Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 15, no. 6, pp. 753–776 (2011)

FP-GraphMiner – A Fast Frequent Pattern Mining Algorithm for Network Graphs

R. Vijayalakshmi[1] *R. Nadarajan*[1] *John F. Roddick*[2] *M. Thilaga*[1] *P. Nirmala*[1]

[1]Department of Mathematics and Computer Applications,
PSG College of Technology,
Coimbatore 641004, Tamil Nadu, India
[2]School of Computer Science, Engineering and Mathematics,
Flinders University,
PO Box 2100, Adelaide, SA 5001, South Australia.

Abstract

In recent years, graph representations have been used extensively for modelling complicated structural information, such as circuits, images, molecular structures, biological networks, weblogs, XML documents and so on. As a result, frequent subgraph mining has become an important subfield of graph mining. This paper presents a novel Frequent Pattern Graph Mining algorithm, FP-GraphMiner, that compactly represents a set of network graphs as a Frequent Pattern Graph (or FP-Graph). This graph can be used to efficiently mine frequent subgraphs including maximal frequent subgraphs and maximum common subgraphs. The algorithm is space and time efficient requiring just one scan of the graph database for the construction of the FP-Graph, and the search space is significantly reduced by clustering the subgraphs based on their frequency of occurrence. A series of experiments performed on sparse, dense and complete graph data sets and a comparison with *MARGIN*, *gSpan* and *FSMA* using real time network data sets confirm the efficiency of the proposed FP-GraphMiner algorithm.

Keywords: frequent pattern mining, frequent subgraph, graph database, graph mining, maximal frequent subgraph, maximum common subgraph.

Submitted:	Reviewed:	Revised:	Reviewed:
February 2011	May 2011	May 2011	August 2011
Revised:	Accepted:	Final:	Published:
August 2011	October 2011	October 2011	November 2011

Article type: Regular paper
Communicated by: G. Liotta

E-mail addresses: rv@mca.psgtech.ac.in (R. Vijayalakshmi) rn@mca.psgtech.ac.in (R. Nadarajan) john.roddick@flinders.edu.au (John F. Roddick) mta@mca.psgtech.ac.in (M. Thilaga) pna@mca.psgtech.ac.in (P. Nirmala)

1 Introduction

The increasing use of large communication, financial, telecommunication and social networks is providing a substantial source of problems to the graph data mining community. These differ from many traditional data mining problems in that the data records representing transactions between set of entities are not considered independent, and the inter-transaction dependencies can be represented as trees, lattices, sequences and graphs. As a result, there has been an increasing interest in studying the properties, models and algorithms applicable to graph-structured data to address these issues.

Much current research in graph based data mining focuses on estimating the reputation and/or popularity of items in a network, mining query logs, performing query recommendations, web and social network applications, and so on. As such, the frequent subgraph discovery problem occupies a significant position among the various graph based data mining algorithms.

The problem of discovering frequent patterns can be stated as follows. Given a transaction database D consisting of a set of transactions $t_1, t_2, \ldots, t_n$ and a user-specified minimum support (or threshold) σ, the frequent pattern mining problem is to discover the complete set of patterns with a minimum support σ in D. The support σ for a given frequent pattern is defined as the ratio of the number of graphs containing the pattern to the total number of graphs. Depending on the specific problem formulation, the input transactions and pattern specification can be an itemset, a sequence, a tree, or a graph. Frequent subgraph mining is a demanding problem as there are an exponential number of subgraphs contained in a graph. For a graph with e edges, the number of possible frequent subgraphs could be as large as 2^e. As the core operation of subgraph isomorphism testing is NP-complete, it is critical to minimise the number of subgraphs that need to be considered [9].

Although the ideas in the paper are widely applicable, in this paper we base our examples and experiments on the discovery of interesting frequent patterns from a large communication network. This has a wide range of applications such as network traffic analysis, detection of node failures in a network, routing algorithms, and so on. For example, to maximise the efficiency of a network, managing network traffic is essential. Once the most frequently used paths are identified better routing algorithms can be devised. To achieve this, a communication network can be modelled as either an undirected or directed graph with the clients and servers (labelled by their IP addresses) as nodes and the communication channels between them as edges. Since the IP addresses in a network are unique, no two nodes have the same label.

The focus of this work is the frequent pattern mining of graphs to discover all frequent subgraphs contained in at least σ of the graphs in the database. The proposed Frequent Pattern Miner (FP-GraphMiner) algorithm calculates the frequent edges present in various graphs efficiently creating a special undirected graph called a Frequent Pattern-Graph (or FP-Graph). Once the graph is constructed, all frequent subgraphs can be determined for any given support. The experimental results validate the effectiveness of the proposed algorithm.

The rest of the paper is organised as follows. The remainder of this section presents the formal definitions and notations used. Section II discusses related work in the area of frequent subgraph mining while Section III describes the proposed FP-GraphMiner algorithm. Section IV provides correctness proofs of the technique while Section V provides a complexity analysis of the algorithms. Section VI deals with the empirical performance evaluation of the algorithm using synthetic datasets consisting of sparse, non-sparse (dense) and complete graph datasets, and a comparative study with *MARGIN*, *gSpan* and *FSMA* using real time data sets. Some conclusions are provided in Section VII. The algorithms themselves are given in an Appendix.

1.1 Definitions and Notations

The definitions and notations used in this paper are described below [3].

Labeled Graph

A labeled graph G is a 4-tuple, $G = (V, E, \alpha, \beta)$ where V is a finite set of vertices, $E \subseteq V \times V$ is a set of edges, $\alpha : V \to L$ denotes a vertex labeling function and $\beta : E \to L$ denotes a edge labeling function. Edge (u, v) originates from node u and terminates at node v. For an undirected graph, (v, u), and (u, v) denote the same edge, thus $\beta(u, v) = \beta(v, u)$.

Graph Isomorphism

Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if they are topologically identical to each other, that is, there is a mapping from G_1 to G_2 such that each edge in E_1 is mapped to a single edge in E_2 and vice versa. In the case of labeled graphs, this mapping must also preserve the labels on the vertices and edges.

Subgraph

A graph $G_2 = (V_2, E_2)$ is a subgraph of another graph $G_1 = (V_1, E_1)$ iff $V_2 \subseteq V_1$, and $E_2 \subseteq E_1 \wedge ((v_1, v_2) \in E_2 \implies v_1 \in V_2 \text{ and } v_2 \in V_2)$.

Induced Subgraph

Let $G_1 = (V_1, E_1, \alpha_1, \beta_1)$ and $G_2 = (V_2, E_2, \alpha_2, \beta_2)$ be graphs. G_2 is an induced subgraph of G_1, $(G_2 \subseteq G_1)$, if $V_2 \subseteq V_1$, $\alpha_1(v) = \alpha_2(v)$ for all $v \in V_2$, $E_2 = E_1 \cap (V_2 \times V_2)$, and $\beta_1(e) = \beta_2(e)$ for all $e \in E_2$. Given a graph $G_1 = (V_1, E_1, \alpha_1, \beta_1)$, if any subset $V_2 \subseteq V_1$ of its vertices uniquely defines a subgraph, this subgraph is called the subgraph induced by V_2.

Subgraph Isomorphism

Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, the problem of subgraph isomorphism is to find an isomorphism between G_2 and a subgraph of G_1, that is, to determine whether or not G_2 is included in G_1.

Frequent Subgraph

Given a labeled graph dataset $GD = \{G_1, G_2, \ldots, G_k\}$, support or frequency of a subgraph g is the percentage (or number) of graphs in GD where g is a subgraph. A frequent subgraph is a graph whose support is no less than a minimum user-specified support threshold.

2 Related Work

Graphs serve as a promising means of generically modelling a variety of relations among data [4]. They can be used to effectively model the structural and relational characteristics of a variety of datasets arising in the areas of physical sciences and chemistry such as fluid dynamics, astronomy, structural mechanics, and ecosystem modelling, life sciences such as genomics, proteomics, health informatics, and information security such as information assurance, infrastructure protection, and terrorist-threat prediction/identification. Much research has focused on finding patterns from a single large network [10], mining patterns using domain knowledge from bioinformatics [6], and finding frequent subgraphs [5, 9, 12]. A strong interdisciplinary research area in graph mining is the problem of finding frequent subgraphs present in huge graph databases. This has application in different fields including network intrusion [11], semantic web [16, 1], behavioural modelling [13] and link analysis [8, 15].

A number of algorithms have used a depth-first search to enumerate candidate frequent subgraphs [22]. The *gSpan* algorithm builds a new lexicographic order among graphs, and maps each graph to a unique minimum DFS code as its canonical label. Based on this lexicographic order, *gSpan* adopts a depth-first search strategy to mine frequent connected subgraphs efficiently [21]. Other subgraph mining algorithms focus on a level wise search scheme based on the Apriori property to enumerate the frequent subgraphs that propose an efficient frequent subgraph mining algorithm [7, 14].

There are two common problems underpinning subgraph mining work such as this. First, the maximum common subgraphs (or MCS) problem often provides a suite of benchmarking activities for assessing the performance of widely used algorithms. These include measuring the similarity between two graphs, finding maximum common edge subgraphs (MCES), and the McGregor, Durand and Pasari algorithms for determining MCS of two given graphs [3]. Second, maximal frequent subgraph mining finds all frequent subgraphs g_i such that no frequent subgraph g_j exists where g_i is a subgraph of g_j. A typical approach to the maximal frequent subgraph mining problem is to modify the Apriori based approach with additional pruning steps. An approach to find the maximal frequent subgraphs from graph lattices has been discussed using the *MARGIN* algorithm [17, 18]. It represents the search space as a graph lattice and mines the maximal frequent subgraphs while pruning the lattice space considerably. The *ExpandCut* algorithm recursively finds the candidate subgraphs. *MARGIN* explores a much smaller search space by visiting the lattice around the *f-cut* nodes.

The Frequent Subgraph Mining Algorithm (*FSMA*) finds all the subgraphs with a given minimum support in a given graph data set [20]. It uses the normalized incidence matrix to present the subgraphs. By scanning the graph database, *FSMA* first finds all the frequent edges, termed 1-edge frequent subgraphs, which are then extended by adding frequent edges to get 2-edge frequent subgraphs. This procedure of subgraph extension is repeated until no more frequent subgraphs can be generated. The algorithm extends the frequent subgraphs by adding only the frequent edges instead of enumerating all subgraphs which greatly reduces the time complexity.

3 The **FP-GraphMiner** Algorithm

Many currently proposed algorithms for mining frequently occurring patterns scan the graph database more than once during the mining process. Since in practice it is commonly disk I/O that most increases response times [2], for large graph databases, multiple scans can increase the time complexity substantially. The proposed study focuses on finding frequent subgraphs in a graph database containing a huge number of related graphs using a single database pass. The objective of this algorithm is to store the details of all frequent subgraphs into a single compact undirected graph by scanning the graph database once and to mine all the frequent subgraphs with any support σ.

As discussed above, a communication network graph with unique node labels is considered for the study. A communication network can be characterised as a time series of graphs, with IP addresses (clients or servers) as nodes and the connection between them as edges. An edge-based array representation, which is more efficient compared to the vertex-based adjacency matrix representation, is used. The memory requirement of this representation is half that of the adjacency list format since it does not store an edge twice.

Each edge of the graph is represented as the 3-tuple $\langle S, D, EL \rangle$, where S is the source node, D is the destination node, and EL is the edge label. Each tuple is read into an Edge Array, EA, which is a collection of all the edges of the graph. For an undirected graph, the edge array has the tuples arranged in lexicographic order of source, destination and edge label. Since no edges are repeated (edges are distinct), the number of tuples in the edge array is the number of edges in the graph. The various definitions and notations used in the proposed algorithm are as follows.

Let $GD = \{G_1, G_2, \ldots, G_k\}$ be a graph database with k graphs. Each *Distinct Edge*, DE is represented as $DE = \langle S, D, EL \rangle$.

BitCode of a Distinct Edge

Let m be the number of distinct edges of k graphs. The BitCode of a distinct edge DE_i denoted as $BitCode(DE_i), 1 \leq i \leq m$, is a k length bit string, each bit corresponding to a graph in GD, consisting of 1's in the positions of the graphs in which the edge is present and 0's if it is absent. The BitCode gives information about the graphs in which the distinct edge is present.

Weight of a BitCode

The weight of a BitCode of an edge DE_i, denoted as $WT(DE_i)$, is the count of 1's in it, (i.e. the number of graphs in which the edge appears). Since the weight of all edges in a given $Node$ or a given $Cluster$ are the same (see below), the term weight can also be applied to $Nodes$ and $Clusters$.

Frequency Table

A Frequency Table FT is defined as a collection of distinct edges of k graphs in GD in decreasing order with respect to the binary encoding of the BitCodes. Each row in the frequency table contains a 2-tuple $\langle DE_i, BitCode(DE_i)\rangle$, where DE_i and $BitCode(DE_i)$ represent the distinct edge and the graphs in which the edge is present respectively.

Frequent Pattern Graph

A Frequent Pattern Graph, FP-Graph $= \{Node, Edge\}$ is a special type of undirected graph constructed as a collection of $Nodes$ and $Edges$ where a $Node$ is a collection of distinct edges with a common property and an $Edge$ is a link between two $Nodes$. The FP-Graph constructed from the frequency table has the following properties.

1. Each $Node$ in the FP-Graph is a collection of subgraphs with the same BitCode (common features). The maximum number of $Nodes$ in an FP-Graph of k graphs is $2^k - 1$.
2. Each $Edge(U, V)$ originates from $Node\ U$ and terminates at $Node\ V$ with an edge label as decimal equivalent of the BitCode of $Node\ V$ where $Node\ V$ is the immediate superset of $Node\ U$, i.e., $WT(Node\ U) < WT(Node\ V)$. The FP-Graph construction algorithm outlined in Section 3.1.1 shows how the $Nodes$ are linked.
3. The $Nodes$ with the same BitCode weights are grouped into $Clusters$. Each $Cluster$ is identified by its unique weight. The maximum number of $Clusters$ in the FP-Graph is k.

   To summarize, each $Node$ contains the subgraphs with the same BitCode and each $Cluster$ contains the $Nodes$ with the same BitCode weight.
4. The $HeaderNode$ is an empty $Node$ pointing to the $Nodes$ in a $Cluster$ with maximum weight (highest support).

DFS Walk in Frequent Pattern Graph

A DFS walk in an FP-Graph is defined as a walk (search) starting from the $Node\ U$ in a $Cluster$ with a given support (σ) to the $HeaderNode$ with no backtracking through a sequence of $Nodes$ $U_1, U_2, \ldots, U_k$, such that $U = U_1$ and $HeaderNode = U_k$, where all U_i are $Nodes$ in the path satisfying the following condition, $WT(BitCode(U_1)) < WT(BitCode(U_2)) < \ldots < WT(BitCode(U_i)) < \ldots < WT(BitCode(U_k))$. The DFS walk from each

Node in the *Cluster* to the *HeaderNode* yields all the subgraphs with the given support.

3.1 The Algorithm

FP-GraphMiner takes the edges of k graphs represented as Edge Arrays as input and constructs the Frequency Table FT with the distinct edges of all graphs stored only once. An important property of the proposed algorithm is that the graph database is scanned only once to construct the frequency table. From the frequency table, the FP-Graph is constructed. The FP-GraphMiner algorithm has two phases.

3.1.1 Phase I: FP-Graph construction

In Phase I, the distinct edges DE are obtained by performing a union operation on the k graphs in the database and are stored in the Frequency Table FT. The distinct edges are arranged in the descending order of their BitCodes. By grouping the edges with the same BitCodes, it is possible to obtain the subgraphs and the details about the graphs in which they are present. As it is difficult to retrieve all the frequent subgraphs for a given support from the frequency table, the FP-Graph is constructed. The *Nodes* with the same BitCode weight are grouped into *Clusters*. Each *Cluster* is a collection of the subgraphs in GD with a unique support. The *Nodes* in each *Cluster* are linked to obtain the FP-Graph according to the two step algorithm below.

1. *Clustering the Nodes.*
   The *Nodes* with same BitCode weights are grouped into *Clusters*. Thus each *Cluster* is a collection of subgraphs with the same support. At the worst case, the maximum number of *Clusters* would be the number of graphs.

2. *Connecting the Nodes in different Clusters.*
   The *Clusters* are arranged in the hierarchy of increasing order of the support values. Let C_i, C_j be two successive *Clusters* with $\sigma(C_i) < \sigma(C_j)$. *Cluster* C_j is termed the nearest superset of *Cluster* C_i. A *Node* p_1 in *Cluster* C_i can be linked to the *Node(s)* p_2 in *Cluster* C_j by undirected edge(s) if the distance measure $d(p_1, p_2)$, $d_{(p_1 \in C_i, p_2 \in C_j)} p_1 \cap p_2 = p_1$ is satisfied. This means that *Node* p_2 in one *Cluster* is a superset of p_1 in another *Cluster* only if the distance between them is p_1 itself. Any *Node* in a *Cluster* can be linked to one or more superset *Nodes*. If $d(p_1, p_2) \neq p_1$ for all the *Nodes* in C_j, the *Nodes* in the next higher level *Cluster* are examined to find the nearest superset *Node(s)* of p_1.

   For instance, given the example described below, if p_2 is the node containing edges $\{ab, ac, bc, bd, df\}$ (in Figure 3 the Cluster with 100% support) and p_1 contains edges $\{de, ef, eg, fg\}$ (the lefthand Cluster with 80% support), then $d(p_1, p_2) = BitCode(p_1) \cap BitCode(p_2) = 11101 \cap 11111 = 11101 = BitCode(p_1)$. Thus p_2 is a superset of p_1.

Both the parameters used in the proposed algorithm and the algorithms themselves are given in the appendix.

To illustrate the construction of an FP-Graph consider a network communication database GD, as shown in Figure 1, consisting of a time series of graphs obtained by measuring the state of connectivity of the network at regular time intervals[1].

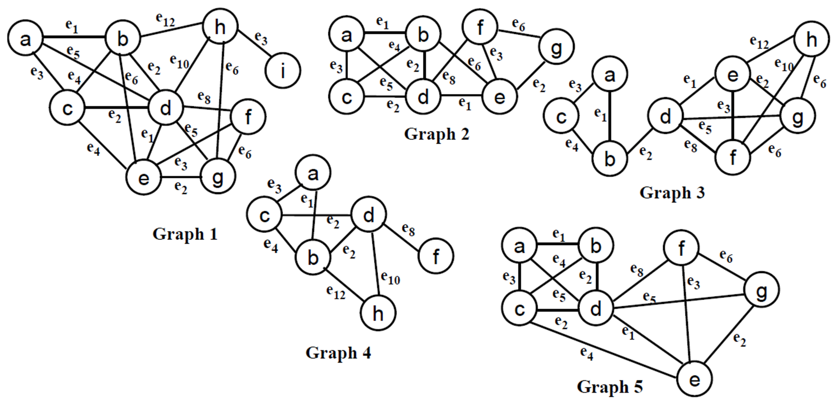


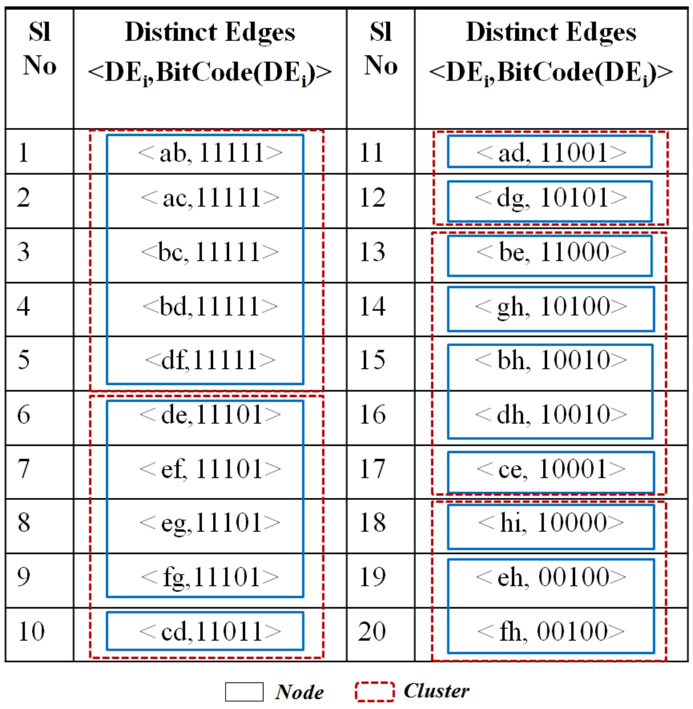
Figure 1: A Communication Network Graph Database GD with 5 graphs

The input to FP-Graph construction algorithm is the edge arrays of these graphs as given below.

$$\begin{aligned}
EA(G_1) &= \{abe_1, ace_3, ade_5, bce_4, bde_2, bee_6, bhe_{12}, cde_2,\\
&\quad cee_4, dee_1, dfe_8, dge_5, dhe_{10}, efe_3, ege_2, fge_6,\\
&\quad ghe_6, hie_3\}\\
EA(G_2) &= \{abe_1, ace_3, ade_5, bce_4, bde_2, bee_6, cde_2, dee_1,\\
&\quad dfe_8, efev_3, ege_2, fge_6\}\\
EA(G_3) &= \{abe_1, ace_3, bce_4, bde_2, dee_1, dfe_8, dge_5, efe_3,\\
&\quad ege_2, ehe_{12}, fge_6, fhe_{10}, ghe_6\}\\
EA(G_4) &= \{abe_1, ace_3, bce_4, bde_2, bhe_{12}, cde_2, dfe_8, dhe_{10}\}\\
EA(G_5) &= \{abe_1, ace_3, ade_5, bce_4, bde_2, cde_2, cee_4, dee_1,\\
&\quad dfe_8, dge_5, efe_3, ege_2, fge_6\}
\end{aligned}$$

Since the edge labels are not significant in the process, they are not considered further. From the edge arrays the Frequency Table FT is constructed as shown in Figure 2.

[1] This example is from the experimental data outlined in Section 6 with each graph obtained by aggregating ten one-second graphs through a UNION operation.



Sl No	Distinct Edges $<DE_i, BitCode(DE_i)>$	Sl No	Distinct Edges $<DE_i, BitCode(DE_i)>$
1	< ab, 11111 >	11	< ad, 11001 >
2	< ac, 11111 >	12	< dg, 10101 >
3	< bc, 11111 >	13	< be, 11000 >
4	< bd, 11111 >	14	< gh, 10100 >
5	< df, 11111 >	15	< bh, 10010 >
6	< de, 11101 >	16	< dh, 10010 >
7	< ef, 11101 >	17	< ce, 10001 >
8	< eg, 11101 >	18	< hi, 10000 >
9	< fg, 11101 >	19	< eh, 00100 >
10	< cd, 11011 >	20	< fh, 00100 >

Node
Cluster

Figure 2: Frequency table of *GD*

Each row in the frequency table is a distinct edge obtained by performing a UNION operation on the edge arrays $EA(G_1), EA(G_2), \ldots, EA(G_5)$. These edge arrays are then sorted in descending order of their BitCodes, with the FP-Graph then constructed from this frequency table. The distinct edges with the same BitCode are grouped into a *Node* in the FP-Graph. For instance, the distinct edges $\langle ab, ac, bc, bd, df\rangle$ form a *Node* with BitCode `11111`. The solid and the dotted rectangles in Figure 2 show the *Nodes* and *Clusters* respectively. The *Nodes* in the various *Clusters* are linked to form the FP-Graph as shown in Figure 3. The graphs in which these edges are present are also listed for ease of understanding. The links of a *Node* to various other *Nodes* are established by finding its superset *Nodes*. Any *Node* in FP-Graph can have one or more superset *Nodes*. This graph can be now mined for various tasks such as finding frequent subgraphs, maximum common subgraphs, maximal frequent subgraphs, the graphs containing the given query graph and its support, and so on.

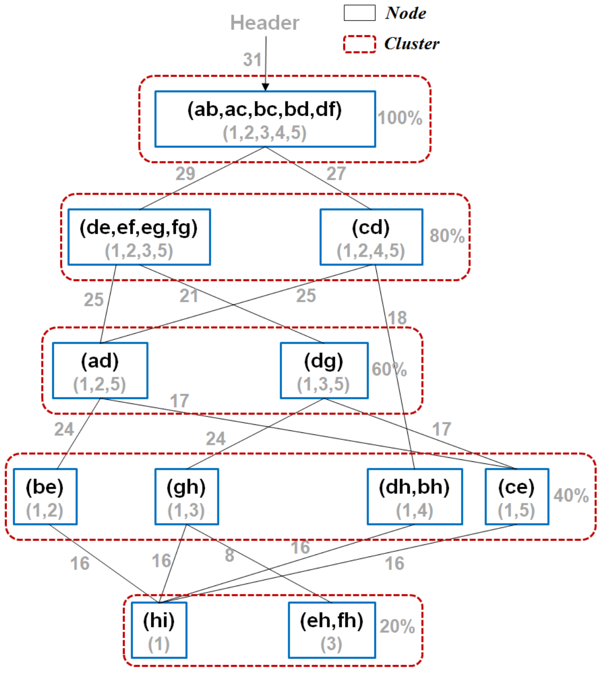


Figure 3: FP-Graph of the graphs in GD

3.1.2 Phase II: FP-GraphMiner

The objective of the FP-GraphMiner algorithm is as follows. Given a support σ, all the frequent subgraphs with at least that support can be determined efficiently from FP-Graph by performing DFS walks starting from each $Node$ in the $Cluster$ with the given support σ to the $HeaderNode$. The collection of all the edges of the $Nodes$ in the DFS path constitutes a frequent subgraph. The number of DFS walks from a $Cluster$ with the given support is the number of frequent subgraphs. If the input graphs are highly dissimilar, the resulting frequent subgraphs are not connected. By clustering all the $Nodes$ with the same support within the same $Cluster$ the time taken to perform the search is significantly reduced.

In the case of a communication network, analyzing the frequent subgraphs with various support values provides information about how efficiently the network is utilized. Conversely, the nodes with lower support values are those communication paths that are used less frequently. This knowledge facilitates improvement in the performance of the overall network by efficiently utilising channels and for devising more effective routing algorithms. Thus, the proposed algorithm serves both as an efficient tool for communication network analysis and for detection of failure nodes.

Finding all frequent subgraphs with a given support σ.
The FP-GraphMiner algorithm (see Appendix A) performs DFS walks starting from the $Cluster$ having the specified support to the $HeaderNode$ to obtain all frequent subgraphs. For instance, the frequent subgraphs with 60% support are shown in Figure 4.

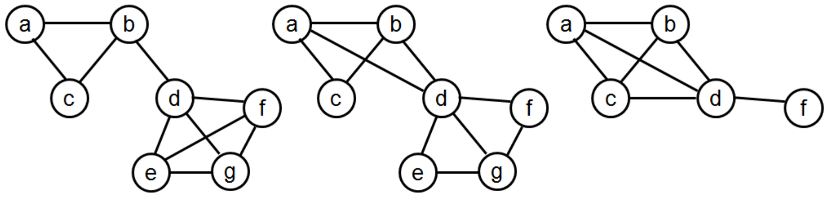


Figure 4: All Frequent Subgraphs with support 60%

The frequent subgraphs extracted from the FP-Graph for a given support need not be induced. Preserving the induced nature of frequent subgraphs obtained by the above algorithm is application dependent. The induced frequent subgraph with 100% support is the maximum common subgraph. In a communication network, finding the frequent subgraphs representing communication paths for a given support need not be induced. On the other hand, if the problem is to find the sub network resulting after node failures, then induced subgraph mining is essential.

Finding graphs in GD containing the Query Graph Q.
Given a query graph Q, the graphs in GD containing it can be easily identified by performing a Breadth First Search BFS starting from the $HeaderNode$ till all the edges of Q are obtained. The BitCodes of these $Nodes$ are collected into an array $BFS(Q)$. An AND operation on the

BitCodes in $BFS(Q)$ gives a BitCode and the position of 1's in the resulting BitCode shows the graphs containing Q. For instance, given a query graph Q as shown in Figure 5(a), the algorithm performs a BFS starting from the $HeaderNode$ to the node containing the last edge. The edges found as a result of BFS on the FP-Graph to find the edges of Q are shown in Figure 5(b).

`BFS(Q)={11111,11101,11011,11000,10001}`

By performing an AND operation on the BitCodes, the query graph code of Q, $QGraphCode(Q)$ is obtained. Thus, for the given example, $QGraphCode(Q)$ is equal to {`10000`}. The positions of 1's in the $QGraphCode$ show the graphs in which Q is contained. Hence in the example, Q is present only in Graph 1. The support of Q in GD is computed as $\sigma(Q) = 1/5 = 20\%$.

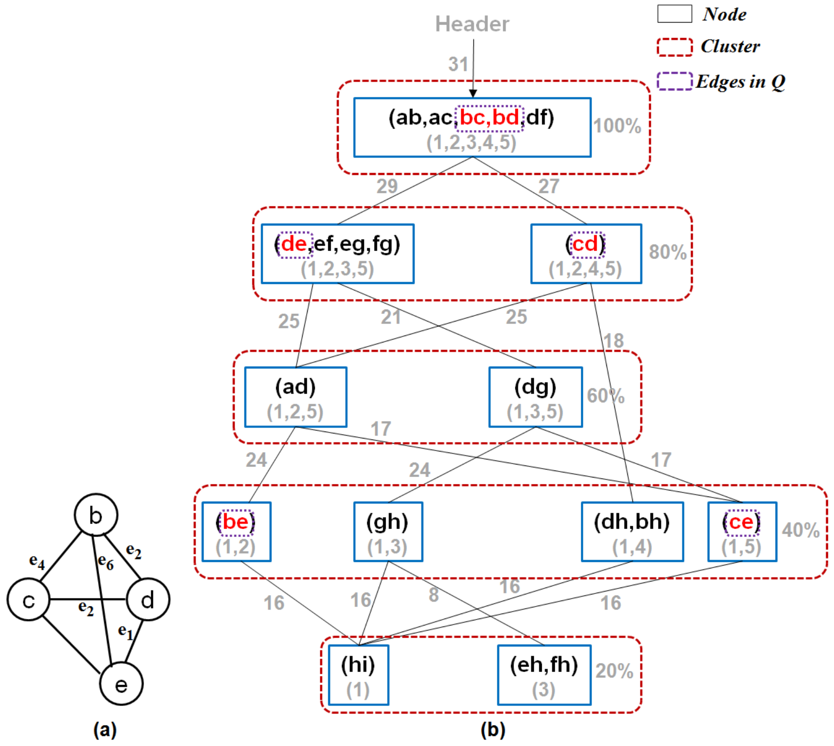


Figure 5: (a) Query Graph Q (b) BFS of FP-Graph for finding Q

The FP-Graph could be efficiently mined for detecting outliers also. For instance, node i in graph 1 has only a single data transfer with node h (with frequency of utilization (support) = 1/5). If this is a server failure, necessary action could be taken to identify and remedy the problem.

The *MARGIN* and *FSMA* algorithms scan the graph database more than once by following an incremental edge growing methods while finding the maximal frequent subgraphs and all the frequent subgraphs respectively. The FP-GraphMiner algorithm scans the graph database once only to construct the FP-Graph. This FP-Graph represents all frequent subgraphs in a single data structure. The frequent subgraphs with any given support can be mined simply by performing DFS walks in the FP-Graph. The maximum number of $Clusters$ scanned during each DFS walk would be the number of $Clusters$ in the FP-Graph. Thus the proposed algorithm is efficient.

4 Proofs of Correctness

In this section, the correctness of the proposed algorithm is shown. First the essential claim that the maximum number of different weights of distinct edges of k network graphs is proved as k.

Claim 1 *For a graph database GD with k network graphs, the maximum number of different weights of distinct edges $DE_i, 1 \leq i \leq m$ is k.*

Proof: Each distinct edge DE_i has a BitCode for which the number of 1's in the BitCode is termed its *weight*. For k graphs, each BitCode has k bits. Hence, given k length BitCodes, the weight of the BitCode can range from 1 to k. □

Claim 2 *The maximum number of $Clusters$ and $Nodes$ in an FP-Graph of k graphs in a graph database GD in the worst case are k and $2^k - 1$ respectively.*

Proof: From Claim 1, it follows that k different combinations of weights of BitCodes are possible with k graphs and hence, by the definition of $Cluster$ formation, there is a maximum of k $Clusters$. Each $Node$ in an FP-Graph has distinct edges with the same BitCode. The length of the BitCode of each edge is k. As k bits are used for representing one $Node$, in the worst case, 2^k-1 distinct combinations are possible excluding the BitCode {000}. Hence, there can be a maximum of $2^k - 1$ $Nodes$. For instance, given $k = 3$, all possible $2^k - 1$ combinations of BitCodes of the distinct edges are {`001,010,100,011,110,101,111`}. Grouping these codes based on weights would yield only 3 groups {`001,010,100`}, {`011,110,101`}, {`111`}. □

Claim 3 *A DFS walk starting from the $Nodes$ in the $Cluster$ with a given support σ to the $HeaderNode$ results in all the frequent subgraphs with support σ.*

Proof: The FP-Graph is a collection of all distinct edges of k graphs arranged in order of their frequencies into various $Clusters$. Performing a DFS walk from any $Node$ to the $HeaderNode$ gives a frequent subgraph of $Nodes$ with at least the given support. The DFS walks starting from each $Node$ in the $Cluster$ with the given support σ to the $HeaderNode$ results in all the frequent subgraphs with $\sigma, 1 \leq \sigma \leq 100$. □

Claim 4 *The number of frequent subgraphs obtained by a DFS walk from any $Node$ with a given support depends on its links with its superset $Nodes$.*

Proof: There are a number of DFS walks starting from a $Node$ and proceeding to the $HeaderNode$ via superset $Nodes$. This produces a number of frequent subgraphs. For instance, in Figure 3, the links of the $Node$ with subgraph ad with its superset $Nodes$ are two. Hence, performing DFS traversals shows two different paths from the $Node$ containing the subgraph ad to the $HeaderNode$, the two frequent subgraphs with the 60% support are obtained as shown in Figure 6. □

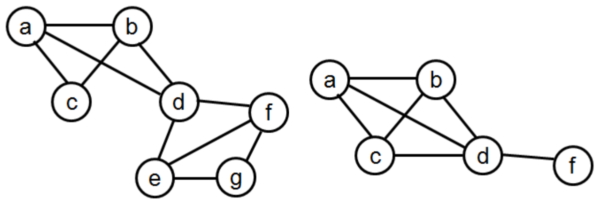


Figure 6: Two Frequent subgraphs of a Node containing subgraph (ad) with the 60% support

5 Computational Complexity Analysis

In this section we provide an analysis of the time complexity of FP-Graph construction and the FP-GraphMiner algorithm.

5.1 FP-Graph Construction

The construction of FP-Graph includes constructing a frequency table and systematically linking the Nodes in various Clusters.

Constructing Frequency Table (FT).

All the edges in the Edge Array of k graphs are scanned once to construct FT. Let the total number of edges in the k graphs be NE and the number of Distinct Edges be m. All distinct edges along with their BitCodes, are stored in FT in descending order of BitCode. The time required for arranging the rows in FT is $mlogm$ using a hash-based implementation. The total time complexity for constructing FT is $O(NE + mlogm)$.

Constructing FP-Graph.

Each group of edges with the same BitCode comprises one $Node$ in the FP-Graph. As the edges are stored in the decreasing order of their BitCodes, the number of comparisons to group the edges into $Nodes$ is m. Let the number of $Nodes$ be $N, (N \leq 2^k - 1$ from Claim 2). To link any

$Node\ i, 1 \leq i \leq N$, to its immediate superset $Node(s)$, the BitCodes of $Nodes$ from $Node\ i-1$ to $Node\ 1$ are compared by performing $i-1$ AND operations. The $Nodes$ within the same $Cluster$ need not be compared. The number of comparisons needed at the worst case to link the Nodes in FP-Graph is $0+1+2+\ldots+(N-1)$ is $N(N-1)/2$. Thus the complexity of $O(N^2)$.

5.2 FP-GraphMiner

The FP-Graph is mined to obtain all frequent subgraphs with the given support.

Finding all Frequent subgraphs for a given support.

Given a specific support, the maximum number of $Nodes$ visited to find a frequent subgraph using a DFS walk in the FP-Graph is $2^k - 1$. So, in the worst case, to find all frequent subgraphs with a given support, the number of $Nodes$ visited is the number of $Clusters * links$ of the $Node$ with its superset $Node(s)$.

Finding graphs containing a given query graph Q.

Given a query graph Q, the number of $Nodes$ visited to find the edges of Q by performing BFS is N. The time needed for AND operation on the BitCodes of resulting $Nodes$ to find the graphs in which Q is present is m. Thus the complexity is $O(Nm)$.

Finding significant nodes in the network.

The edges of the graphs are taken into account in constructing the FP-Graph. From the FP-GraphMiner algorithm and the experimental study, it is difficult to analyze the efficiency of the nodes where different nodes of the network have different degrees (number of nodes to which a node establishes communication links) at different support values. This means that the nodes with different support values have different significance values. Thus to calculate the utilization of the various nodes of the network, some statistical measure is needed to find the contribution of each. The consistency of each node can be measured using a statistical weighted average which takes into account the proportional relevance of each component.

Let the maximum degree of each node in the graph database be D and $\sigma(\mathrm{node_i})$ be the support of the cluster containing $(\mathrm{node_i})$, $1 \leq i \leq D$. In the given example, the number of nodes in GD is 9. As each node can be connected to a maximum of 8 other nodes $D = 8$. From the FP-Graph given in Figure 3, the weighted average (WA) of each node is computed by considering the nodes in all the clusters using the equation:

$$\mathrm{WA}(\mathrm{node_i}) = \lfloor \Sigma(Degree(\mathrm{node_i}) * \sigma(\mathrm{node_i}))/D \rfloor, 1 \leq i \leq D$$

Table 1: Weighted Averages of nodes in FP-Graph of Figure 6

σ	a	b	c	d	e	f	g	h	i
100	2	3	2	2	0	1	0	0	0
80	0	0	1	2	3	2	2	0	0
60	1	0	0	2	0	0	1	0	0
40	0	3	1	1	2	0	0	3	0
20	0	0	0	0	1	1	0	3	1

The weighted averages of all the nodes are computed as follows.

$$\begin{aligned} WA(a) &= \lfloor((2*100)/8) + ((1*60)/8)\rfloor = 32 \\ WA(b) &= \lfloor((3*100)/8) + ((3*40)/8)\rfloor = 53 \\ WA(c) &= \lfloor((2*100)/8) + ((1*80)/8) + ((1*40)/8)\rfloor = 40 \\ &\vdots \\ WA(i) &= \lfloor(1*20)/8\rfloor = 2 \end{aligned}$$

The degrees of the nodes present in the subgraphs of FP-Graph with various support values are computed as above and are listed along with the weighted averages in Table 1.

After finding the weighted averages, the nodes can be ranked based on their frequency of usage. The node with highest WA is the most frequently used node. In our example, it might be found that server d has been used more compared to others and thus the failure of server d would affect the functioning of the network.

6 Experimental Analysis

The experiments were conducted on a 2.8 GHz Intel Pentium Dual Core machine with 504MB RAM using Microsoft Windows XP with the algorithms coded in C. A synthetic Graph Database GD consisting of sparse, non-sparse and complete graphs was generated to analyse the behaviour of the FP-GraphMiner algorithm and to investigate the time complexities. The time taken for detecting all frequent subgraphs in GD with k=100, 500 and 1000 is shown in Table 2. Synthetic data sets containing a maximum of 100, 500, and 1000 nodes for various support values of 25%, 50%, 75% and 100% were analysed.

The experimental study shows that the time taken to detect all frequent subgraphs is inversely proportional to the support of frequent subgraphs. If the graphs in the database are more related, the time taken to detect the frequent subgraphs is lower. On the other hand, when the support of a subgraph is less, the length of the DFS path starting from the $HeaderNode$ to the last edge of the subgraph is higher.

No. of Nodes	Max. No. of Edges	No. of Graphs in GD	Time taken in Seconds % of FSG			
			25	50	75	100
SPARSE GRAPH DATA SET						
100	1237	100	0.60	0.28	0.16	0.09
		500	13.95	7.03	4.29	3.57
		1000	55.69	27.96	18.24	14.06
500	31187	100	22.69	15.71	13.58	12.19
		500	360.31	185.48	115.40	97.94
		1000	1442.38	724.17	472.42	364.16
1000	124875	100	748.76	349.88	199.93	115.57
		500	2350.98	1098.58	627.76	369.27
		1000	10698.17	4992.48	2846.56	1601.19
NON-SPARSE GRAPH DATA SET						
100	3712	100	1.87	1.04	0.60	0.55
		500	42.02	21.20	12.85	10.77
		1000	167.14	83.93	54.81	42.30
500	93562	100	894.74	721.4	390.85	152.28
		500	14208.18	12238.86	7236.77	6664.08
		1000	56877.67	35548.54	10156.72	2901.92
1000	374625	100	4673.28	2610.7	1635.87	984.51
		500	80130.12	48283.96	24164.08	16736.67
		1000	280455.42	165364.06	103223.52	42681.72
COMPLETE GRAPH DATA SET						
100	4950	100	2.47	1.38	0.54	0.88
		500	56.08	28.44	17.25	14.39
		1000	222.94	111.94	73.10	56.47
500	124750	100	1221.74	943.47	585.02	398.43
		500	22493.71	19631.32	17898.77	8523.22
		1000	78728.83	43982.58	25423.46	23324.28
1000	499500	100	7223.73	4218.79	2509.11	1384.51
		500	93908.51	51038.29	28673.20	26305.69
		1000	328679.77	194484.47	113072.37	60982.83

Table 2: Analysis of FP-GraphMiner on Synthetic Datasets Consisting of Sparse, Non-Sparse, and Complete Graphs

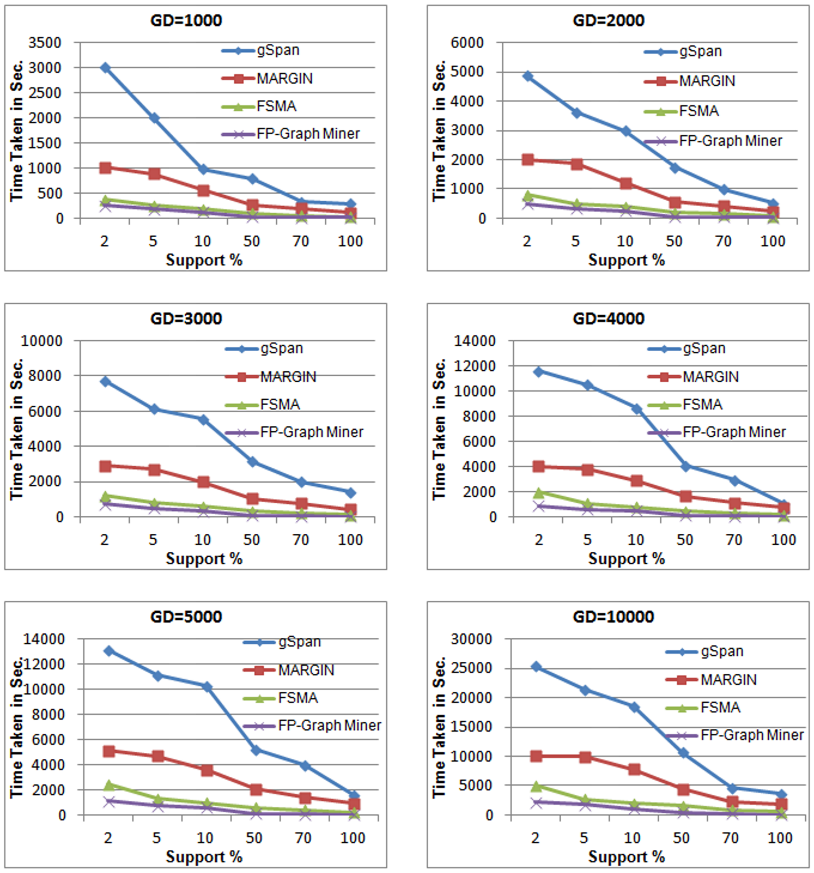


Figure 7: Run Time comparison of FP-GraphMiner with *MARGIN*, *gSpan* and *FSMA*

A comparative study of the FP-GraphMiner algorithm was conducted against the *MARGIN* [17, 18], *gSpan* [22] and *FSMA* [20] algorithms relative to their performance on the real time network data from a large enterprise network using a dataset created through the *Wire Shark Network Monitoring Tool*[2]. The network data uses static IP addresses which are unique, hence, were suitable for our experiments. Each network graph was generated by taking the aggregate at an interval of ten minutes. Six data sets are collected with 1,000, 2,000, 3,000, 4,000, 5,000 and 10,000 time series graphs. In these experiments, parameter settings were minimum support values of 2%, 5%, 10%, 50%, 70% and 100%, average number of edges, $|E_{avg}| = 50$ and number of Node, $|V_{avg}| = 50$. Figure 7

[2] An interesting additional point of reference is the quantitative performance comparison of the *MoFa*, *gSpan*, *FFSM*, and *Gaston* algorithms [19] in which the *gSpan* algorithm performed well.

shows the run time comparison of the FP-GraphMiner with *MARGIN*, *gSpan* and *FSMA* algorithms under different support values for each data set.

From this analysis, it is clear that the FP-GraphMiner algorithm performs well in comparison to *MARGIN*, *gSpan* and *FSMA*. As the support increases, the FP-GraphMiner algorithm is relatively more efficient. This is because all the frequent subgraphs are obtained by simple DFS walks from the $Nodes$ in the $Cluster$ having the given support to the $HeaderNode$ respectively. On the other hand, *FSMA* extends the subgraph directly by adding one frequent edge which requires more computational time compared to the proposed algorithm.

The *MARGIN* algorithm recursively invokes its ExpandCut procedure on each newly found cut which can be a time consuming process. FP-GraphMiner required only a graph traversal process without backtracking to find all the frequent subgraphs with any given support. The experimental results show that FP-GraphMiner is approximately 4 times faster than *MARGIN* and 1.5 times than FSMA. It can be observed that with the increase in the size of data set, the time complexity increases slowly compared to *MARGIN* and *FSMA*, because the distinct edges of all the graphs are stored only once in the FP-Graph which reduces the access time.

7 Conclusions and Future Work

The FP-GraphMiner algorithm constructs an FP-Graph that stores the distinct edges in all graphs only once thereby conserving memory space without loss of information. This graph can be efficiently mined to obtain all frequent subgraphs with given support. If the first cluster is a frequent subgraph with 100% support, it can easily be converted into the maximum common subgraph of GD by making it induced. No other common path exists that has more communication paths or nodes than the maximum common subgraph.

The algorithm could be efficiently enhanced for any network to make useful decisions. The FP-Graph constructed could be used for other mining concepts like graph indexing, graph classification (such as selecting discriminating features, transform graphs in to feature representation, learning classification model), and so on.

The BitCode concept also lends itself to identifying temporal cliques and alternatives. For example, a sequence of BitCodes for an edge might indicate that one edge was only used when another was not or that the use of one edge was dependent on another edge being used at the same time. Timestamping the graphs might also provide useful information regarding the use of a communications network at specific points in the day. These are areas for future research.

References

[1] B. Berendt, A. Hotho, and G. Stumme. Towards semantic web mining. In I. Horrocks and J. Hendler, editors, *The Semantic Web, ISWC 2002*, volume 2342 of *LNCS*, pages 264–278. Springer, 2002.

[2] B. Bouqata, C. D. Carothers, B. K. Szymanski, and M. J. Zaki. Understanding filesystem performance for data mining applications. In *6th International Workshop on High Performance Data Mining: Pervasive and Data Stream Mining (HPDM:PDS'03) at the Third International SIAM Conference on Data Mining*, San Francisco, CA, 2003.

[3] D. Conte, P. Foggia, and M. Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *Journal of Graph Algorithms and Applications*, 11(1):99–143, 2007.

[4] D. J. Cook and L. B. Holder, editors. *Mining graph data.* Wiley-Blackwell, Hoboken, NJ, 2007.

[5] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *3rd IEEE International Conference on Data Mining (ICDM'03)*, Melbourne, Florida, 2003. IEEE.

[6] J. Huan, W. Wang, A. Washington, J. Prins, R. Shah, and A. Tropsha. Accurate classification of protein structural families using coherent subgraph analysis. In *Pacific Symposium of Biocomputing (PSB)*, page 411, Big Island, Hawaii, 2004. World Scientific.

[7] 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*, volume 1910 of *LNCS*, pages 13–23. Springer, 2000.

[8] J. M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[9] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *IEEE International Conference on Data Mining*, pages 313–320, San Jose, California, 2001. IEEE.

[10] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *Data Mining and Knowledge Discovery*, 11:243–271, 2005.

[11] W. Lee and S. J. Stolfo. A framework for constructing features and models for intrusion detection systems. *ACM Transactions on Information and System Security*, 3(4):227–261, 2000.

[12] X. T. Li, J. Z. Li, and H. Gao. An efficient frequent subgraph mining algorithm. *Journal of Software*, 18(10):2469–2480, 2007.

[13] R. J. Mooney, P. Melville, L. R. Tang, J. Shavlik, I. de Castro Dutra, D. Page, and V. S. Costa. Relational data mining with inductive logic programming for link discovery. In H. Kargupta, A. Joshi, K. Sivakumar, and Y. Yesha, editors, *Data Mining: Next Generation Challenges and Future Directions*, pages 1–19. AAAI Press, 2004.

[14] P. C. Nguyen, T. Washio, K. Ohara, and H. Motoda. Using a hash-based method for Apriori-based graph mining. In J.-F. Boulicaut, F. Esposito, F. Giannotti, and D. Pedreschi, editors, *Knowledge Discovery in Databases: PKDD 2004*, volume 3202 of *LNCS*, pages 349–361. Springer, 2004.

[15] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. ANF: A fast and scalable tool for data mining in massive graphs. In *8th ACM SIGKDD Internal Conference on Knowledge Discovery and Data Mining (KDD-2002)*, pages 81–90, Edmonton, Canada, 2002. ACM.

[16] G. Stumme, A. Hotho, and B. Berendt. Semantic web mining: State of the art and future directions. *Web Semantics: Science, Services and Agents on the World Wide Web*, 4(2):124–143, 2006.

[17] L. T. Thomas, S. R. Valluri, and K. Karlapalem. Margin: Maximal frequent subgraph mining. In *6th International Conference on Data Mining (ICDM'06)*, pages 1097–1101, Hong Kong, China, 2006. IEEE.

[18] L. T. Thomas, S. R. Valluri, and K. Karlapalem. MARGIN: Maximal frequent subgraph mining. *ACM Transactions on Knowledge Discovery from Data*, 4(3), 2010.

[19] M. Wörlein, T. Meinl, I. Fischer, and M. Philippsen. A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston. In A. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, editors, *9th European Conference on Principles and Practice of Knowledge Discovery in Databases, PKDD 2005*, volume 3721, pages 392–403, Porto, Portugal, 2005. Springer.

[20] J. Wu and L. Chen. A fast frequent subgraph mining algorithm. In *9th International Conference for Young Computer Scientists*, pages 82–87. IEEE, 2008.

[21] X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *Maebashi City, Japan*, page 721, Second IEEE International Conference on Data Mining (ICDM'02), 2002. IEEE.

[22] X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In *Ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 286–295, New York, NY, 2003. ACM.

Algorithms

Table 3: Algorithm Parameters

Parameter	**Meaning**
GD	Graph Database
K	Number of Graphs
E_i	Number of Edges in a Graph i
m	Number of Distinct Edges in k Graphs
S	Source Node Label
D	Destination Node Label
EL	Edge Label
$EA(G_i)$	Edge Array of Graph G_i
DE	Distinct Edge
$BitCode(DE)$	Collection of k bits of DE
$WT(BitCode(DE))$	Weight of BitCode of DE
$MAXWT$	Maximum Weight of BitCode
$FTRow$	Row in Frequency Table
$Node$	Collection of Distinct Edges with same BitCode
$Cluster$	Collection of $Nodes$ with same BitCode Weight
$HeaderNode$	Empty Top $Node$ of FP-Graph
σ	Support
W	DFS walk from a $Node$ in a $Cluster$ to $HeaderNode$

Algorithm 7.1 Frequent Pattern Graph (FP-Graph) Construction

```
1: Input : $GD = EA(G_1), EA(G_2), \ldots, EA(G_k)$
2: Output: FP-Graph
3:
4: //Generate Frequency Table ($FT$)
5: for each $DE_i$ of $k$ graphs do
6:     Construct $FTRow_i \leftarrow \{DE_i, BitCode(DE_i)\}$
7: end for
8: Sort $FT$ in the descending order of $WT(BitCode(DE))$
9:
10: //Construct $Nodes$
11: for each $Node\ i$ do
12:     Construct $Node[i] \leftarrow (\{DE_1, DE_2, ..., DE_x\}, BitCode[i])$,
          $1 \le i \le 2^k, 1 \le x \le m$
13: end for
14:
15: //Group $Nodes$ into $Clusters$
16: for each $WT(BitCode(DE_i))$ in $FT$ do
17:     $Cluster[j] \leftarrow \{(Node[1], Node[2], \ldots)\}$,
          where $WT(Node[1]) = WT(Node[2]) \ldots$,
          $\sigma(Cluster[j]) = \lfloor (WT_j/k) * 100 \rfloor$
18: end for
19:
20: Let X=$Cluster$ with MAXWT
21: $\sigma(Cluster[1]) < \ldots < \sigma(Cluster[j]) \ldots <$
        $\sigma(Cluster[X]), 1 \le j \le X$
22:
23: //Link $Nodes$ in $Clusters$
24: EL($Node[i]$,$Node[j]$)=Edge Label from $Node\ i$ to $Node\ j$
25: $DV_i$=Decimal Equivalent of BitCode of $Node[i]$, $1 \le i \le 2^k$
26:
27: //Link $HeaderNode$ with all $Nodes$ in $Cluster$ with MAXWT
28: for each Node $x$ in $X$, $1 \le x \le k$ do
29:     Link($HeaderNode$) $\leftarrow Node[x]$
30:     $EL$(HeaderNode,Node[x]) $\leftarrow DV_x$
31: end for
32:
33: //Link $Nodes$ in the other $Clusters$
34: Let $Cluster[1] = Cluster$ with least support
35: LABEL1:
36: for each $Node\ x$ in $Cluster[i]$, $1 \le i \le X$ do
37:     LABEL2:
38:     for each $Node\ y$ in $Cluster[j]$, $i + 1 \le j \le X$ do
39:         if $d_{(x \in Cluster[i], y \in Cluster[j])}$ $BitCode(x) \cap BitCode(y) = BitCode(x)$,
            $\sigma(Cluster[i]) < \sigma(Cluster[j])$ then
40:             //$Node[y]$ is immediate superset of $Node[x]$
41:             $Node[x]$.link $\leftarrow Node[y]$
42:             EL($Node[x]$, $Node[y]$)$\leftarrow DV_y$
43:         else
44:             if $(j + 1) \ne HeaderNode$ then
45:                 $j \leftarrow j + 1$
46:                 go to LABEL2
47:             end if
48:         end if
49:     end for
50: end for
51: $i \leftarrow i + 1$
52: go to LABEL1
```

Algorithm 7.2 Frequent Pattern Graph Mining (FP-GraphMiner) Algorithm

```
1: //Find all frequent subgraphs of k graphs
2: Input: FP-Graph Support $\sigma$
3: Output: All FSG in $GD$ with given $\sigma$
4:
5: for each DFS Walk $w$ starting from each $Node\ i$ in $Cluster\ j$ with $\sigma \approx \sigma_j, 1 \leq j \leq m$ do
6:     $FSG_w \leftarrow Nodes$ in $DFS$ path from $Cluster\ j$ up to $HeaderNode$
7: end for
8:
9: //Find graphs in $GD$ containing the Query Graph $Q$
10: Input: FP-Graph $Q = QueryGraph$
11: Output: Graphs in $GD$ containing Query Graph $Q$, support $\sigma$ of $Q$ in $GD$.
12:
13: //Starting from Header, perform BFS and collect all edges of $Q$ from various $Nodes$.
14: $BFS(Q) = Node[1], Node[2], \ldots, Node[q], 1 \leq q \leq 2^k$,
15: Collection of $Nodes$ containing edge(s) of $Q$ //Perform AND operation on the $Node(s)$
    in which edges of Q are present to obtain the Query Graph Code of Q.
16: $QGraphCode(Q) = \cap_{i=1}^{2^k} BitCode(Node_i)$
17: Graphs containing Q = positions of 1's in QGraphCode(Q)
18: $\sigma(Q) \leftarrow (Q)$ =Number of graphs containing $Q/k * 100$
```