karypis, Grew—A scalable frequent subgraph discovery algorithm, in *Proc. Fourth IEEE Int. Conf. Data Mining*, Brighton, UK, 2004, pp. 439–442.

[6] S. Ghazizadeh and S. S. Chawathe, SEuS: Structure extraction using summaries, in *Discovery Science*, S. Lange, K. Satoh, and C. H. Smith, eds. Springer, 2002, pp. 71–85.

[7] N. Vanetik, E. Gudes, and S. E. Shimony, Computing frequent graph patterns from semistructured data, in *Proc. 2002 IEEE Int. Conf. Data Mining*, Maebashi City, Japan, 2002, pp. 458–465.

[8] K. Yoshida, H. Motoda, and N. Indurkhya, Graph-based induction as a unified learning framework, *Appl. Intell.*, vol. 4, no. 3, pp. 297–316, 1994.

[9] L. B. Holder, D. J. Cook, and S. Djoko, Substucture discovery in the SUBDUE system, in *Proc. Workshop on Knowledge Discovery in Databases*, 1994, pp. 169–180.

[10] U. Kang, C. E. Tsourakakis, and C. Faloutsos, PEGASUS: A peta-scale graph mining system implementation and observations, in *Proc. Ninth IEEE Int. Conf. Data Mining*, Miami, FL, USA, 2009, pp. 229–238.

[11] S. P. Reinhardt and G. Karypis, A multi-level parallel implementation of a program for finding frequent patterns in a large sparse graph, in *Proc. 12th Int. Workshop on High-Level Parallel Programming Models and Supportive Environments*, Rome, Italy, 2007, pp. 1–8.

[12] B. Wu and Y. L. Bai, An efficient distributed subgraph mining algorithm in extreme large graphs, in *Artificial Intelligence and Computational Intelligence*, F. L. Wang, H. Deng, Y. Gao, and J. Lei, eds. Springer, 2010, pp. 107–115.

[13] L. Dehaspe, H. Toivonen, and R. D. King, Finding frequent substructures in chemical compounds, in *Proc. Fourth Int. Conf. Knowledge Discovery and Data Mining*, 1998.

[14] A. Inokuchi, T. Washio, and H. Motoda, An apriori-based algorithm for mining frequent substructures from graph data, in *Principles of Data Mining and Knowledge Discovery*, D. A. Zighed, J. Komorowski, and J. Zytkow, eds. Springer, 2000, pp. 13–23.

[15] A. Inokuchi, T. Washio, K. Nishimura, and H. Motoda, *A fast algorithm for mining frequent connected subgraphs*, IBM Research report, RT0448, 2002.

[16] X. F. Yan and J. W. Han, gSpan: Graph-based substructure pattern mining, in *Proc. 2002 IEEE Int. Conf. Data Mining*, Maebashi City, Japan, 2002, pp. 721–724.

[17] J. Huan, W. Wang, and J. Prins, Efficient mining of frequent subgraphs in the presence of isomorphism, in *Proc. Third IEEE Int. Conf. Data Mining*, Melbourne, FL, USA, 2003, pp. 549–552.

[18] S. Nijssen and J. N. Kok, A quickstart in frequent structure mining can make a difference, in *Proc. Tenth ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, New York, NY, USA, 2004, pp. 647–652.

[19] S. Fortin, The graph isomorphism problem, Technical Report 96–20, University of Alberta, Edomonton, Alberta, Canada, 1996.

[20] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu, Mining behavior graphs for "Backtrace" of noncrashing bugs, in *Proc. 5th SIAM Int. Conf. on Data Mining*, 2005, pp. 286–297.

[21] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, Network motifs: Simple building blocks of complex networks, *Science*, vol. 298, no. 5594, pp. 824–827, 2002.

[22] R. Milo, N. Kashtan, S. Itzkovitz, M. E. J. Newman, and U. Alon, On the uniform generation of random graphs with prescribed degree sequences, arXiv preprint arXiv:cond-mat/0312028v2, 2003.

[23] C. Borgelt and M. R. Berthold, Mining molecular fragments: Finding relevant substructures of molecules, in *Proc. 2002 IEEE Int. Conf. Data Mining*, Maebashi City, Japan, 2002, pp. 51–58.

[24] B. Srichandan and R. Sunderraman, Oo-fsg: An object-oriented approach to mine frequent subgraphs, in *Proc. Ninth Australasian Data Mining Conf*, Darlinghurst, Australia, 2011, pp. 221–228.

[25] S. Hill, B. Srichandan, and R. Sunderraman, An iterative MapReduce approach to frequent subgraph mining in biological datasets, in *Proc. ACM Conf. Bioinformatics, Computational Biology and Biomedicine*, New York, NY, USA, 2012, pp. 661–666.

[26] J. Dean and S. Ghemawat, MapReduce: Simplified data processing on large clusters, *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[27] B. Jena, C. Khan, and R. Sunderraman, SparkFSM: A highly scalable frequent subgraph mining approach using apache spark, in *Proc. ICDM Workshops 2018*, 2018.

[28] J. W. Han and M. Kamber, *Data Mining: Concepts and Techniques, 3rd edition*. San Francisco, CA, USA: Morgan Kaufmann, 2012.

[29] R. Agrawal, T. Imieliński, and A. Swami, Mining association rules between sets of items in large databases, *ACM Sigmod Record Homep.*, vol. 22, no. 2, pp. 207–216, 1993.

[30] D. Burdick, M. Calimlim, and J. Gehrke, MAFIA: A maximal frequent itemset algorithm for transactional databases, in *Proc. 17th Int. Conf. Data Engineering*, Heidelberg, Germany, 2001, pp. 443–452.

[31] R. J. Bayardo Jr, Efficiently mining long patterns from databases, *ACM Sigmod Record*, vol. 27, no. 2, pp. 85–93, 1998.

[32] J. W. Han, J. Pei, and Y. W. Yin, Mining frequent patterns without candidate generation, *ACM SIGMOD Record*, vol. 29, no. 2, pp. 1–12, 2000.

[33] M. J. Zaki and C. J. Hsiao, CHARM: An efficient algorithm for closed itemset mining, in *Proc. 2nd SIAM Int. Conf. Data Mining*, 2002, pp. 457–473.

[34] M. J. Zaki and K. Gouda, Fast vertical mining using diffsets, in *Proc. Ninth ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, New York, NY, USA, 2003, pp. 326–335.

[35] J. Pei, J. W. Han, B. Mortazavi-Asl, H. Pinto, Q. M. Chen, U. Dayal, and M. C. Hsu, PrefixSpan,: Mining sequential patterns efficiently by prefix-projected pattern growth, in *Proc. 17th Int. Conf. Data Engineering*, Heidelberg, Germany, 2001.

[36] H. Mannila, H. Toivonen, and A. I. Verkamo, Discovery of frequent episodes in event sequences, *Data Min. Know. Disc.*, vol. 1, no. 3, pp. 259–289, 1997.

[37] R. Agrawal and R. Srikant, Mining sequential patterns, in *Proc. Eleventh Int. Conf. Data Engineering*, Washington, DC, USA, 1995, pp. 3–14.

[38] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa, Efficient substructure discovery from large semi-structured data, in *Proc. 2002 SIAM Int. Conf. Data Mining*, Arlington, VA, USA, 2002.

[39] M. J. Zaki, Efficiently mining frequent trees in a forest, in *Proc. Eighth ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, 2002, pp. 71–80.

[40] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad, A tree projection algorithm for generation of frequent item sets, *J. Parall. Distrib. Comput.*, vol. 61, no. 3, pp. 350–371, 2001.

[41] M. Kuramochi and G. Karypis, Frequent subgraph discovery, in *Proc. IEEE Int. Conf. Data Mining*, San Jose, CA, USA, 2001, pp. 313–320.

[42] X. Yan, Mining, indexing and similarity search in large graph data sets, PhD dissertation, University of Illinois at Urbana-Champaign, IL, USA, 2006.

[43] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. Boston, MA, USA: MIT Press and McGraw-Hill, 2001.

[44] A. Savasere, E. Omiecinski, and S. B. Navathe, An efficient algorithm for mining association rules in large databases, in *Proc. 21th Int. Conf. Very Large Data Bases*, San Francisco, CA, USA, 1995, pp. 432–444.

[45] C. Wang, W. Wang, J. Pei, Y. T. Zhu, and B. L. Shi, Scalable mining of large disk-based graph databases, in *Proc. Tenth ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining*, New York, NY, USA, 2004, pp. 316–325.

[46] J. M. Wang, W. Hsu, M. L. Lee, and C. Sheng, A partition-based approach to graph mining, in *Proc. 22nd Int. Conf. Data Engineering*, Atlanta, GA, USA, 2006, p. 74.

[47] S. N. Nguyen, M. E. Orlowska, and X. Li, Graph mining based on a data partitioning approach, in *Proc. Nineteenth Conf. Australasian Database*, Darlinghurst, Australia, 2007, pp. 31–37.

[48] S. N. Nguyen and M. E. Orlowska, Improvements in the data partitioning approach for frequent itemsets mining, in *Knowledge Discovery in Databases: PKDD 2005*, A. M. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, eds. Springer, 2005, pp. 625–633.

[49] S. Chakravarthy, R. Beera, and R. Balachandran, DB-Subdue: Database approach to graph mining, in *Advances in Knowledge Discovery and Data Mining*, H. Dai, R. Srikant, and C. Zhang, eds. Springer, 2004, pp. 341–350.

[50] D. J. Cook and L. B. Holder, Graph-based data mining, *IEEE Intell. Syst. Their Appl.*, vol. 15, no. 2, pp. 32–41, 2000.

[51] J. Rissanen, *Stochastic Complexity in Statistical Inquiry*. Teaneck, NJ, USA: World Scientific Publishing Co., Inc, 1989.

[52] R. Balachandran, S. Padmanabhan, and S. Chakravarthy, Enhanced DB-subdue: Supporting subtle aspects of graph mining using a relational approach, in *Advances in Knowledge Discovery and Data Mining*, W. K. Ng, M. Kitsuregawa, J. Li, and K. Chang, eds. Springer, 2006, pp. 673–678.

[53] S. Padmanabhan and S. Chakravarthy, HDB-subdue: A scalable approach to graph mining, in *Data Warehousing and Knowledge Discovery*, T. B. Pedersen, M. K. Mohania, and A. M. Tjoa, eds. Springer, 2009, pp. 325–338.

[54] S. Chakravarthy and S. Pradhan, DB-FSG: An SQL-based approach for frequent subgraph mining, in *Database and Expert Systems Applications*, S. S. Bhowmick, J. Küng, and R. Wagner, eds. Springer, 2008, pp. 684–692.

[55] H. F. Li and N. Zhang, Mining maximal frequent itemsets on graphics processors, in *Proc. 2010 Seventh Int. Conf. Fuzzy Systems and Knowledge Discovery*, Yantai, China, 2010, pp. 1461–1464.

[56] W. B. Fang, M. Lu, X. Y. Xiao, B. S. He, and Q. Luo, Frequent itemset mining on graphics processors, in *Proc. Fifth Int. Workshop on Data Management on New Hardware*, New York, NY, USA, 2009, pp. 34–42.

[57] H. F. Li, A GPU-based maximal frequent itemsets mining algorithm over stream, in *Proc. 2010 Int. Conf. Computer and Communication Technologies in Agriculture Engineerin*g, Chengdu, China, 2010, pp. 289–292.

[58] R. R. Amossen and R. Pagh, A new data layout for set intersection on GPUs, in *Proc. 2011 IEEE Int. Parallel & Distributed Processing Symp.*, Anchorage, AK, USA, 2011, pp. 698–708.

[59] G. Teodoro, N. Mariano, W. Meira Jr, and R. Ferreira, Tree projection-based frequent itemset mining on multicore CPUs and GPUs, in *Proc. 2010 22nd Int. Symp. Computer Architecture and High Performance Computing*, Petropolis, Brazil, 2010, pp. 47–54.

[60] D. W. Cheung, J. W. Han, V. T. Ng, A. W. Fu, and Y. J. Fu, A fast distributed algorithm for mining association rules, in *Proc. Fourth Int. Conf. Parallel and Distributed Information Systems*, Miami Beach, FL, USA, 1996, pp. 31–42.

[61] L. Liu, E. Li, Y. M. Zhang, and Z. Z. Tang, Optimization of frequent itemset mining on multiple-core processor, in *Proc. 33rd Int. Conf. Very Large Data Bases*, Vienna, Austria, 2007, pp. 1275–1285.

[62] H. Y. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, PFP: Parallel FP-growth for query recommendation, in *Proc. 2008 ACM Conf. Recommender Systems*, New York, NY, USA, 2008, pp. 107–114.

[63] I. Miliaraki, K. Berberich, R. Gemulla, and S. Zoupanos, Mind the gap: Large-scale frequent sequence mining, in *Proc. 2013 ACM SIGMOD Int. Conf. Management of Data*, New York, NY, USA, 2013, pp. 797–808.

[64] Y. Liu, X. H. Jiang, H. J. Chen, J. Ma, and X. Y. Zhang, MapReduce-based pattern finding algorithm applied in motif detection for prescription compatibility network, in *Advanced Parallel Processing Technologies*, Y. Dou, R. Gruber, and J. M. Joller, eds. Springer, 2009, pp. 341–355.

[65] C. Wang and S. Parthasarathy, Parallel algorithms for mining frequent structural motifs in scientific data, in *Proc. 18th Ann. Int. Conf. Supercomputing*, New York, NY, USA, 2004, pp. 31–40.

[66] M. Coatney and S. Parthasarathy, Motifminer: A general toolkit for efficiently identifying common substructures in molecules, in *Proc. Third IEEE Symp. Bioinformatics and Bioengineering*, Bethesda, MD, USA, 2003, pp. 336–340.

[67] D. J. Cook, L. B. Holder, G. Galal, and R. Maglothin, Approaches to parallel graph-based knowledge discovery, *J. Parall. Distrib. Comput.*, vol. 61, no. 3, pp. 427–446, 2001.

[68] T. Meinl, I. Fischer, and M. Philippsen, Parallel mining for frequent fragments on a shared-memory multiprocessor-results and java-obstacles, in *Proc. Lernen, Wissensentdeckung und Adaptivität*, Saarbrücken, Germany, 2005.

[69] Y. Shiloach and U. Vishkin, An $O(\log n)$ parallel connectivity algorithm, *J. Algorithms*, vol. 3, no. 1, pp. 57–67, 1982.

[70] B. Awerbuch and T. Singh, New connectivity and MSF algorithms for ultracomputer and PRAM, *IEEE Transcactions on Computers*, vol. 36, no. 10, pp. 1258–1263, 1987.

[71] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate, Computing connected components on parallel computers, *Commun. ACM*, vol. 22, no. 8, pp. 461–464, 1979.

[72] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec, Hadi: Mining radii of large graphs, *ACM Trans. Know. Discovery Data*, vol. 5, no. 2, p. 8, 2011.

[73] Z. Zhao, G. Y. Wang, A. R. Butt, M. Khan, V. A. Kumar, and M. V. Marathe, SAHAD: Subgraph analysis in massive networks using Hadoop, in *Proc. 2012 IEEE 26th Int. Parallel and Distributed Processing Symp.*, Shanghai, China, 2012, pp. 390–401.

[74] N. Alon, P. Dao, I. Hajirasouliha, F. Hormozdiari, and S. C. Sahinalp, Biomolecular network motif counting and discovery by color coding, *Bioinformatics*, vol. 24, no. 13, pp. i241–i249, 2008.

[75] N. Alon, R. Yuster, and U. Zwick, Color-coding, *J. ACM*, vol. 42, no. 4, pp. 844–856, 1995.

[76] F. N. Afrati, D. Fotakis, and J. D. Ullman, Enumerating subgraph instances using map-reduce, in *Proc. 2013 IEEE 29th Int. Conf. Data Engineering*, Brisbane, Australia, 2013, pp. 62–73.

[77] F. N. Afrati and J. D. Ullman, Optimizing multiway joins in a map-reduce environment, *IEEE Trans. Know. Data Eng.*, vol. 23, no. 9, pp. 1282–1298, 2011.

[78] J. G. Xiang, C. Guo, and A. Aboulnaga, Scalable maximum clique computation using MapReduce, in *Proc. 2013 IEEE 29th Int. Conf. Data Engineering*, Brisbane, Australia, 2013, pp. 74–85.

[79] Di G. Fatta and M. R. Berthold, Dynamic load balancing for the distributed mining of molecular structures, *IEEE Trans. Parall. Distrib. Syst.*, vol. 17, no. 8, pp. 773–785, 2006.

[80] Y. F. Luo, J. H. Guan, and S. G. Zhou, Towards efficient subgraph search in cloud computing environments, in *Proc. 16th Int. Conf. Database Systems for Advanced Applications*, Hong Kong, China, 2011, pp. 2–13.

[81] G. Buehrer, S. Parthasarathy, and Y. K. Chen, Adaptive parallel graph mining for CMP architectures, in *Proc. Sixth Int. Conf. Data Mining*, Hong Kong, China, 2006, pp. 97–106.

[82] S. Aridhi, L. D' Orazio, M. Maddouri, and M. E. Nguifo, Density-based data partitioning strategy to approximate large-scale subgraph mining, *Inf. Syst.*, vol. 48, pp. 213–223, 2013.

[83] M. A. Bhuiyan and M. A. Hasan, MIRAGE: An iterative MapReduce based frequent subgraph mining algorithm, arXiv Preprint arXiv: 1307.5894, 2013.

[84] C. H. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, Arabesque: A system for distributed graph mining, in *Proc. 25th Symp Operating Systems Principles*, New York, NY, USA, 2015, pp. 425–440.

[85] W. Q. Lin, X. K. Xiao, and G. Ghinita, Large-scale frequent subgraph mining in MapReduce, in *Proc. 2014 IEEE 30th Int. Conf. Data Engineering*, Chicago, IL, USA, 2014, pp. 844–855.

[86] F. C. Qiao, X. Zhang, P. Li, Z. Y. Ding, S. S. Jia, and H. Wang, A parallel approach for frequent subgraph mining in a single large graph using Spark, *Appl. Sci.*, vol. 8, no. 2, p. 230. 2018.

[87] A. M. Petermann, M. Junghanns, and E. Rahm, DIMSpan-transactional frequent subgraph mining with distributed in-memory dataflow systems, in *Proc. Fourth IEEE/ACM Int. Conf. Big Data Computing, Applications and Technologies*, New York, NY, USA, 2017.

Bismita S. Jena is pursuing PhD in computer science at Georgia State Untiversity. She got a master's degree in computer science from Georgia State Untiversity in 2012. Her research interests include data mining, machine learning, and cloud computing. She also works as a BigData Solution Architect with a fortune 50 healthcare company. She encourages and actively promotes Women in Computing.

Cynthia Khan is a student at Georgia State University, where she is pursuing a master's degree in computer science with a concentration on data science and analytics. She obtained a bachelor of science degree from Georgia State University in 2018 as Summa Cum Laude and have been pursuing various research projects in the field of data mining since then. Currently, her work involves topics such as optimizing neural network models

to predict tweet geolocations, creating open source pipelines to collect, clean, and present medical data to be queried by external applications, and the use of big data resources for advanced graph mining. Besides academic pursuits, Cynthia is an active participant in promoting diversity in IT.

Rajshekhar Sunderraman received the PhD degree in computer science in 1988 from Iowa State University. He has held faculty positions at Wichita State University (1988-1996) and Georgia State University (1996-present). His research interests include deductive databases and logic programming, data modeling, knowledge engineering, semantic web, bioinformatics, and geoinformatics and he has over 150 publications in various computer science journals and conference proceedings. He is the author of a popular text book, *Oracle 10G Programming: A Primer*, published by Addison Wesley in 2008. He served as Chair of the Computer Science Department at Georgia State University from 2013 to 2017 and is currently Professor and Associate Chair.