CHAPTER 5

FP-GROWTH BASED FREQUENT SUBGRAPH MINING WITH
GRAPHGAIN

5.1 Introduction

One of the two major trends in frequent pattern mining is the Pattern-growth approach or Frequent Patter Growth (FP-Growth) Approach [110] [22] [83]. The key difference between the Apriori with that of Patter-growth is, how they generate candidate subgraphs. By reducing the size of the candidate set, the Apriori heuristic yields good performance gain. But, in situations like prolific frequent patterns, long patterns, or quite low minimum support thresholds, an Apriori-like algorithm may still suffer due to some nontrivial costs. So, by avoiding the generation of a huge set of candidates, the mining performance can be substantially improved. This approach is handled by FP-Growth algorithm and earns the same achievements of Apriori without candidate generation [43].

The FP-growth approach requires the creation of an FP-tree. Mining the FP-tree, which is created for a normal transaction database, is easier compared to large document graphs, mostly because the itemsets in a transaction database is smaller compared to the edge list of our document-graphs. Original FP-tree mining procedure is also easier because the items of a traditional transaction database are stand-alone entities and have no direct connection to each other. In contrast, as in the case of subgraphs in graphs, they become related to each other in the context of connectivity. Hence, this chapter proposes the modification on FP-growth approach and making it possible to generate frequent subgraphs from the FP-tree.
By modifying the FP-Growth approach suitably and applying the proposed novel graphgain concept, the modified FP-Growth algorithm can be extended to rank the frequent subgraphs with their ranking and thus by obtaining the better performance of a graph database. This ranking algorithm named as FP-Growth Based Graphgain (FPGBG) will provide the substantial and essential techniques in improving the performance of the frequent subgraph mining and ranking.

5.2 Existing Work

Recent research on pattern discovery has progressed from mining frequent itemsets and sequences to mining complicated structures, including trees and graphs [83]. The pattern growth approach [47] [62] avoids two costly operations of Apriori approach [52]: joining two $k$ edge graphs to produce a new $(k+1)$ edge candidate graph and checking the frequency of these candidates separately. Moreover, efficient data structures have been developed for effective database compression and fast in-memory traversal [42]. One algorithm, Diagonally Subgraphs Pattern Mining (DSPM), has been developed that uses some features from both Apriori and FP-growth approach [22]. A number of pattern-growth algorithms have been invented and used for frequent subgraph mining. Among these, Molecular Fragment Miner (MoFa) [11] is extensively used in bioinformatics. Graph–Based Substructure Pattern Mining (gSpan) [5ad] uses canonical labeling and the right-most extension technique. GrAph/Sequence/Tree 10 extraction (Gaston) [76] begins by searching for frequent-path, then frequent free trees and finally cyclic graphs. Yan et al. presented another new method: Closed graph pattern mining (CloseGraph) [5af].

The existing FP-Growth algorithm [43], propose a novel FP-tree structure, which is an extended prefix-tree structure for storing compressed, crucial information about frequent patterns, and develop an efficient FP-tree-based mining method, FP-growth, for mining the complete set of frequent patterns by pattern fragment growth.

In the FP-Growth Pattern Mining, efficiency of mining is achieved with three techniques: (1) a large database is compressed into a highly condensed, much smaller data structure, which avoids costly, repeated database scans, (2) our FP-tree-based mining adopts a pattern fragment growth method to avoid the costly generation of a
large number of candidate sets, and (3) a partitioning-based, divide-and-conquer method is used to decompose the mining task into a set of smaller tasks for mining confined patterns in conditional databases, which dramatically reduces the search space. The performance study shows that this FP-growth method is efficient and scalable for mining both long and short frequent patterns, and is about an order of magnitude faster than the Apriori algorithm and also faster than some recently reported new frequent pattern mining methods.

The problem of FP-Growth technique is approached in the following three aspects. In the first approach, a novel, compact data structure, called frequent pattern tree (FP-tree), is constructed, which is an extended prefix-tree structure for storing crucial, quantitative information about frequent patterns. Only frequent length-1(initial suffix) items will have nodes in the tree, and the tree nodes are arranged in such a way that more frequently occurring nodes will have better chances of sharing nodes than less frequently occurring ones.

Second, an FP-tree-based pattern fragment growth mining method, is developed, which starts from a frequent length-1 pattern (as an initial suffix pattern), examines only its conditional pattern base (a “sub-database” which consists of the set of frequent items co-occurring with the suffix pattern) and constructs its (conditional) FP-tree. This also performs mining recursive with such a tree. The pattern growth is achieved via concatenation of the suffix pattern with the new ones generated from a conditional FP-tree. Since the frequent itemset in any transaction is always encoded in the corresponding path of the frequent pattern trees, pattern growth ensures the completion of the result. So this method becomes a test only and not like generation in Apriori. The major operations of mining are count accumulation and prefix path count adjustment, which are usually more cheeper than candidate generation and pattern matching operations performed in most Apriori-like algorithms.

Third, the search technique employed in mining is a partitioning-based, divide-and-conquer method rather than Apriori-like bottom-up generation of frequent itemsets combinations. This dramatically reduces the size of conditional pattern base generated at the subsequent level of search as well as the size of its corresponding conditional FP-tree. Moreover, it transforms the problem of finding long frequent
patterns to looking for shorter ones and then concatenating the suffix. It employs the least frequent items as suffix, which offers good selectivity. All these techniques contribute to substantial reduction of search costs.

The advantages and disadvantages of FP-Growth approach is depicted as follows:

**Advantages**

Long pattern of transaction cannot be broken by this method. It conserves complete information for frequent pattern mining. This method reduces irrelevant information. The frequency descending ordering is more likely to be shared. It does not make transaction set larger than the original database. Finally, it is much faster than Apriori algorithm.

**Disadvantages**

This algorithm generates frequent pattern tree and that may not fit in memory. Building frequent pattern tree is expensive. When support is high, time is wasted, as the only pruning that can be done is on single items. The support can be calculated only when the entire dataset is added to the FP-Tree.

**5.3 Basic Concept**

This method mines the complete set of frequent subgraphs without candidate generation. The Divide-and-conquer policy method is adopted for this technique. In section 2.3.2, the basics of FP-Growth mining rules are already discussed. This method follows two phases which are as follows:

**Phase I**: During the first scan, a list of frequent items are collected based on the frequency descending order. Then the database is compressed into a frequent pattern tree (FP-tree).
Phase II: In the second phase the FP-tree is mined by starting from each frequent length-1 pattern, constructing its conditional pattern base, then constructing its conditional FP-tree, and performing mining recursively on such a tree. The pattern growth is achieved by the concatenation of the suffix pattern with the frequent patterns generated from a conditional FP-tree.

5.4 Proposed Algorithm

In this section a new algorithm, “FP-Growth based Frequent Subgraph Mining with Graphgain” (FPGBG) is presented for finding all connected subgraphs that appear frequently in a large graph database and the ordering of them. Since the amount of text data stored in computer repositories is growing every day, it is needed more than ever a reliable way to group or categorize text documents. Most of the existing document clustering techniques uses a group of keywords from each document to cluster the documents. In this thesis, a sense based approach to cluster documents is used instead of using only the frequency of the keywords. This method uses a relationship between the keywords to cluster the documents. The relationships are retrieved from the WordNet ontology and represented in the form of a graph [74]. The document-graphs, which reflect the essence of the documents, are searched in order to find the frequent subgraphs. To discover the frequent subgraphs, the Frequent Pattern Growth (FP-growth) approach is used, which was originally designed to discover frequent patterns. The common frequent subgraphs discovered by the FP-growth approach are later ranked by the proposed measure.

5.4.1 FP-Growth with GG

FP-tree creation is required by the FP-growth approach [43]. Compared to large document graphs, mining of FP-tree is easier. This is due to the fact that, itemsets in a transaction database is smaller compared to the edge list of document-graphs. In original FP-tree mining procedure, it discovers frequent patterns (i.e. association rules) and there is no direct connection between the transactions. In contrast, they become related to each other in the context of connectivity of the subgraph. This thesis has addressed the problem and also the discovery of frequent
subgraphs by modifying FP-growth approach. This work has also addressed the ranking of subgraphs from the above method.

In this thesis, the Frequent Pattern growth (FP-growth) approach [43] is modified so that it can discover frequent subgraphs. Originally, this algorithm was designed to mine frequent itemsets in the domain of market basket analysis [56], which examines the trend of consumers’ choices during purchases. Unfortunately, it was not designed with graph mining in mind and does not efficiently mine frequent subgraphs. The necessary changes are made to this algorithm so it as to perform graph mining. In these models, a document is represented as a vector whose elements are the keywords with their frequencies. If there are thousands of documents, then large amount of information about the keywords have to be stored. Also, there is no way to keep the information concerning the relationship between the keywords in each document. Many words have multiple meanings, and once they are stored as individual units, it is hard to identify the specific meaning of a keyword in that document. In the BoW representation of the documents, the overall concept of a document does not become apparent. Therefore, it may provide low quality clustering. The proposed approach has three major steps:

1. Transforming documents into graphs,
2. Discovering frequent subgraphs in the document-graphs, and
3. Ranking the discovered frequent subgraphs.

This work uses WordNet’s IS-A relationship to create a graph to represent the relationship among the keywords in a document. The graph, called the document-graph (DG), is the hierarchical representation of the keywords appearing in a document. Such hierarchical document-graphs for every document in the archive were constructed. In the second stage of this process, these document-graphs are searched for frequent subgraphs using the FP-growth approach. In the context of document-graphs, frequent subgraphs hold the key concepts that appear frequently in the document archive. After the generation of frequent subgraphs, the resultant graphs need not contain all of the original keywords. Instead, this work focuses on the hierarchy of concepts related to those keywords. Therefore, the process of frequent subgraph mining in documents can also be viewed as a process of frequent concept
The overall procedure followed in graph-based text document clustering is shown in Figure 5.1.

![Figure 5.1: FP-Growth based Graph-document ranking approach.](image)

5.4.1.1 Description of the Algorithm of FP-tree Creation

The algorithm that creates the Frequent Pattern Tree (FPT) is shown in Figure 5.2. This algorithm first scans the Transaction Data Base (TDB) to find the edges that appear in the DG database. These edges and their corresponding supports are added to a list called $F$. Then the hash table is created and is called transactionDB for all the DGs. Hash table is a data structure that associates keys with values. Given a key it finds the corresponding value without linearly searching the entire list. The document ids are stored as the keys of our hash table. A value against a key stores a vector consisting of all the (Depth First Search) DFS codes of the edges for the DG in the corresponding key. Based on the minimum support ($min\_sup$), provided by the user the list $F$ is pruned and sorted in descending order. This newly created frequent edge list is denoted as the $FEList$. Thus $FEList = \{(e_i, \beta_i) | e_i \in MDG, \beta_i \geq min\_sup \text{ and } \beta_i \geq \beta_{i+1}\}$. Based on the sorted order of $FEList$, transactionDB is also altered so that all the infrequent edges are removed from each entry and the most frequent edges appear at the beginning.
After the sorting of $FEList$ and $transactionDB$, FP-tree is then created. Initially the root of the FP-tree is an empty node referred to as null. Then call the $insert_tree([p| Pi], FPT)$ method to insert the sorted edges of DG$_i$ in the FP-tree, denoted as FPT, where $p$ is the first edge of DG$_i$ and $P_i$ is the remaining edge list of DG$_i$. Every time $insert_tree([p| Pi], FPT)$ is called, it checks whether FPT has a child $N$ (node for an edge) from root that is identical to $p$. This check is performed by selecting the root node of the existing FP-tree and listing its outgoing nodes. If any of the outgoing nodes of the root contains $p$, then the child $N$’s (node containing $p$) count is increased by one. At this point, the pointer to the root node is changed to $N$ (node containing $p$) and the next edge of $P_i$ is selected for addition in T following the same procedure. This continues until there are no more edges left in $P_i$ to add in FPT or until there is a root that does not contain $p$ as its outgoing node indicating that FPT does not have a child $N$. In such case, the program creates a new node $N$ with the count of $N$ set to one. This new node $N$ points to the root node as its parent, and its node-link points to the nodes with the same DFS edge id. This method continues recursively as long as there are edges remaining in the $P_i$ list of DG$_i$. The same edge can appear in different branches (at different levels) of the FP-tree, because different DG$_i$ can contain them. Thus, for every edge a list of node links are maintained to point out where they appear in the FP-tree. $FEList$ holds the edges with their corresponding support and a pointer to an edge in the FP-tree. The edge in the FP-tree further creates a chain of pointers if there are other edges with the same DFS code.

<table>
<thead>
<tr>
<th><strong>Input</strong></th>
<th>DG database DB and Minimum support threshold, $min_sup$.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Output</strong></td>
<td>Frequent Pattern Tree (FPT) made from each DG$_i$ in the DB.</td>
</tr>
<tr>
<td><strong>Step 1.</strong></td>
<td>Scan the DB once.</td>
</tr>
<tr>
<td><strong>Step 2.</strong></td>
<td>Collect the set of edges $F$ with the corresponding support.</td>
</tr>
<tr>
<td><strong>Step 3.</strong></td>
<td>Sort $F$ in descending order and create the list of frequent edges $FEList$.</td>
</tr>
<tr>
<td><strong>Step 4.</strong></td>
<td>Create $transactionDB$</td>
</tr>
<tr>
<td><strong>Step 5.</strong></td>
<td>Create the root of an FP-tree FPT, and label it as “null”.</td>
</tr>
</tbody>
</table>
Step 6. For each DG\textsubscript{i} in the transactionDB do the following:

Step 7. Select and sort the frequent edges in DG\textsubscript{i} according to FE\textit{List}.

Step 8. Let the sorted frequent-edge list in DG\textsubscript{i} be \([p|P_i]\), where \(p\) is the first element and \(P_i\) is the remaining list.

Step 9. Call \textit{insert\_tree}(\([p|P_i]\), \(\text{FPT}\))

\textbf{Figure 5.2} : Creation of FP-tree

Step 1 : If \(N\).item-name = \(f\).item-name (ie, if \(FPT\) has a child node \(N\)) then Compute \(N\).count = \(N\).count + 1 go to step 4 else go to step 2

Step 2 : Create a new node \(N\) with \(N\).count = 1

Step 3 : link its parent \(FPT\) to \(N\), and its node-link to the nodes with the same item-name

Step 4 : If \(L\) is nonempty, call \textit{insert\_tree}(\(L;N\)) recursively.

\textbf{Figure 5.3} : Insert\_tree(\([fjL];K\))

From the FP-tree creation process, one needs exactly two scans of the transaction database, DB: the first collects the set of frequent items, and the second creates the FP-tree. The cost of inserting a transaction \(FPT\) into the FP-tree is \(O(|\text{Trans}|)\), where \(|\text{Trans}|\) is the number of frequent items in \(FPT\). We will show that the FP-tree contains the complete information for frequent pattern mining.
5.4.1.2 Description of the FPGBG Algorithm

To convert the existing FP-growth algorithm suitable for graph mining, the algorithm is changed so that it discovers frequent connected subgraphs and performs better. The hash table called *transactionDB* for all the *Document Graphs* (DG) which is similar to original FP-growth procedure is the starting process of this algorithm (Figure 5.2). Next, the *headerTable* (same as the FEList) is created from all the edges appearing in the Master Document Graph (MDG). After creating *transactionDB* and *headerTable*, the method *FilterDB* is called with *transactionDB*, *headerTable* and minimum support as parameters. Depending on the minimum support provided by the user, this method reconstructs both of the lists (i.e. *transactionDB* and *headerTable*)

<table>
<thead>
<tr>
<th>Input</th>
<th>Document graphs’ database <em>DB</em></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Master Document graph <em>MDG</em></td>
</tr>
<tr>
<td></td>
<td>Minimum support <em>min_sup</em></td>
</tr>
<tr>
<td>Output</td>
<td>Frequent subgraphs <em>FS</em></td>
</tr>
</tbody>
</table>

1. Create a *transactionDB* for all *DG* ∈ *DB*
2. Create *headerTable* for all edge *a* ∈ *MDG*
3. *FilterDB*(transactionDB, headerTable, min_sup)
4. *FPTreecreation()*
5. *FPSGMining()*
6. For each subgraph *FS* do
7. include document ids for all Subgraphs
8. Compute the Lift value for each subgraph such that
   \[
   Lift[FS^i] = \left(\frac{n_{of \, graph}}{n_{of \, F_h \times number \, of \, F_h}}\right) = \frac{P(AUB)}{P(A) \cdot P(B)}
   \]
9. Calculate the MDCG value for all the subgraphs such as
   \[
   MDCG[FS^i] = lift[FS^i] + \sum_{i=2}^{p} \frac{lift[FS^i]}{\log_2 i}
   \]
10. Then calculate the value of IMDCG for every subgraph (IMDCG[FS^i]) by considering the descending ordering of the lift values of all subgraphs and follow the calculation method in the above step.
11: Evaluate the Graphgain (new normalized values) values at each point (for each subgraph) as follows:

\[
Graphgain = GG[F_S^i] = \frac{MDCG[F_S^i]}{IMDCG[F_S^i]}
\]

12: Set the value 1 for i and i+1 for j
13: If FS\(^i\) is less than FS\(^n\) do
14: If FS\(^j\) is less than FS\(^n\) do
15: if GG[FS\(^i\)] > GG[FS\(^j\)] go to step 26
16: Temp = GG[FS\(^i\)]
17: GG[FS\(^i\)] = GG[FS\(^j\)]
18: GG[FS\(^j\)] = Temp
19: increment j by 1; if j > n then go to step 27 else go to step 21
20: increment i by 1; if i < n compute j = i+1 and go to step 20
21: GG[FS\(_1\)], GG[FS\(_2\)], …… GG[FS\(_n\)]

**Figure 5.4**: FPGBG algorithm

by removing infrequent edges and sorting them in descending order by frequency. Before constructing the FP-tree, we prune the header table at top and bottom for a second time to reduce too specific and abstract edges. transactionDB is updated to reflect this change in the header table. After this refinement, FP-tree is created by calling FPTreeConstructor() method. Later, the method FPSGMining() generates the frequent subgraphs by traversing the FP-tree. For each frequent subGraph, a list of document ids is created in which the subGraph appears.
**Input**  FP-tree FPT
   FList headerTable
Frequent pattern FP (initially, FP =null)

**Output**  Frequent subgraphs FSG
1. If FPT contains a single path and  If (FPT.length> SPThreshold)
2. Delete ( FPT.length - SPThreshold ) {delete number of edges from the top of FPT}
3. for each combination of the nodes in FPT do
4. If (isConnected(FSG, FP) == true) then
5. generate FSG∪ FP, support = MIN (support of nodes in FSG)
6. else for each ai in the headerTable of FPT do
7. create pattern FSG = ai∪ FP with support = ai.support;
8. if TreeFSG ≠ ∅ then
9. call FPSGMining(TreeFSG, headerTableFSG, FSG);

**Figure 5.5 :** FPSG Mining Algorithm

### 5.5 Description with sample dataset

Since the output of the FP-Growth method is again a subgraphs list, the calculation of Lift, MDCG, IMDCG and GG are applied to this algorithm and the ranking order of the subgraphs obtained are found out. Since the main purpose of GG is the ranking of subgraphs mined and in this experimental analysis, the application of GG will be a resemblance of the same calculations done in the previous chapter. Only change is, the rules and their lift values. Hence, the worked out procedure for this sample data set is not discussed.
5.6 Experimental Study and Analysis

The FPGBG algorithm has been implemented in Java. All experiments reported in this chapter have been performed on Intel Pentium(R) Dual core E5400 @ 2.70 GHz with 1.99 GB main memory, running Microsoft Windows XP. In this FPGBG analysis, a novel frequent pattern tree (FP-tree) structure, which is an extended structure for storing compressed, crucial information about the frequent patterns, is constructed. An efficiency of mining is achieved with the following three techniques:

(1) A large database is compressed into much smaller and highly condensed data structure, to avoid the cost and repeated database scans.

(2) FP-tree-based mining adopts a pattern fragment growth method to avoid the costly generation of a large number of candidate sets, and

(3) A partitioning-based, divide-and-conquer method is used to decompose the mining task into a set of smaller tasks in conditional databases, which dramatically reduces the search space.

The performance study shows that the FPGBG method is an efficient and scalable for mining and ranking of both long and short frequent patterns, and is about an order of magnitude faster than the Apriori algorithm. The figures 5.6 and 5.7 illustrates the comparison of execution time between FP-Growth method and the proposed FPGBG method according to the user’s specified support in two different data sets. The first one is with the chemical data set and the second one with synthetic data set. These comparisons show that there is a small amount of increase in execution time for both the methods and with the threshold value less than 80. This analysis also conclude that beyond certain threshold value (for eg. Greater than 80 in this study), the proposed FPGBG algorithm shows it efficiency over that of FP-Growth even though there is an addition of ranking of subgraphs involved. Hence, the average performance of the proposed algorithm is efficient than the existing FP-Growth method in the view of user’s specified support.
Figure 5.6: Comparison of execution time between FP-Growth and proposed FPGBG for user’s specified support in chemical data set
The figures 5.8 and 5.9 compare the execution time between FP-Growth method and the proposed FPGBG by considering the number of subgraphs in both the above mentioned data sets. These figures also illustrates that the overall performance of the proposed algorithm is well doing when compared with the existing method. There is a gradual increase in execution time in both the cases. This uniform slight increase in the execution time is due to ranking method included in the proposed algorithm. The experimental study is made with the chemical data set of 340 graphs and synthetic data set with 1000 graphs. The analysis also shows that ranking of subgraphs even in a large data set will not take much amount of time.
Figure 5.8: Comparison of execution time between FP-Growth and proposed FPGBG for the number of subgraphs in chemical data set
Figure 5.9: Comparison of execution time between FP-Growth and proposed FPGBG for the number of subgraphs in synthetic data set
5.7 Summary

FPGBG have proposed a novel and efficient mining of frequent patterns in large databases. There are several advantages of FPGBG over other approaches: (1) it constructs a highly compact FP-tree, which reduces the database size and scanning process. (2) it applies a pattern growth method which avoids costly candidate generation. In this context, mining is not Apriori-like (restricted) generation-and-test but frequent pattern (fragment) growth only. (3) it applies a partitioning-based divide-and-conquer method which dramatically reduces the size of the subsequent conditional pattern bases and conditional FP-trees. The GG method is implemented in the FP-growth method, to study its performance of finding out the ranking of the generated frequent subgraphs. This performance study shows that the addition of ranking of both short and long patterns efficiently in large databases will not cause the efficiency of the algorithm to the worst side but the deviations are considerable due to the modification and the quality.