FP-GROWTH APPROACH FOR DOCUMENT CLUSTERING

by

Monika Akbar

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2008

APPROVAL

of a thesis submitted by

Monika Akbar

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citation, bibliographic style, and consistency, and is ready for submission to the Division of Graduate Education.

Dr. Rafal A. Angryk

Approved for the Department Computer Science

Dr. John Paxton

Approved for the Division of Graduate Education

Dr. Carl A. Fox

## STATEMENT OF PERMISSION TO USE

Monika Akbar

April, 2008

To
Family,
Friends
&
Well-wishers

# TABLE OF CONTENTS

LIST OF TABLES

# LIST OF FIGURES

LIST OF FIGURES – CONTINUED

# ABSTRACT

Since the amount of text data stored in computer repositories is growing every day, we need more than ever a reliable way to group or categorize text documents. Most of the existing document clustering techniques use a group of keywords from each document to cluster the documents. In this thesis, we have used a sense based approach to cluster documents instead of using only the frequency of the keywords.

We use relationships between the keywords to cluster the documents. The relationships are retrieved from the WordNet ontology and represented in the form of a graph. The document-graphs, which reflect the essence of the documents, are searched in order to find the frequent subgraphs. To discover the frequent subgraphs, we use the Frequent Pattern Growth (FP-growth) approach, which was originally designed to discover frequent patterns. The common frequent subgraphs discovered by the FP-growth approach are later used to cluster the documents.

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 we look for subgraphs in graphs, they become related to each other in the context of connectivity. The computation cost makes the original FP-growth approach somewhat inefficient for text documents.

We modify the FP-growth approach, making it possible to generate frequent subgraphs from the FP-tree. Later, we cluster documents using these subgraphs.

CHAPTER 1

INTRODUCTION

Organizations and institutions around the world store data in digital form. As the number of documents grows, we need a robust way to extract information from them. It is vital to have a reliable way to cluster massive amounts of text data. In this thesis we present a new way to mine documents and cluster them using graphs.

Graphs are data structures that have been used in many domains, such as natural language processing [1], bioinformatics [2] and chemical structures [3]. Nowadays, their role in data mining and database management is increasing rapidly. The domain of graph mining includes scalable pattern mining techniques, indexing and searching graph databases, clustering, classification and various other applications and exploration technologies. Graph mining focuses mainly on mining complicated patterns from graph databases. This can be very expensive, as subgraph isomorphism [4] has very high time and space complexity compared to other operations of different data structures [5].

The problem of frequent subgraph mining is to find the frequent subgraphs in a collection of graphs. A subgraph is frequent if its support (occurring frequency) in a given graph database is greater than a minimum support. Currently, there are two major trends in frequent subgraph mining: the Apriori-based approach and the Pattern-growth approach [3] [6] [7]. The key difference between these two approaches is how they generate candidate subgraphs.

In this thesis, we have utilized the Frequent Pattern growth (FP-growth) approach [8] that discovers frequent patterns (i.e. association rules), and modified this approach so that it can discover frequent subgraphs. Originally, this algorithm was designed to mine frequent itemsets in the domain of market basket analysis [9], 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. We made the necessary changes to this algorithm so it can be used for graph mining.

Using an algorithm for association rule mining [10] to cluster documents based on their concept is a fairly new approach. Most of the current document mining techniques depend on the frequency of the keywords or the bag-of-words (BoW) approaches [11] to cluster documents. In these models, a document is represented as a vector whose elements are the keywords with their frequencies. In most cases, this is not sufficient to represent the concept of the document and can give an ambiguous result. If we have thousands of documents, we have to store a large amount of information about the keywords. 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.

## Scope and Objectives

In this thesis, we mostly focus on retrieving the senses of the documents to cluster them. Sense-based document clustering is more human-like and we believe that it provides more accurate clustering. Our approach has three major steps:

1. Transforming documents into graphs,

2. Discovering frequent subgraphs in the document-graphs, and

3. Clustering documents using the discovered frequent subgraphs.

We have used the 20NG [12] dataset as our benchmark corpora, which consists of 19,997 articles divided into 20 categories. From these articles, a set of keywords is selected using the Bow [13] toolkit. We use WordNet [14] to retrieve part of speech information about the keywords. In this work, we have concentrated only on the noun keywords.

Once we have the noun keywords, we use WordNet's IS-A relationship to create a graph to represent the relationship between the keywords in a document. The graph, called the *document-graph*, is the hierarchical representation of the keywords appearing in a document. We construct such hierarchical *document-graph*s for every document in the archive. In the second stage of this process, these *document-graph*s are searched for frequent subgraphs using the FP-growth approach. As we stated earlier, FP-growth has been developed primarily for discovering frequent itemsets. These itemsets are stand-alone entities which do not need to maintain properties of graphs. We have made necessary modifications to the FP-growth algorithm so that it can be used in the graph-mining domain to discover frequent subgraphs. To make our FP-growth approach faster,
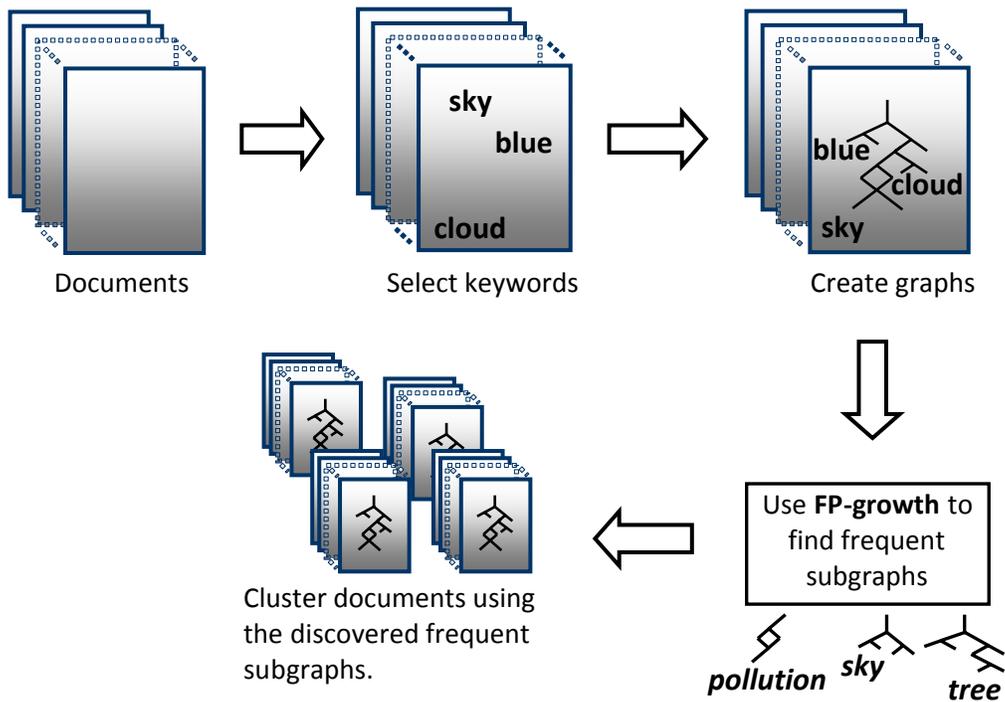
Figure 1: Graph-based document clustering approach.

we do not generate all possible frequent subgraphs; rather we generate subgraphs that are important for clustering. Subgraphs found at the top levels of the ontology are not good candidates for clustering because they appear in most of the documents. Similarly, subgraphs found at the bottom levels of the hierarchy are too specific to perform accurate clustering. Our FP-growth approach generates most of the subgraphs from the mid-levels of the ontology.

In the context of *document-graph*s, frequent subgraphs hold the key concepts that appear frequently in the document archive. After the generation of frequent subgraphs, the resultant graphs do not have to contain all of the original keywords. Instead, we focus 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 mining. The overall procedure followed in our graph-based text document clustering is shown in Figure 1.

## Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 describes the background literature. We give an overview of FP-growth and explain the methodology along with the details of the modification in Chapter 3. The results and their comparisons comprise Chapter 4. Finally, we conclude in Chapter 5 by describing the outcome and the future direction of this research.

CHAPTER 2

BACKGROUND

In this chapter, we will explain some of the key terms that have been used frequently in the thesis. We start by describing the principle of the WordNet [14] ontology. We use the WordNet ontology to create the *document-graph*s for each document. The *document-graph*s are used to discover frequent subgraphs using association rule mining [10]. Thus, we also explain the idea behind association rule mining. This is followed by a section that describes a specific approach of graph mining (pattern-growth). At the end, we give a brief overview of text mining.

WordNet

WordNet [14] is a semantic lexicon for the English language. We have utilized it to retrieve the semantic relations between noun keywords of the documents. The benefit of WordNet is that it produces a combination of a dictionary and a thesaurus that is more intuitively usable. It divides the lexicon into five categories: nouns, verbs, adjectives, adverbs, and function words. WordNet groups words into sets of synonyms called *synsets* and also provides short, general definitions and examples. Every synset contains a group of synonymous words or collocations; different senses of a word appear in different synsets. Most synsets are connected to other synsets via a number of semantic relationships. These relationships vary based on the type of word.

In WordNet nouns are organized in lexical memory as topical hierarchies, verbs are organized by a variety of entailment relations, and adjectives and adverbs are organized as *N*-dimensional hyperspaces. Each of these lexical structures reflects a different way of categorizing experience. Attempts to impose a single searching principle on all categories will badly misrepresent the psychological complexity of lexical knowledge. This is why we chose only one part of speech, the noun, for this thesis. Also, since WordNet has different kinds of semantic relations among nouns, it might become ambiguous if we try to incorporate all of them into a single graph. This is why we only use the hypernymy relation (IS-A) between nouns. Definitions of common nouns in WordNet typically give a super-ordinate term along with its distinguishing features. The distinguishing features are entered in such a way that it creates a lexical inheritance system, a system in which each word inherits the distinguishing features of all its super-ordinates.

We transform the documents into graphs using the WordNet ontology of the keywords that the documents contain. Later we utilized the FP-growth approach, which is a special kind of association rule mining algorithm, to discover the frequent subgraphs from the *document-graph*s.

## Association Rule Mining

The aim of association rule mining [10] is to discover frequent patterns that appear together in a large database. In the context of market basket analysis, let $I = \{i_1, i_2, \ldots, i_d\}$ be the set of items and $T = \{t_1, t_2, \ldots, t_N\}$ be the set of transactions made

by consumers. Each transaction $t_i$ contains a subset of the items from the set *I*. A set of items in a transaction is called an *itemset*. The rules are generated depending on the itemsets that appear frequently. The quality of an association rule can be determined by the *support* and *confidence* count. Support of a rule $X \Rightarrow Y$ is the percentage of transactions that contains both the items *X* and *Y*. Confidence $X \Rightarrow Y$ is the number of transactions that contain *X* and *Y* divided by the total number of transactions that contain *X*. Usually an association rule is said to be interesting if it satisfies a minimum support threshold and a minimum confidence threshold.

Two most popular approaches to association rule mining are Frequent Pattern growth (FP-growth) [8] and Apriori [15]. Apriori algorithm uses prior knowledge of frequent itemsets to generate the candidate for larger frequent itemsets. It relies on the relationship between itemsets and subsets. If an itemset is frequent, then all of its subsets must also be frequent. But generation of the candidates and checking their support at each level of iteration can become costly. FP-growth introduces a different approach here. Instead of generating the candidates, it compresses the database into a compact tree form, known as the FP-tree, and extracts the frequent patterns by traversing the tree.

We have developed an FP-growth-based approach for document clustering. As in our system the documents are represented by graphs, we made the necessary changes to the FP-growth algorithm so that instead of generating frequent itemsets it can generate frequent subgraphs. Details of the FP-growth algorithm are described in Chapter 3. Since we are using graphs to discover the frequent subgraphs, we will now present a review of

pattern-growth approach of graph mining. We have used DFS coding in our *document-graph*s inspired by one of these approaches (i.e. gSpan [3]).

## Pattern Growth

Recent research on pattern discovery has progressed from mining frequent itemsets and sequences to mining complicated structures, including trees and graphs [7]. As a general data structure, graphs can model arbitrary relationships among objects. Graph mining focuses on mining frequent subgraph sets [16] and special path mining [17]. The pattern growth approach [18] [19] avoids two costly operations of Apriori approach [20]: joining two $k$ edge graphs to produce a new $(k+1)$ edge candidate graph and checking the frequency of these candidates separately. These two operations are the performance bottleneck of the Apriori algorithms. Pattern growth mines frequent patterns without generating candidates. The method adopts a divide-and-conquer approach to partition databases based on the currently discovered frequent patterns. Moreover, efficient data structures have been developed for effective database compression and fast in-memory traversal [21]. One algorithm, Diagonally Subgraphs Pattern Mining *(DSPM)*, has been developed that uses some features from both Apriori and FP-growth approach [6].

A number of pattern-growth algorithms have been invented and used for frequent subgraph mining. Among these, Molecular Fragment Miner (MoFa) [2] is extensively used in bioinformatics. Graph–Based Substructure Pattern Mining (gSpan) [3] uses canonical labeling and the right-most extension technique. GrAph/Sequence/Tree

extraction (Gaston) [22] 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) [23]. The success of this method lies in the development of the novel concepts of *equivalent occurrence* and *early termination*, which help CloseGraph prune the search space substantially with small additional cost.

In our thesis, the frequent subgraphs discovered by the modified FP-growth approach were used to cluster the documents. Next section will give a brief overview of some of most commonly used text mining techniques.

<u>Text Mining</u>

There have been a number of efforts to make organizing text documents easier and more efficient [24] [25]. To improve the efficiency of document clustering, they have been transformed to vectors that describe the content of that document [26] [27]. Most terms in a document are function words; they are usually removed by almost all clustering methods at the very beginning. *Stemming* [28] is another useful operation that collapses different word forms to a common root. Thus, the verb *go*, *went* and *going* will all be mapped to the same root *go*.

A word appearing multiple times in a document may or may not be relevant to document clustering. Term frequency, $tf_{i,j}$, indicates the relevancy of the word. It indicates the frequency of a term $fw_i$ appearing in a document $d_j$. To indicate how informative a word is, the vector space model uses inverse document frequency (*idf*),

defined as: $idf = \log \dfrac{|D|}{|\{t \in d\}|}$, where $|D|$ is the total number of documents in the

document set and $|\{t \in d\}|$ is the number of documents containing the term $t$. In the

classic vector space model [26], the term-specific weights in the document vectors are

products of $tf_{i,j}$ and $idf$. The weight vector for document $d$ is

$v_d = [wt_{1,d}, wt_{2,d}, ..., wt_{N,d}]^T$ where $wt_{i,j} = tf_{i,j} \cdot \log \dfrac{|D|}{|\{i \in j\}|}$. The similarity in vector

space models is determined by using various measures between the document vectors

where overlapping word indicates similarity. The most popular similarity measure is the

cosine coefficient, which measures the angle between two document vectors [29].

Aside from the vector space model, thematic description treats the text as a set of

tokens, which can be either equal or completely different. Conceptual graph

representation, however, permits us to describe the text as a set of concepts and relations

between them [30] [31]. They can partially differ and partially coincide. Extensive

studies have been conducted with document graphs in the field of document

summarization [32] [33]. A document graph contains two kinds of nodes, concept/entity

nodes and relation nodes.

Although we also name our graph representation of the documents as *document-graph*, our *document-graph*s have different structure than the document graphs described

in [32] [33]. We use only one type of node that stores the synsets, and we use the noun

keywords with their hypernymy relationship to construct our *document-graph*s.

CHAPTER 3

METHODOLOGY AND IMPLEMENTATION

In this chapter, we explain the implementation details of this thesis. The overall mechanism we used in this thesis is briefly portrayed in Figure 2.

Pre-processing

| | |
|---|---|
| **Dataset: 20NG** | **Document Clustering** |

**Bow** → **Keywords *W* and their frequency per document**

**Hierarchical Agglomerative Clustering (HAC)**

**WordNet** → **Create and label Master Document Graph (*MDG*), and Document-Graphs (*DG*) for each document**

FP-growth

**Labeled DGs** ↔ **Create the FP-tree** ↔ **Frequent Subgraph Discovery** ↔ **Frequent Subgraphs**
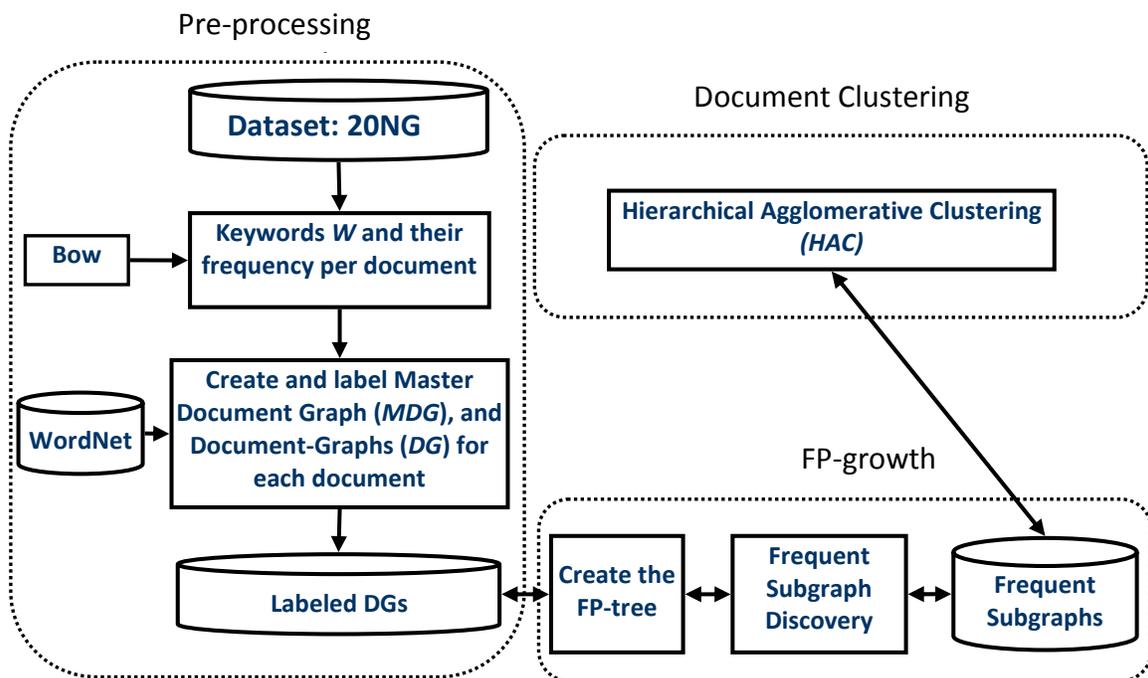
Figure 2: Overall mechanism of graph-based document clustering using FP-growth.

In the following sections we describe the 20NG dataset [12], the preprocessing required for this work, the details of the FP-growth algorithm, and our modified version of FP-algorithm for frequent subgraph discovery. Additionally, we analyze FP-growth approach and point out its problems for subgraph discovery. We also describe the clustering mechanism and its evaluation used for documents' clustering.

## Dataset

20 NewsGroup (20NG) [12] is a dataset that consists of 20,000 text documents taken from 20 Usenet newsgroups. Around 4% of the articles are multi-labeled. It means that one article may belong to multiple groups due to its content. The articles are normal postings on various topics: they have headers including subject lines, signature files and quoted portions of other articles. A number of subsets exist for the 20NG dataset [34]. In this thesis, we use customized subsets of 20NG. For our experiments, we have used two subsets of 20NG: one contains 500 documents from 5 groups and the other contains 2500 documents from 15 groups. We use Bow [13] to extract keywords from the documents' corpora. Bow contains a library of C code that has been mostly used for writing statistical text analysis, language modeling and information retrieval. Based on the set of retrieved keywords ($W$), two files are generated: one contains the sparse matrix for each document, the other file contains the set of selected keywords $(w)$ appearing in the documents and their corresponding frequencies. Table 1 gives a brief description of the files generated by Bow.

Table 1: Description of 2 files generated by Bow for 20NG dataset.

| File | Content | | | | | | |
|------|---------|---|---|---|---|---|---|
| Sparse Matrix | **Filename** | **Group** | | *w1* | *w2* | *w3* | *w4……* |
| | 42222 | alt.atheism | | 1 | 1 | 1 | 1.. |
| | 42223 | alt.atheism | | 0 | 1 | 0 | 0.. |
| | 42224 | alt.atheism | | 1 | 1 | 0 | 0.. |
| Word-frequency | **Filename** | **Group** | **Word-frequency** | | | | |
| | 42222 | alt.atheism | article 2  writes 1  game 4 … | | | | |
| | 42223 | alt.atheism | writes 3   … | | | | |
| | 42224 | alt.atheism | article 1  writes 1  god 3 … | | | | |

## Pre-processing

In our thesis, we use the *word-frequency* file generated by Bow to create the *document-graph* for each document. The relationship between the keywords in a document is retrieved using the WordNet ontology. In WordNet, words are related to each other based on different semantic relationships among them. We have considered the hypernymy relation of noun keywords, which is a super-ordinate or IS-A ("a kind of") relationship. We believe that, for sense-based document clustering, this relation is more suitable than the others because it provides the generalization of a concept or a word. For example, the word *Orion* will create a hypernymy hierarchy as stated below:

*Orion @→…@→Universe @→ Natural object @→ Physical object @→ Entity*

where *@→* denotes a semantic relation that can be read '*is a.*' If we can utilize this relationship during the clustering of the documents, a document containing the word *Orion* will have a high chance of being in the cluster that has documents related to *Universe*, even though the other documents in the cluster may not contain the exact word *Orion*.

In the most updated version of WordNet, *Entity* is the highest level of abstraction and all the words connect to it. Thus, for each keyword of a document, all of their hypernymy hierarchies will be connected up to *Entity*. This results in a connected graph called a *document-graph* (*DG*). Once we have all the DGs, we create a Master Document Graph (MDG) [35] by selecting all the keywords and creating one graph based on the keywords' IS-A relationship in the WordNet. We use the MDG for a number of reasons. Instead of taking the entire WordNet as the background knowledge we use MDG that

contains the ontology related to the keywords only. This allows us to concentrate on the area of the WordNet which is relevant to our dataset. Second reason why we use the MDG is that it can facilitate the use of the DFS codes. The DFS code helps us to mark each edge uniquely, so even if the same edge appears in various DGs, all of them will have the same DFS code. Finally, MDG can help our modified FP-growth approach for checking the connectivity constraint.

FP-growth was designed for mining frequent itemsets in market basket dataset [9]. We transformed our graph representation of the documents to a table consisting of list of edges appearing in each document. This helps us to fit the frequent subgraph discovery task to frequent itemset discovery problem. Each edge of the DG is considered to be an item of a transaction, and each document is considered a transaction. We used DFS coding inspired by gSpan [3] to identify the edges. The code is generated by traversing the MDG in Depth First Search order. This gives us the DFS traversal order (DFS code) for the entire MDG. This unique DFS code is used to mark every edge of all the DGs. DFS codes are mostly used as canonical labels. They also provide a useful way to point out the edges that create cycle in the MDG. Cycles are very common in WordNet as one word may have different meanings all of which can share one common higher abstraction level. The edge that creates a cycle is known as the *backward edge*. All the other edges are called *forward edge*s.

The mechanism to create the MDG from the DGs and generating DFS codes are described from Figure 3 through Figure 5. Figure 3 shows the DGs of three documents. The keywords (marked as black nodes) are retrieved by Bow [13]. We generate the DGs

by traversing the WordNet ontology up to the *Entity* level from each keyword. For a keyword, the more we traverse to the upper levels of the hierarchy the concept of that keyword becomes more abstract. This behavior plays an important role in our sense-based clustering approach. Very high abstractions do not provide relevant information for clustering. All the keywords can be linked back to *Entity* causing the high level abstraction to appear in all the DGs. A concept that appears so frequently does not provide enough information to separate the clusters properly. This can influence the clustering towards a single cluster as well.

Figure 4 (a) shows the MDG for the three DGs presented in Figure 3. The MDG is created by taking all the keywords and creating a single graph based on their IS-A relationship in the WordNet. Figure 4 (b) shows the construction of the DFS codes of the MDG. The dashed lines show the backward edges of the graph. Each of the DFS codes points to a unique edge of the MDG. Once we have the DFS codes of the edges of the MDG, we use this information to mark the edges of the DGs. Figure 5 shows each DG with the imposed DFS codes from the MDG. Finally, we get a set of documents having all the edges marked using consistent DFS codes. We use these DFS codes for the edges of each document as items appearing in a transaction. This makes our dataset compatible for the original FP-growth that was designed for a transaction database where each of the transactions contained a list of items purchased by the consumer.
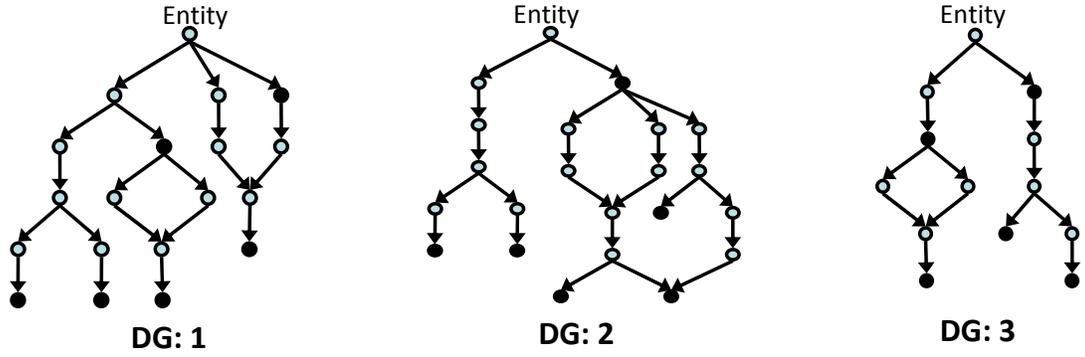
Figure 3: DGs for 3 documents.



(a) MDG

(b) Construction of DFS codes for MDG
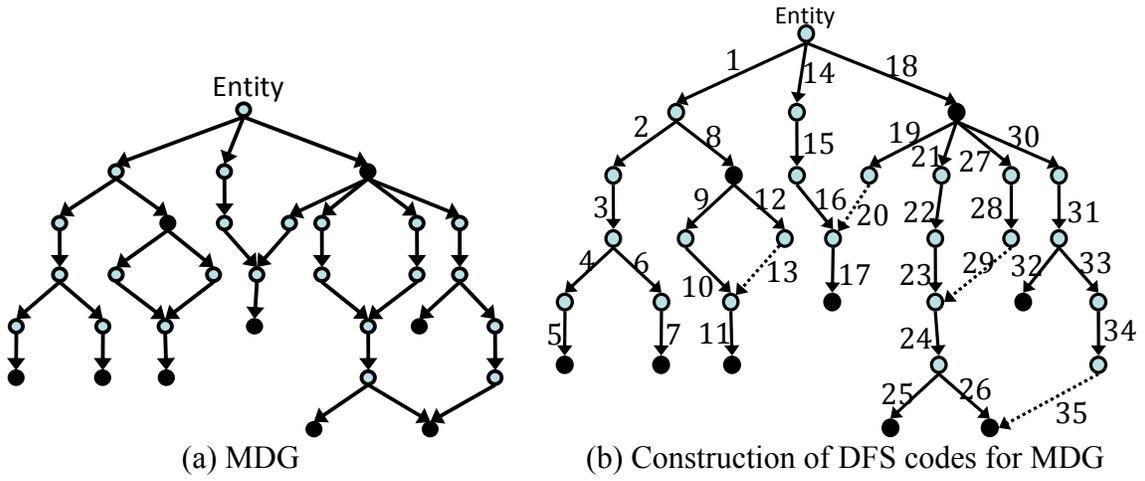
Figure 4: MDG for the repository of the 3 DGs of Figure 3.
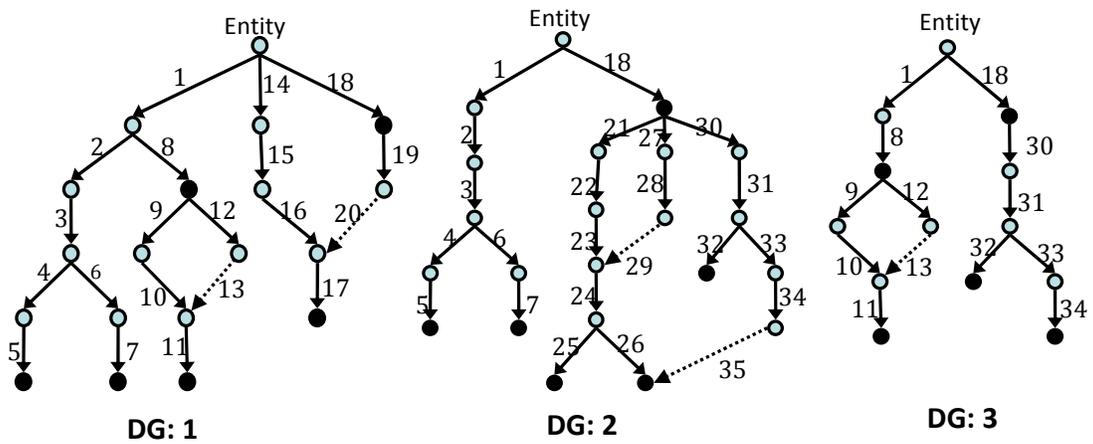


Figure 5: DGs with DFS coding.

## The FP-growth Approach

The FP-growth algorithm consists of two major steps. First we must create an FP-tree which is a condensed representation of the dataset. Then we need to create a mining algorithm that would generate all possible frequent patterns from the FP-tree by recursively traversing the conditional trees generated from the FP-tree.

### Creating the FP-tree

The algorithm that creates the *Frequent Pattern* (FP) *Tree* is shown in Table 2. This algorithm first scans the DG database (DB) to find the edges that appear in the DG database. These edges and their corresponding supports are added to a list called *F*. Then we create a hash table 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 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 *FList*. Thus, $FList = \{(e_i, \beta_i) \mid e_i \in MDG, \beta_i \geq \min\_sup \ and \ \beta_i \geq \beta_{i+1}\}$. Based on the sorted order of *FList*, *transactionDB* is also altered so that all the infrequent edges are removed from each entry and the most frequent edges appear at the beginning.

Once we have the sorted *FList* and *transactionDB*, we create the FP-tree. Initially the root of the FP-tree is an empty node referred to as *null*. We then call the *insert_tree([p|*

Table 2: Algorithm for FP-tree Construction [8]

| | |
|---|---|
| **Input** | DG database DB and |
| | Minimum support threshold, *min_sup*. |
| **Output** | Frequent Pattern Tree (T) made from each $DG_i$ in the DB. |

*1.* Scan the DB once.

*2.* Collect *F*, the set of edges, and corresponding support of every edge.

*3.* Sort *F* in descending order and create *FList*, the list of frequent edges.

*4.* Create *transactionDB*

*5.* Create the root of an FP-tree *T*, and label it as "*null*".

*6.* For each $DG_i$ in the *transactionDB* do the following:

*7.* Select and sort the frequent edges in $DG_i$ according to *FList*.

*8.* Let the sorted frequent-edge list in $DG_i$ be *[p|$P_i$]*, where *p* is the first element and $P_i$ is the remaining list.

*9.* Call **insert_tree([p|$P_i$], T)** which performs as follows:

*10.* If *T* has a child *N* such that *N.edge_dfs = p.edge_dfs*,
then N.count++;
Else
create a new node *N*, with N.count=1
Link N to its parent *T* and link it with the same edge_dfs via the node-link structure.
If $P_i$ is nonempty, call **insert_tree([p|$P_i$], N)** recursively.

$P_i$], T) method to insert the sorted edges of $DG_i$ in the FP-tree, denoted as *T*, 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|$P_i$], T)* is called, it checks whether *T* 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 $T$ or until there is a root that does not contain $p$ as its outgoing node indicating that $T$ 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 we also need to maintain a list of node links to point out where they appear in the FP-tree. *FList* 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.

Figure 6 describes the sorting procedure for the document edge list ($P_i$). If we have three DGs named $X$, $Y$ and $Z$, we collect the DFS code (DFS id) of every edge

| DG | DFS Id list ($P$) |
|----|-------------------|
| $X$ | 1, 3, 4 |
| $Y$ | 5, 2, 3 |
| $Z$ | 2, 1, 6 |

(a)

*min_sup* = 1

| DFS Id | Frequency |
|--------|-----------|
| 1 | 2 |
| 2 | 2 |
| 3 | 2 |
| 4 | 1 |
| 5 | 1 |
| 6 | 1 |

(b) *FList*

| DG | DFS Id list ($P$) |
|----|-------------------|
| $X$ | 1, 3, 4 |
| $Y$ | 2, 3, 5 |
| $Z$ | 1, 2, 6 |

(c)
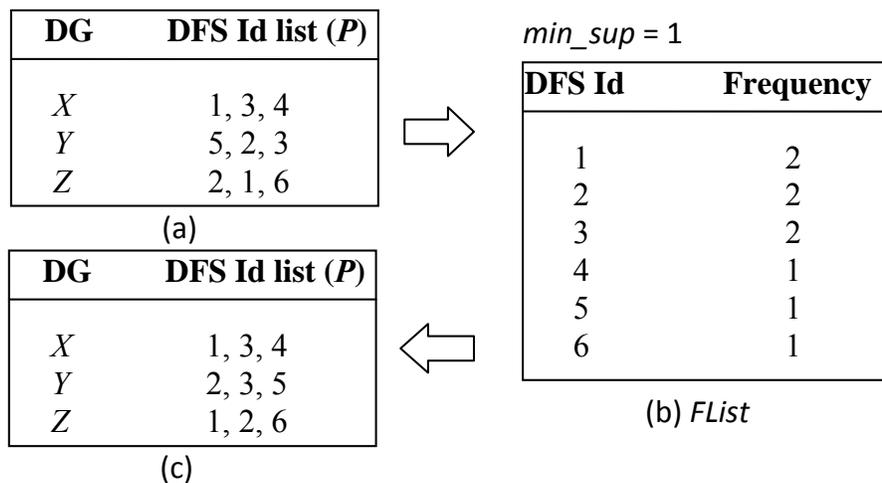
Figure 6: Sorting Document Edge list by Support

Document edge list

| DG | DFS Id list (*P*) |
|---|---|
| *X* | 1, 3, 4 |
| *Y* | 2, 3, 5 |
| *Z* | 1, 2, 6 |

Header table *(FList)*

| DFS Id | Frequency |
|---|---|
| 1 | 2 |
| 2 | 2 |
| 3 | 2 |
| 4 | 1 |
| 5 | 1 |
| 6 | 1 |
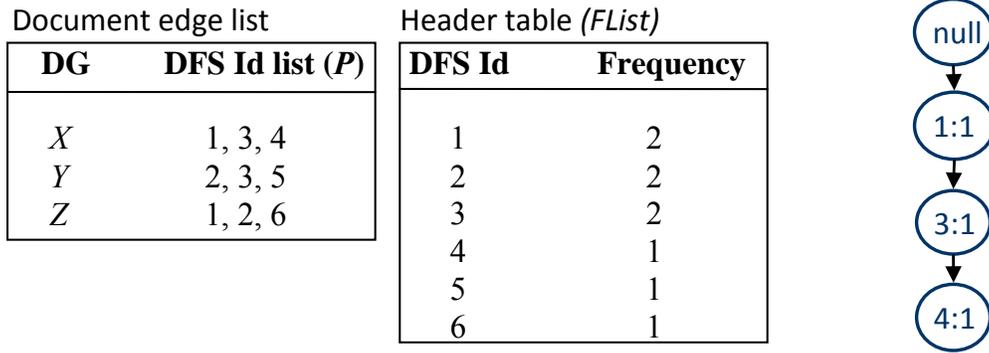
Figure 7: Creating the FP-tree (for $DG_i = X$).

Header table *(FList)*

| DFS Id | Frequency | Node-link |
|---|---|---|
| 1 | 2 | |
| 2 | 2 | |
| 3 | 2 | |
| 4 | 1 | |
| 5 | 1 | |
| 6 | 1 | |
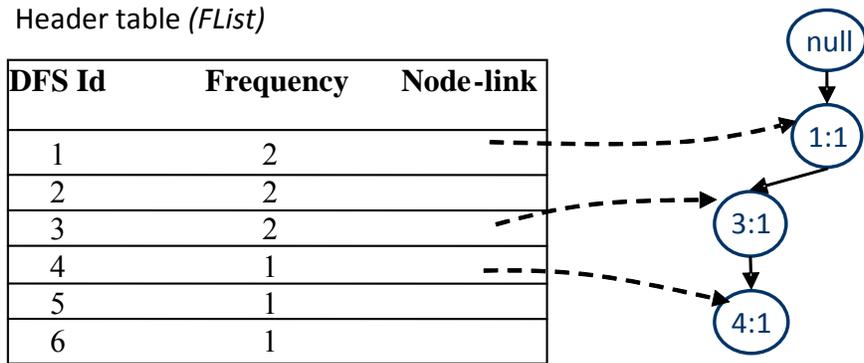
Figure 8: Node-link field in the header table

Header table (*FList*)

| DFS Id | Frequency | Node-link |
|---|---|---|
| 1 | 2 | |
| 2 | 2 | |
| 3 | 2 | |
| 4 | 1 | |
| 5 | 1 | |
| 6 | 1 | |
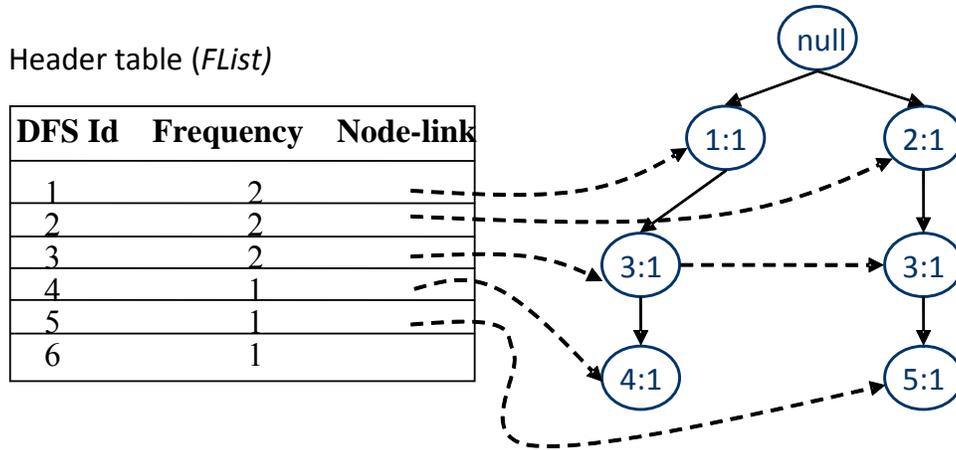
Figure 9: Adding edges in the FP-tree (for $DG_i = Y$)

that belongs to these DGs. Figure 6 (a) shows each DG with its respective DFS edge ids. We then create a list *F* based on the DFS edge ids and store the corresponding frequencies (support). Consider that in this case the minimum support provided by the user is *1*; thus every edge in *F* will be frequent. If the user provides a minimum support of *2*, edges with DFS edge id *4*, *5* and *6* should be removed from *F* since their support is lower than the provided minimum support of *2*. We get *FList* (Figure 6 (b)) after pruning *F* according to the minimum support (i.e. *1*) and sorting the DFS edge ids in descending order of supports. The *FList* is usually referred to as the *header table* in original FP approach and is used to sort the edge list of all the DGs (Figure 6 (c)). The header table (*FList*) and the sorted document edge lists ($P_i$) are used to create the FP-tree.

We start with DG database (DB) and select each $DG_i$ to add its edges in the FP-tree. For example, DG *X* is selected first during the FP-tree creation process. We now select edge list of DG *X* (i.e. *{1, 3, 4}*) and create the FP-tree based on them. To create the FP-tree we first create a *null* node and link it to the first edge of the edge list $P_x$ (i.e. the edge with DFS id *1*). Then we add the second edge from the edge list to the tree and link it to the first edge. Every node in the FP-tree contains two fields, the DFS edge id and the support of that edge encountered so far. These two fields are separated by a semicolon from Figure 7 through Figure 12. The FP-tree created for DG *X* is shown in Figure 7. Creating an FP-tree also requires us to store the pointer to the edges in the tree. For this, we add another field to the header table (*FList*) called *node-link*, which contains the pointer to the edge in the FP-tree. The pointer to the FP-tree in the header table is shown in Figure 8.

If an edge is added to the FP-tree for the first time, the *node-link* pointer only holds the pointer of this edge. If an edge is already present in the FP-tree and our current set of edges *(Pᵢ)* requires us to place it as a separate node in the FP-tree, we link the new node to the already existing node. Once we have created the branches related to DG *X*, we select the next DG$_i$ (i.e. DG *Y*) from the *transactionDB*. We follow the same procedure to add the edges of *Y* (i.e. *{2, 3, 5}*) in the FP-tree shown in Figure 9. Since the tree already contains edge *3* but it was placed as the edge co-occurring with edge *1*, we create a link between the previously inserted node *3* (co-occurring with edge *1* in *X*) and the newly added node for *3* (co-occurring with edge *2* in *Y*).

Next, we select the edge list of *Z* which is *{1, 2, 6}* (see Figure 7). We start from the root node *(null)* of the FP-tree (see Figure 8) to see if it contains *1* as its child. Since there is already a node *1* as child, we increase its support by one and change our pointer to root to this child (i.e. node with DFS edge id *1*). We then follow the same procedure to check if the root (i.e. node *1*) has any children with label *2* (second edge in the current edge list of *Z* (*{1, 2, 6}*). Since *2* does not exist as *1*'s child in the FP-tree, we create a new node for DFS edge id *2*, and create a link between the node with *1* (currently the *root* node) and the newly created node for edge *2*. The pointer to the *root* is also changed to point to the newly created node *2*. Although *2* is a new child for *1*, the FP-tree already contains a node for edge *2*. So the header table contains a pointer to the first node with DFS edge id *2*. In this case, we only need to link the earlier node designated for the DFS edge id *2*, to the newly created node with the same DFS edge id. Both the nodes for edge *2* represent one edge in the MDG. They are stored separately in the FP-tree to denote the

Header table *(FList)*

| DFS Id | Frequency | Node-link |
|--------|-----------|-----------|
| 1 | 2 | |
| 2 | 2 | |
| 3 | 2 | |
| 4 | 1 | |
| 5 | 1 | |
| 6 | 1 | |
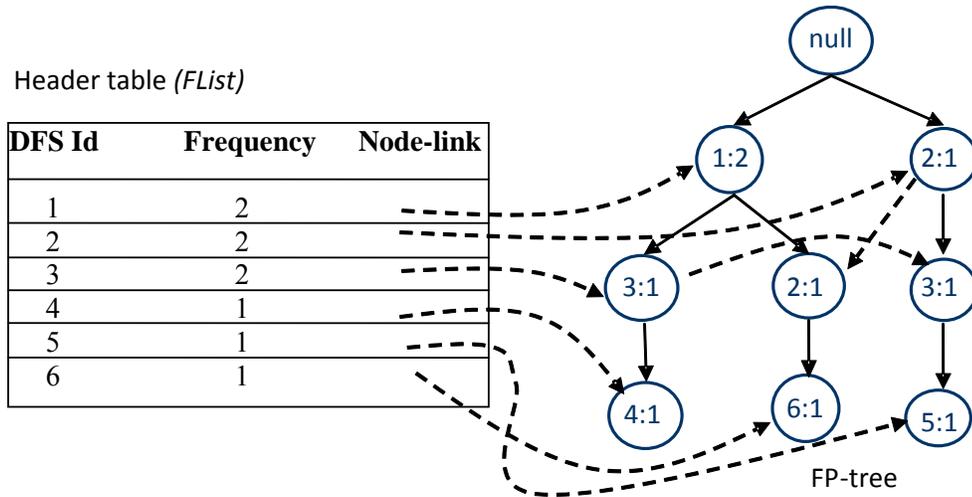
Figure 10: Adding edges in the FP-tree (for $DG_i = Z$).

characteristics of individual DGs. In the document-graph *Y*, edge *2* co-occurred with edge *3* and *5* while in *Z* it co-occurred with edge *1* and *6*. Therefore, DFS edge id *2* is placed in two different branches of the FP-tree (Figure 10).

Edge *6* is new in the currently investigated sub tree of the FP-tree, so we add this new node as the child of *2* and create a new pointer to the respective row in the header table. This process continues until all the DGs in the database (DB) are mapped into the FP-tree. The final FP-tree with its header table is shown in Figure 10. We have described an example with larger document-graphs in the Appendix.

Mining the FP-tree [8]

After we create the FP-tree, we need an efficient algorithm to discover the entire set of frequent patterns. The algorithm that we will explain takes the FP-tree and the minimum support *min_sup* as inputs and discovers all possible patterns of the FP-tree that are frequent. The idea behind the FP-tree mining algorithm is to transform the problem of finding long frequent patterns into looking for shorter ones recursively. This is done

using divide-and-conquer strategy. We start with all the frequent edges (i.e. 1-edge subgraphs) in our data repository (DB). Then we create a conditional pattern base for each of them.

A conditional pattern base for an edge is the prefix path of the FP-tree that has the corresponding edge as its suffix. For example, in Figure 10 the prefix path for edge *6* contains nodes *1* and *2*. Thus *1* and *2* creates the conditional pattern base for edge *6*. This conditional pattern base (i.e. *1, 2*) is used to create the conditional FP-tree for the edge *6*. Instead of using the original FP-tree, we need to generate a conditional FP-tree for each edge since some edges may be frequent for the entire FP-tree but may not appear frequently with other frequent edges. This conditional FP-tree is divided further using the same principle.

The FP mining algorithm (Table 3) starts with the header table *(FList)* and selects the edges starting from the least frequent ones. A conditional pattern base is created for the edge $a_i$ by traversing the edge $a_i$-specific pointers to the FP-tree and selecting the branches that end with $a_i$. After generating the conditional pattern base, a conditional FP-tree is created based on the conditional pattern base. We check the frequency of the edges appearing in the conditional pattern base for $a_i$ and select only those that support the *min_sup* for adding them in the conditional FP-tree. The conditional FP-tree is then recursively mined to discover all possible frequent patterns. Table 3 describes the algorithm used for generating frequent patterns from the FP-tree.

As long as there are branches in the conditional FP-tree (multi-path), we increase the size of $a_i$ by adding edges from the conditional header table (step 5 of Table 3). We

Table 3: *FP_growth* algorithm for mining the FP-tree [8]

| | |
|---|---|
| **Input** | The FP-tree *T* |
| | Frequent pattern $\alpha$ (at the beginning, $\alpha$ =*null*) |
| | Header table *hTable*, with edges denoted as $a_i$ |
| **Output** | Frequent patterns $\beta$ |
| *1.* | if *T* contains a single path *P* then |
| *2.* | for each combination (denoted as $\beta$) of the nodes in path *P* |
| *3.* | generate pattern ($\beta \cup \alpha$) with *support* =MIN*( supports of all the nodes in $\beta$)* |
| *4.* | else for each $a_i$ in the *hTtable* of *T* |
| *5.* | generate pattern $\beta = a_i \cup \alpha$ with *support* = $a_i$.*support*; |
| *6.* | construct $\beta$'s conditional pattern base and use it to build $\beta$'s conditional FP-tree *Tree$_\beta$,* construct $\beta$'s conditional header table *hTable$_\beta$* |
| *7.* | if *Tree$_\beta$* $\neq \varnothing$ then **FP_growth**(*Tree$_\beta$, $\beta$, hTable$_\beta$*); |

then create the new $a_i$'s conditional pattern base along with its conditional FP-tree (step 6). This process continues until there is a single path in the tree (step 1). The algorithm then generates all possible combinations of this single path and adds these combinations with $a_i$ to generate the set of frequent patterns for $a_i$. The frequent patterns of all the frequent edges that appear in the header table *(FList)* form the entire set of frequent patterns appearing in the database.

We will now describe the FP-growth algorithm with an example. Figure 11 shows the process for mining frequent patterns related to edge *6*. We first create the conditional pattern base for *6* by traversing the FP-tree. This conditional pattern base consists of the edges *1* and *2*. Both of their supports are kept as *1*, as the support of 6 is *1*.

This conditional pattern base is used to retrieve the conditional FP-tree for edge *6* (Figure 11) from the original FP-tree. Since the conditional tree does not have multiple branches in it, it will be considered as a single path consisting of the edges *1* and *2*. Following the algorithm described in Table 3, we generate the combinations of this single path with DFS edge ids *1* and *2*. The combinations are: *{1}*, *{2}*, *{1, 2}*. These combinations are added with the suffix *6*, thus resulting in the frequent patterns *{1, 6}*, *{2, 6}* and *{1, 2, 6}*.
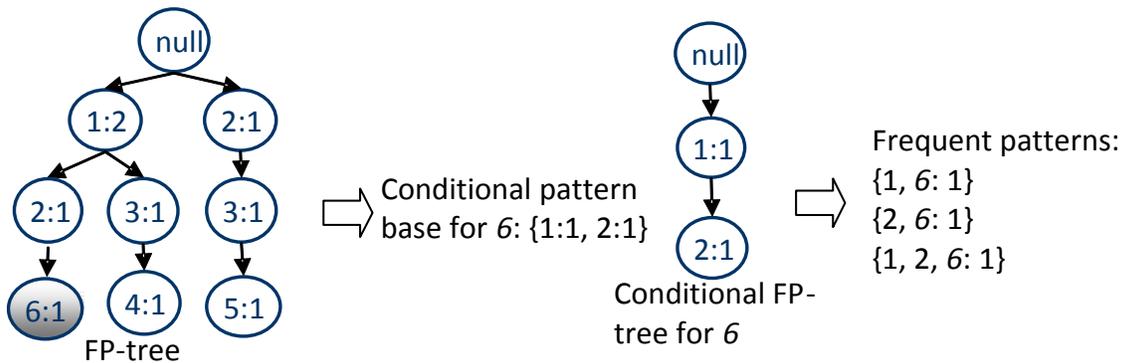


Figure 11: Mining the FP-tree (single path).
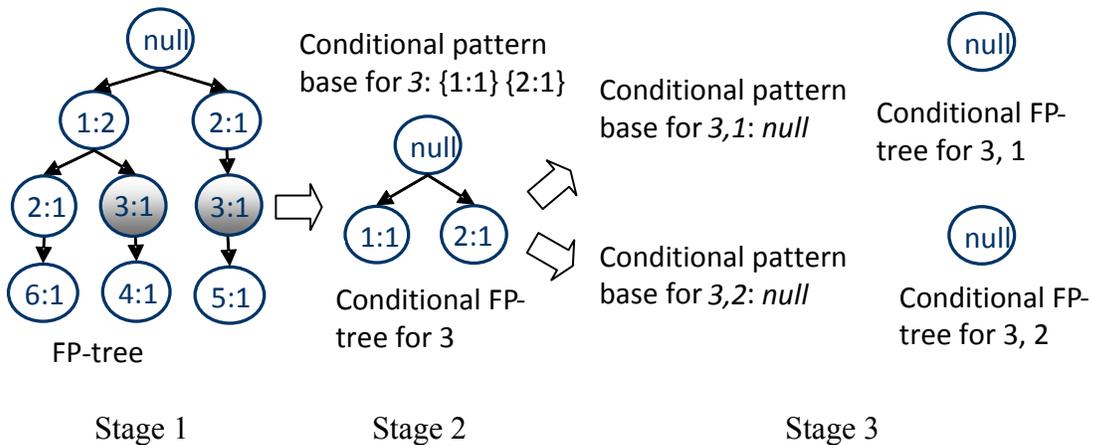


Figure 12: Mining the FP-tree (multi-path)

Figure 12 shows the decomposition process of the conditional FP-tree if there is a multi-path in the tree. In such case, the suffix that generated the conditional FP-tree (e.g. node *3*) is combined with the elements of the conditional header table to create the new conditional pattern bases (stage 3 of Figure 12). Later, the conditional FP-tree is created with the elements of conditional pattern base (stage 3 of Figure 12). The process terminates when the conditional FP-tree becomes empty (contains *null* node.) The frequent patterns discovered whose appearance is conditional to *3* are {*3, 1*} and {*3, 2*}.

FP-growth is efficient for a number of reasons. First of all, it compresses the dataset to a tree format which does not contain the infrequent edges. The size of the FP-tree can be smaller than the size of the original dataset if lots of infrequent edges have been removed. The algorithm reads each of the DGs from the database, removes infrequent edges and then maps the remaining edges to the path of the FP-tree. DGs that share a common substructure in their graphs will create an overlap in this tree in the path that represents this common segment. The more the same substructure is shared, the better the compression of DGs into the FP-tree. This compression makes the future search for frequent patterns easier. Although, most of the time, the size of the FP-tree is smaller than that of the original dataset; sometimes the tree can be as large as the original dataset if the documents do not share concepts. The physical storage requirement of such FP-tree can be higher, since its nodes need to store pointers along with their supports.

A second benefit of the FP-growth is that it discovers the complete set of frequent patterns without the expensive candidate generation process. Instead of looking for large patterns, it finds the smaller ones that are frequent and concatenates them with their

suffix pattern. This divide-and-conquer strategy begins by selecting the least frequent edges over more frequent ones. The order of the edges plays a vital role in the size of the FP-tree. Since it begins the pattern mining by first selecting the least frequent edges, it offers good selectivity in most cases [36]. Creating an FP-tree starting with the least frequent edges, usually results in less branching at the top of the tree. This practice also reduces the number of nodes in the FP-tree [36].

## Feasibility of FP-growth in Text Clustering

Graphs have been used in this thesis to represent the documents. We create a DG of an individual document using the IS-A relationship between its noun keywords in the WordNet. Our intention is to find frequent subgraphs within these DGs. We believe that these frequent subgraphs will better reflect the sense of the documents, provided that the documents are focused on a specific subject which is reflected by their keywords. The existing FP-growth approach was designed to find the frequent items, and it requires extensive tree traversing to discover those patterns (i.e. itemsets). The original FP-growth algorithm, when applied to our problem (which we made possible by mapping edges to items, and DGs to transactions), generates a set of frequent edges which do not necessarily constitute to a connected subgraph. The DGs that we have contain a list of connected edges. Generation of all possible frequent patterns not only outputs all possible frequent subgraphs but also generates a lot of overhead in the form of all possible sets of frequent, but disconnected, edges. This causes unnecessary costs during the mining of the DGs as we are only looking for frequent subgraphs and these sets of frequent

disconnected edges bear no useful information for our document clustering. The time and space required to generate and store these disconnected frequent edges have negative impact on the overall performance of the FP-growth approach.

This leads us to the second problem, which is related to the computational cost and the memory requirement of the FP-growth approach. The FP-growth compresses the original dataset into a single FP-tree structure. Creating this tree is not computationally expensive, but if the database consists of thousands of DGs then the tree can become large. Our main concern was the second step of the FP-growth. Usually the DGs contain hundreds of edges. Generating all possible frequent patterns for these edges often creates a huge number of frequent patterns, and the memory runs out. FP-tree requires extensive mining to discover the frequent patterns. This mining became extensive based on the minimum support. If the minimum support of a subgraph is very low it can appear rarely in the DGs. Also, if the number of documents is large, the cost of generating these subgraphs becomes enormous. Moreover, if during the FP-tree mining process we find a single branch in the FP-tree, we generate all possible combination of that branch. This often caused our program to run out of memory even with high minimum support. The compression factor of the dataset plays a vital rule on the runtime performance of FP-growth. The performance of this algorithm degrades drastically if the resulting FP-tree is very bushy. In this case, it has to generate a large number of sub groups and then merge the results returned by each of them [34]. Originally, FP-growth was tested on datasets which have small numbers of items. On the contrary, in our case every document has a large amount of edges which constitute the DG. To achieve high quality clusters in

frequent pattern-based clustering, the support is usually kept very low (i.e., 3-5%). With the normal FP-growth approach, we often ran out of memory (we had a machine with 1 GB of RAM) even when the minimum support was as high as 70% to 80%.

## FP-growth Approach for Frequent Subgraph Discovery

To make the existing FP-growth algorithm suitable for graph mining, we changed the algorithm so that it discovers frequent connected subgraphs and performs better for even bushy DGs. The outline of our process of frequent subgraph mining is described briefly in Table 4. We start by creating a hash table called *transactionDB* for all the *DG*s which is similar to original FP-growth procedure described in Table 2. Next, we create the *headerTable* (same as the *FList*) from all the edges appearing in the MDG. After creating *transactionDB* and *headerTable*, we call the method *FilterDB* 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.

Table 4: Modified FP-growth algorithm

| | |
|---|---|
| **Input** | Document graphs' database *DB* |
| | Master Document graph *MDG* |
| | Minimum support *min_sup* |
| **Output** | Frequent subgraphs *subGraph_j* |
| *1.* | Create a list called *transactionDB* for all $DG_i \in DB$ |
| *2.* | Create *headerTable* for all edge $a_i \in MDG$ |
| *3.* | *FilterDB(transactionDB, headerTable, min_sup)* |
| *4.* | *FPTreeConstrcutor()* |
| *5.* | **FPMining()** *// see* Table 5 *for details* |
| *6.* | For each subgraph *subGraph_j* |
| *7.* | *includeSubgraphSupporteDocs(subGraph_j)* |

*transactionDB* and *headerTable*) by removing the 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. This mechanism is explained in Chapter 4. *transactionDB* is updated to reflect this change in the header table. After this refinement, we create the FP-tree by calling *FPTreeConstructor()* method. Later, the method *FPMining()* generates the frequent subgraphs by traversing the FP-tree. Most of the modifications we did were on the FP-Mining part which traverses the FP-tree to discover the frequent patterns. For each frequent *subGraph*, a list of document ids is created in which the *subGraph* appears.

We will now explain the modified mining algorithm for FP-tree, which is also

Table 5: Algorithm for Modified FP-Mining: *FPMining()*

| Input | FP-tree $T$ |
|---|---|
| | FList *headerTable* |
| | Frequent pattern $\alpha$ (initially, $\alpha$ =*null*) |
| **Output** | Frequent subgraphs $\beta$ |
| *1.* | If $T$ contains a single path |
| *2.* |     If ($T.length > SPThreshold$) |
| *3.* |         Delete ( $T.length - SPThreshold$ ) number of edges from the top of $T$. |
| *4.* |     for each combination (denoted as $\beta$) of the nodes in path $T$ |
| *5.* |         If (**isConnected**($\beta, \alpha$) == true) // see Table 6 for details |
| *6.* |             generate $\beta \cup \alpha$, support = MIN *(support of nodes in $\beta$)* |
| *7.* | else for each $a_i$ in the *headerTable* of $T$ |
| *8.* |     generate pattern $\beta = a_i \cup \alpha$ with *support* = $a_i.support$; |
| *9.* |     If (**isConnected**($\beta, a_i$) == true) |
| *10.* |         construct $\beta$'s conditional pattern base and use it to build $\beta$'s conditional FP-tree *Tree$_\beta$* and $\beta$'s conditional header table *headerTable$_\beta$* |
| *11.* |         if *Tree $_\beta$* $\neq \varnothing$ then |
| *12.* |             call **FP-Mining**(*Tree$_\beta$, headerTable$_\beta$, $\beta$*); |

described in Table 5. We start by calling the FP-Mining algorithm with the *FP-Tree,*

*headerTable* and initially a *null* value which is the suffix for a possible frequent pattern.

If the FP-Tree does not contain any single path, the algorithm selects the least frequent

edge $a_i$ from the header table and combines this edge with the *null* value. The result of

this combination is $\beta$ (i.e. after the first iteration, $\beta = (a_i)$). A new conditional pattern base

is created for $\beta$. The size of $\beta$ may keep growing by adding edges from the conditional

header table provided that the support of the growing pattern (possible subgraph) is

higher than the *min_sup* threshold. This means that the pattern is frequent. Once we have

the frequent pattern $\beta$, we create a conditional FP-tree *Tree*$_\beta$ and conditional header table

*headerTable*$_\beta$ based on it. If *Tree*$_\beta$ is not empty the FP-Mining algorithm is called again

with *Tree*$_\beta$, *headerTable*$_\beta$ and $\beta$ as parameters. If at any point, a single-path is

encountered in the FP-tree, we prune edges from the top of the single path based on the

user provided threshold *SPThreshold*. After that, each combination of the existing single

path is generated and checked to see if the edges are connected in the MDG using the

method *isConnected()*. We will explain the details of this process later. All the

combinations of the edges that are connected create frequent subgraphs and their

discovery is conditioned on $\beta$. The support of the combined subgraph is determined by

the support of $\beta$ before the merging.

One of the other concerns of this work was to generate all possible combinations

of a single path appearing in any conditional FP-tree. The depth of the MDG can reach up

to 18 levels, this is the maximum height of the IS-A hierarchy of WordNet. Since our

DGs contain hundreds of edges, the depth of the FP-tree can reach up to hundreds

Table 6: Algorithm for checking connectivity: ***isConnected (β, α)***

| | |
|---|---|
| **Input** | Combination of edges, *β* |
| | Frequent pattern, *α* |
| **Output** | Returns *true* if *β* and *α* composes a connected subgraph, otherwise returns *false*. |
| **Global variable** | *connectedList* |
| **Method** | ***isConnected(β , α)*** |
| 1. | *connectedList=null;* |
| 2. | *edge* = the first edge of *β;* |
| 3. | ***Iterate(edge, β);***　　　　// is *β* connected |
| 4. | if (*connectedList.size* ≠ *β.size*) |
| 5. | 　　　return *false*; |
| 6. | for each edge $e_i$ in *β*　　　// is *α* and *β* connected |
| 7. | 　　　*edge* = the first edge in *β* |
| 8. | 　　　If (***isConnected***(edge, α) == true) |
| 9. | 　　　　　return *true*; |
| 10. | return *false*; |
| | |
| **Method** | ***Iterate(edge, subset)*** |
| 11. | *connectedList = connectedList* ∪ *edge* |
| 12. | *neighbors* = all incoming and outgoing edges of *edge* |
| 13. | for each edge $e_i$ in *neighbors* |
| 14. | 　　　If (*subset* contains $e_i$ && *connectedList* does not contain $e_i$ ) |
| 15. | 　　　　　***Iterate***($e_i$, *subset*) |

depending on the number of edges in a DG. While searching for a single path, if we reach a path which consists of 10 to 12 levels of depth, it becomes a large number to generate the combinations. We used an algorithm described by Rosen [37], which is one of the fastest, to generate all possible combinations.

One of the key modifications done in the original FP-growth is checking the connections during the subgraph discovery part. Conventional FP-growth generates all possible subsets of a single path for every edge-conditional FP-tree. Instead of accepting all possible combinations of the single path we only kept the combinations that were

Figure 13: Generation of combinations for a single path in the FP-tree

connected to generate a subgraph. This was done by first taking one of the edges from the combination of the single path and adding it to a list called *connectedList*. A neighboring edge list, *neighborList_i*, was created for this edge using the MDG. The construction of the *neighborList_i* was dependent on the source and target vertex of the edge *i*. We selected these vertices for each edge and retrieved all of their incoming and outgoing edges. The *neighborList_i* was checked to see if it contained any edge from the combination of the conditional FP-tree's single path. If there were such edges we followed the same procedure for all of them until no new edges from the combination of the single path were added to the *connectedList* (Table 6). At the end, the size of the *connectedList* was compared with the size of the combination (in step 4 of Table 6). If both of their sizes were the same, then the whole combination must be connected generating a subgraph. Then we try to combine this subgraph *β* with the frequent subgraph *α* in step 6 through 9 of Table 6.

For example assume that, for *DG₁*, there is a single path in the conditional FP-tree of an edge with DFS Id *7*, as shown in Figure 13. According to the original FP mining algorithm, we need to generate all possible combinations of this single path, and all of these combinations will count towards the set of frequent patterns. But for frequent subgraphs mining, we are only interested in the sets of connected edges (i.e. subgraphs).

A single path of length *k*, can result in a maximum *k*-edge subgraph. Four of the possible combinations of the single path of Figure 13 are shown in Figure 13 (a) through (d), two of which are connected (Figure 13 (a) and (c)). The rest of them are disconnected (Figure 13 (b) and (d)). Let us assume that, for the first combination (Figure 13 (a)), the
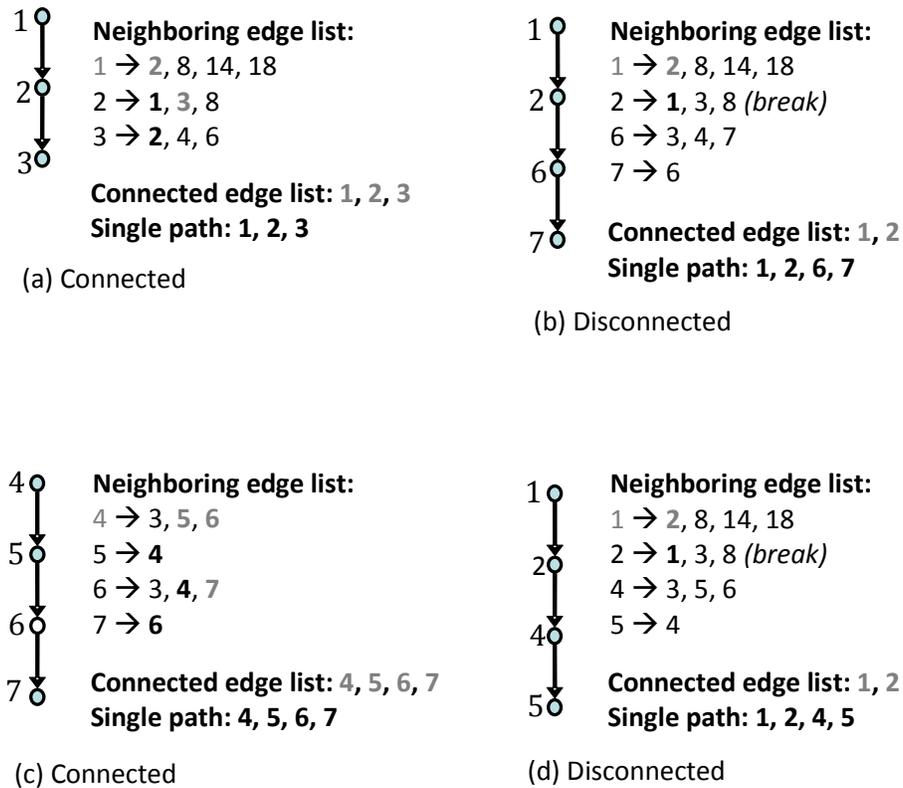
**Neighboring edge list:**
1 → **2**, 8, 14, 18
2 → **1**, **3**, 8
3 → **2**, 4, 6

**Connected edge list: 1, 2, 3**
**Single path: 1, 2, 3**

(a) Connected

**Neighboring edge list:**
1 → **2**, 8, 14, 18
2 → **1**, 3, 8 *(break)*
6 → 3, 4, 7
7 → 6

**Connected edge list: 1, 2**
**Single path: 1, 2, 6, 7**

(b) Disconnected

**Neighboring edge list:**
4 → 3, **5**, **6**
5 → **4**
6 → 3, **4**, **7**
7 → **6**

**Connected edge list: 4, 5, 6, 7**
**Single path: 4, 5, 6, 7**

(c) Connected

**Neighboring edge list:**
1 → **2**, 8, 14, 18
2 → **1**, 3, 8 *(break)*
4 → 3, 5, 6
5 → 4

**Connected edge list: 1, 2**
**Single path: 1, 2, 4, 5**

(d) Disconnected

Figure 14: Checking the connectedness of some possible combination

*isConnected* method of Table 6 takes *{1, 2, 3}* as its input parameter and finds whether this combination contains a connected subgraph or not. *isConnected* uses another method *Iterate* that adds the first edge (*1*) to the *connectedList* (marked in gray in Figure 14 (a)) and creates its neighboring edge list *neighborList*. The neighboring edges are selected from the MDG (Figure 4(a)).

To create the *neighborList* for an edge *e* we select its source vertex and target vertex from the MDG. All the incoming and outgoing edges of these two vertices are added to the *neighborList* of *e* except for the edge *e* itself. The edges from the *neighborList* are compared with the edges of a combination one by one. At any point, if one edge from the *neighborList* that is not listed in the *connectedList* is found in the combination, then that edge is selected for the next recursive call to **Iterate()** (see Table 6). This edge is then combined with the *connectedList* (marked in gray in Figure 14). The process continues until there is no new edge to add from the combination of the single path to the *connectedList*. Finally *isConnected* method checks the size of the final *connectedList* and compares it with the size of the combination. If both of them are the same then the combination contains a connected subgraph; otherwise the combination of the edges produces a disconnected subgraph. The process is shown in Figure 14 for four different combinations of Figure 13. For all the combinations, the edge ids that are inserted to the *connectedList* for the first time are marked in gray. Edge ids from the connected list that are already traversed during the iteration are marked in bold and black. The final *connectedList* also contains edge ids in gray.

It should be noted that *isConnected* method checks for the connectivity of one combination at a time. If the edges under consideration of this combination do not compose a connected subgraph, but composes multiple disconnected subgraphs, then some other combinations of the single path will generate these smaller connected subgraphs. So, we do not loose disjoint but smaller connected subgraphs of a larger disconnected combination of the single path. Additionally, we control the single path length by a threshold so that our FP-growth approach performs faster. We discuss about the single path threshold in Chapter 4.

## Clustering

Finding the frequent subgraphs leads us to the next step of our thesis – clustering the documents based on the occurrence of the frequent subgraphs. Clustering groups similar objects in such a way that the similarity between the groups (inter-cluster) is low and the similarity between the objects within each group (intra-cluster) is high.

One of the goals of this thesis is to cluster text documents based on their senses rather than keywords, which up until now, has been the most commonly used form of document clustering. We represented the text documents in a graph using *noun* keywords and the WordNet IS-A ontology of those keywords. Mining those document graphs resulted in the discovery of frequent subgraphs. These subgraphs can be viewed as a concept appearing frequently within the documents. If we evaluate similarity of the documents based on the co-occurrence of the same frequent subgraphs, and then use this similarity values to cluster the documents, we will get a sense-based clustering of the text

documents. Currently many techniques are available for clustering datasets [36]. Hierarchical Agglomerative Clustering (HAC) is one of the most popular among them.

Hierarchical Agglomerative Clustering (HAC)

Given $n$ elements, HAC creates a hierarchy of clusters such that at the bottom level of the hierarchy, every element is considered as a single independent cluster and at the top level all the elements are grouped in a single cluster. Unlike other clustering techniques, HAC does not require the number of clusters as its input. The desired number of clusters can be achieved by cutting the hierarchy at a desired level. The graphical representation of the hierarchy that represents the clusters in HAC is known as the *Dendrogram*. Dendrograms are very tall trees since with each iteration of HAC, only two clusters are merged. The example of Figure 15 shows a Dendrogram of hierarchical clustering of 5 points.

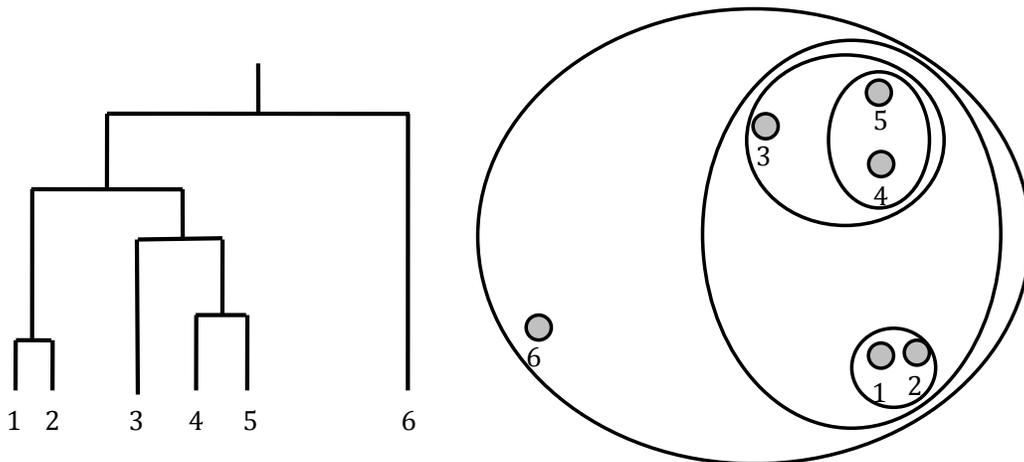There are two major approaches towards HAC, agglomerative and divisive.



Figure 15: Dendrogram- a result of HAC.

Agglomerative clustering starts with *n* elements and merges the closest pair of elements into a single cluster. This process of merging the clusters continues until all the clusters are finally merged into a single cluster. While HAC follows a bottom-up approach, divisive clustering follows a top-down approach. It begins by grouping all the elements in a single cluster. It then repeatedly splits the clusters to the point where every cluster consists of a single element.

One of the key concerns of HAC is how to define the closest pair of clusters. A number of distance measures, like Single Link (MIN), Complete Link (MAX) and Average Link (Group Average) have been used to define the distance between any two clusters. All of these are suitable for graph based clusters. We have implemented the Group Average method to cluster the documents where the distance between two clusters is defined by the average distance between points in both the clusters.

A number of similarity measures exist that can be used to measure the distance between documents to find the closest or distant pair of documents to merge during HAC. Among them, cosine measure [38] is the most frequently used one which penalizes less in cases where the number of frequent subgraphs on each document differs significantly. Document clustering often compares documents of various sizes, and having different sizes does not guarantee that the documents are not similar. Since cosine measure focuses more on the component in the documents and is not influenced by the document length, it has been used widely in document clustering. We have chosen this measure to compute the similarity between two document-graphs:

$$Similarity_{cosine}(DG1,DG2) = \frac{count\left(FS(DG1) \cap FS(DG2)\right)}{\sqrt{count\left(FS(DG1)\right) \times count\left(FS(DG2)\right)}}$$

To cluster the documents we have used a dissimilarity matrix which stores the dissimilarity between every pair of document-graphs using the formula, *dissimilarity = 1 – similarity*. The value of *dissimilarity* can range from 0 to 1.

Evaluation Mechanism

We have compared our clustering results with the clustering results of the traditional frequency based system. Traditional frequency based system uses the frequency of keywords to measure the similarity between the documents. We used the following equation:

$$Similarity = \frac{\sum\left(f_{ai} \times f_{bi}\right)}{\sqrt{\sum\left(f_{ai}^{2}\right)\sum\left(f_{bi}^{2}\right)}}, \text{ where } f_{ai} \text{ is the frequency of keyword } w_i \text{ in } Doc_a \text{ and}$$

$f_{bi}$ is the frequency of keyword $w_i$ in $Doc_b$. The similarity value ranges from 0 to 1.

To measure the quality of clustering we have used average Silhouette Coefficient (SC) [38]. Given a document-graph *x* in a cluster *A*, *a(x)* is the average distance between the document-graph *x* and the other document-graphs in *A* while *b(x)* is the average distance between the document-graph *x* and the other document-graphs in a second cluster *B*. The Silhouette Coefficient of *x* is defined as [38]:

$$SC(x) = \frac{b(x)-a(x)}{max\{b(x),a(x)\}}$$

The value of the SC can range from -1 to 1. A negative SC value is undesirable since it denotes highly overlapping clusters. The other evaluations of the SC value of a cluster are described as [38]:

$0.7 < SC \leq 1$ = Strong seperation of clusters
$0.5 < SC \leq 0.7$ = Medium seperation of clusters
$0.25 \leq SC$ = No cluster clearly distinguishable

We have used average SC value to evaluate the results of our clustering. The overall quality of clustering can be measured by the average SC value of all clusters. It is calculated by taking the average of all the SC values of the document-graphs in a cluster. If there are $N$ document-graphs in cluster $j$ then,

$$Average\,SC_j = \frac{\left(\sum_{i=1}^{N} SC_i\right)}{N} \; .$$

CHAPTER 4

EXPERIMENTAL RESULTS

In this chapter we explain our experimental results and evaluate the efficiency of our modified FP-growth approach for document clustering. We have used two different subsets of 20NG [12] for our experiments: the first subset contains 500 documents from 5 groups and the second one consists of 2500 documents from 15 groups.

To make our FP-growth approach time efficient, we reduced the size of the original header table. The header table contains the edges that maintain the *min_sup* threshold. Each entry of this table contains an edge, its corresponding frequency and a pointer to the FP-tree. The header table is sorted according to the frequency of the edges in descending order. As a result, the edges at the top of the header table are the most frequent ones, usually representing links to very abstract concepts and edges at the bottom are the least frequent ones representing very specific concepts. Concepts that are very abstract appear in many documents while specific concepts tend to appear in a very small number of documents. Therefore, very-abstract and very-specific concepts are not good candidates for clustering. We consider the edges that appear between certain minimum support threshold values to make the clustering more meaningful.

We concentrated on the combination generation part of the FP-growth algorithm as a major modification (Table 5). As stated in Chapter 3, whenever a single path is encountered in the FP-tree (or recursively generated conditional FP-trees), each of its combinations is generated and checked to make sure that it follows the connectivity

constraint. The length of the single path can be as large as the number of all the edges appearing in a DG$_i$. Taking each combination of a large single path and checking the connectivity constraints is computationally intensive. The construction of the FP-tree forces the edges with higher frequencies to appear at the top of the FP-tree. So, it is more likely that nodes at the top levels of the FP-tree indicate edges at more abstract levels of the MDG. After observing that a higher abstraction level of MDG does not provide enough information for reliable clustering, we restricted the generation of combination of a single path of the FP-tree to a certain length. When we mine the FP-tree, we start from the edges appearing the least in the pruned header table. Thus, if we reach a single path of length greater than the single path threshold, we prune the upper part of the single path above that threshold and generate each combination of the lower part only. This mechanism prunes nodes of single paths of the FP-tree at the upper levels which are representative top level edges of the MDG.

<u>Experimental Results on Modified FP Growth Approach</u>

Table 7 to Table 11 contains some of our experimental results with the subset of 500 documents from 5 different groups of 20NG. The *min_sup* was kept 5% and we selected 397 edges out of 2092 edges (i.e. 19%) from the middle of the header table to use them in our FP-growth approach. After the pruning, the support of the most frequent edge in the header table was 481 whereas the support of the least frequent edge was 25.

Consider that the number of edges in the largest subgraph is *k* for a certain single path threshold *t*. Table 7 shows corresponding *k* and support count for the *k*-edge

Table 7: Largest subgraph with varying Single Path (SP) threshold

# of documents: 500, #of groups =5, *min_sup=5%*

| SP threshold ( t ) | Size of the largest k-edge subgraph ( k ) | Support of k-edge subgraphs |
|---|---|---|
| 2 | 3 | 476 |
| 4 | 5 | 455 |
| 6 | 7 | 432 |
| 8 | 9 | 363 |
| 10 | 11 | 327 |
| 12 | 13 | 245 |
| *14* | *15* | *95* |
| 16 | 17 | 122 |
| 18 | 17 | 122 |
| 20 | 20 | 1 |

subgraphs discovered with different single path threshold. It shows that as we increase the single path threshold (*t*), we get larger frequent subgraphs from the same FP-tree. Therefore, we can control the maximum size of the discovered subgraphs by varying the single path threshold. As the size of the largest *k*-edge subgraph increases, the frequency of its appearances in the document-graphs becomes smaller. In other words, frequent, larger subgraphs tend to appear in smaller number of DGs than the smaller ones. The second column of Table 7 shows the evidence: if the $k_m$-edge subgraph is the largest *k*-edge subgraph when the single path threshold is *m* and $k_n$-edge subgraph is the largest *k*-edge subgraph for the single path threshold *n*, then $k_n \geq k_m$, if *n>m*.

Due to the applied threshold of the single path we do not discover the set of all frequent subgraphs; rather our approach discovers subgraphs with maximum size controlled by the SP threshold. Moreover, an SP threshold of 16 does not guarantee that all discovered subgraphs are subsets of the set of the frequent subgraphs discovered using an SP threshold of 18. In Table 7, the row for SP threshold of 14 has a lower support than

Table 8: Frequency of subgraphs for different SP threshold.

# of documents: 500, #of groups =5, *min_sup=5%*

| SP threshold ($t$) | Frequency of all subgraphs |
|---|---|
| 2 | 1450 |
| 4 | 2384 |
| 6 | 3298 |
| 8 | 4130 |
| 10 | 4947 |
| 12 | 5727 |
| 14 | 6037 |
| 16 | 6621 |
| 18 | 6621 |
| 20 | 6728 |

its upper and lower rows. This can happen if the subset of the smaller subgraphs of *17*-edge subgraphs, discovered using SP threshold of 16, appears to be disconnected in a SP threshold of 14. This means that the 2 important edges were responsible for the connectivity of the edges when SP threshold was 16. As they are missing in the single path with SP threshold 14, the combinations of the single path appear to be disconnected.

Now we would describe the impact of SP threshold in total number of all *k*-edge subgraphs. Table 8 depicts the frequencies of all *k*-edge subgraphs discovered at different single path thresholds. As we increase the threshold, the number of discovered subgraphs also increases or remains the same. The number remains the same when added new edges are totally disconnected from the edges of the previous single path. Inclusion of these new edges in the single path does not result in additional subgraphs.

Increasing the single path threshold has different effects on the number of discovered frequent subgraphs. This is shown in Table 9 where *N(k)* is the number of *k*-edge frequent subgraphs discovered for a certain single path threshold. For the same *k*-

Table 9: Decomposition of subgraphs for different SP threshold.

# of documents: 500, #of groups =5, *min_sup=5%*

| SP threshold (t) | ….. | N(6) | ….. | N(9) | ….. | N(13) | ….. | N(17) | ….. | N(19) | N(20) |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | | 473 | ….. | ***425*** | ….. | 0 | ….. | 0 | ….. | 0 | 0 |
| 12 | | 479 | ….. | ***450*** | ….. | 245 | ….. | 0 | ….. | 0 | 0 |
| 14 | | 480 | ….. | 450 | ….. | 277 | ….. | ***0*** | ….. | 0 | 0 |
| 16 | ….. | 480 | ….. | 456 | ….. | 317 | ….. | ***122*** | ….. | 0 | 0 |
| 18 | ….. | 480 | ….. | 456 | ….. | 317 | ….. | 122 | ….. | 0 | 0 |
| 20 | ….. | 480 | ….. | 456 | ….. | 321 | ….. | 114 | ….. | 37 | 28 |

edge subgraph, the support of the subgraph increases as single path threshold is increased. For example, when SP threshold is 10, 425 *9*-edge subgraphs (i.e. **N(9)** in Table 9) are discovered. Increasing the SP threshold to 12 introduces 450 *9*-edge subgraphs. Also, higher order subgraphs are discovered as a result of increased single path threshold. So when we increase the SP threshold from 14 to 16, 122 new *17*-edge subgraphs (i.e. **N(17)** in Table 9) are discovered. For a specific SP threshold, larger subgraphs appear in fewer documents compared to the smaller ones.

Table 10 merges Table 7 and Table 8. It gives the overview of two of the previous

Table 10: Impact on largest subgraphs with varying SP threshold.

# of documents: 500, #of groups =5, *min_sup=5%*

| SP threshold (t) | Size *k* of the largest *k*-edge subgraph | # of unique largest *k*-edge subgraphs | Support of largest *k*-edge subgraphs | Frequency of all subgraphs |
|---|---|---|---|---|
| 2 | 3 | 4 | 476 | 1450 |
| 4 | 5 | 1 | 455 | 2384 |
| 6 | 7 | 1 | 432 | 3298 |
| 8 | 9 | 1 | 363 | 4130 |
| 10 | 11 | 2 | 327 | 4947 |
| 12 | 13 | 4 | 245 | 5727 |
| 14 | 15 | 2 | 95 | 6037 |
| 16 | 17 | 3 | 122 | 6621 |
| ***18*** | ***17*** | ***3*** | ***122*** | ***6621*** |
| 20 | 20 | 1 | 1 | 6728 |

Table 11: Subgraphs generated for each edge in the header table.

# of documents: 500, #of groups =5, *min_sup=5%*, (SP threshold =18)

| Row of header table | N(2) | N(3) | N(4) | N(5) | N(6) | ……. | N(14) | N(15) | N(16) | N(17) |
|---|---|---|---|---|---|---|---|---|---|---|
| ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. |
| 308 | 5 | 5 | 7 | 16 | 23 | ……. | 5 | 1 | 0 | 0 |
| ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. |
| 302 | 14 | 9 | 0 | 0 | 0 | ……. | 0 | 0 | 0 | 0 |
| ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. |
| 279 | 5 | 5 | 1 | 0 | 0 | ……. | 0 | 0 | 0 | 0 |
| ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. |
| 186 | 1 | 1 | 1 | 1 | 1 | ……. | 0 | 0 | 0 | 0 |
| ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. |
| 84 | 16 | 8 | 0 | 0 | 0 | ……. | 0 | 0 | 0 | 0 |
| ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. |
| 27 | 1 | 3 | 5 | 7 | 8 | ……. | 0 | 0 | 0 | 0 |
| ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. |
| 16 | 19 | 26 | 30 | 31 | 30 | ……. | 0 | 0 | 0 | 0 |
| ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. |
| 12 | 8 | 14 | 17 | 19 | 19 | ……. | 0 | 0 | 0 | 0 |
| ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. | ……. |
| Total | 312 | 387 | 446 | 534 | 583 | ……. | 154 | 71 | 21 | 3 |

tables in the current section. It also includes the number of unique largest *k*-edge subgraphs in its third column.

Table 11 gives a partial description of one row of Table 10. The corresponding row of Table 10 is highlighted by bold-italic text. In Table 11, a value *N(k)* for a certain row of the header table indicates the number of *k*-edge subgraphs discovered by our FP-growth approach, taking the edge of that row as the starting of a recursive decomposition.

Now, column **N(6)** of Table 11 shows that our algorithm has detected a total of 583 *6*-edge subgraphs. In contrast, Table 9 shows that **N(6)** is 480 when the single path threshold is 18. This number is different because **N(6)** of Table 9 indicates the number of unique subgraphs but **N(6)** of Table 11 indicates total subgraphs discovered by different

rows of the header table. In Table 11, some subgraphs discovered by the *i*-th row of the header table may be already encountered by a previous row *j* of the header table, where *i<j*. Therefore, the last row of Table 11 shows total number of subgraphs detected by all the header table rows, where Table 9 shows how many of them are unique. For example, a *3*-edge subgraph consisting of edges *{4, 5, 7}* will be discovered for each of the corresponding edges of the header table.

The behavior of the number of discovered *k*-edge subgraphs from Table 11 is shown in Figure 16. The number of *1*-edge frequent subgraphs is same as the number of edges appearing in the header table after pruning. We start the plot from *2*-edge subgraphs. This number increases as the size of subgraph increases, but after a certain point it starts to decrease since large subgraphs become rare in the document-graphs.

Figure 17 shows a similar plot for our second dataset with 2500 documents from 15 groups with 400 keywords. The single path threshold is 15 for Figure 17. There were
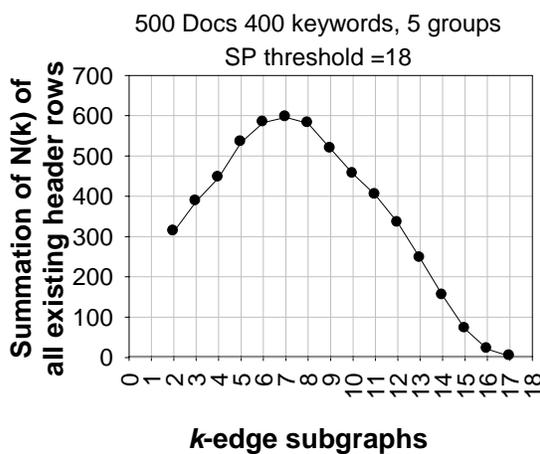
Figure 16: Discovered subgraphs of modified FP-growth (500 documents)
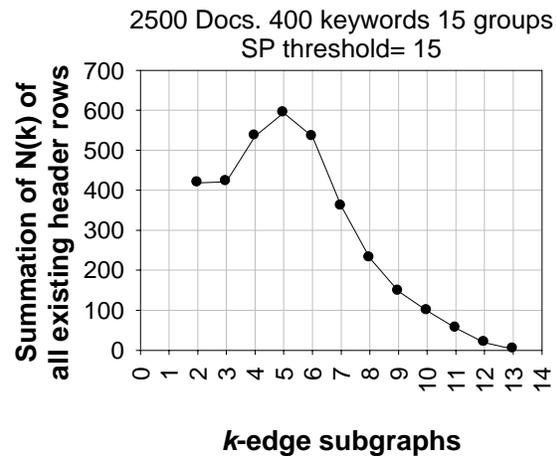
Figure 17: Discovered subgraphs of modified FP-growth (2500 documents)

2145 unique edges in the corresponding MDG of this dataset. After pruning the header table by the *min_sup* (5%) we further reduced the number of edges from the top and bottom of the header table because edges at the top of the header table are very frequent and edges at the bottom are very rare. There were 402 edges left in the header table from the mid level of it. After pruning, the most frequent edge of this table had a support of 2327 and the support of the least frequent edge was 122. As mentioned earlier in this section, we take 397 edges for 500 documents from the header table.

Figure 18 and Figure 19 show the corresponding runtime behaviors of frequent subgraph discovery process using our FP-growth approach. Figure 18 is for 500 documents and Figure 19 plots runtime for 2500 documents. As our FP-growth approach starts from the bottom of the existing header table with the least frequent edges, the X-axis of each of the plots indicates the sequence of the edges from the bottom of the header table. The plots show the time required for discovering all subgraphs for each
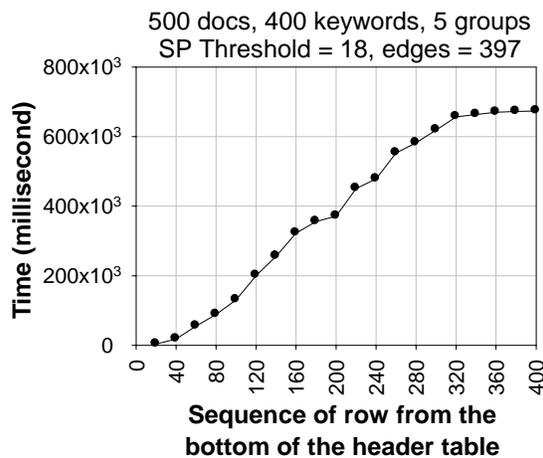


Figure 18: Running time for FP-Mining (500 documents, SP threshold=18).
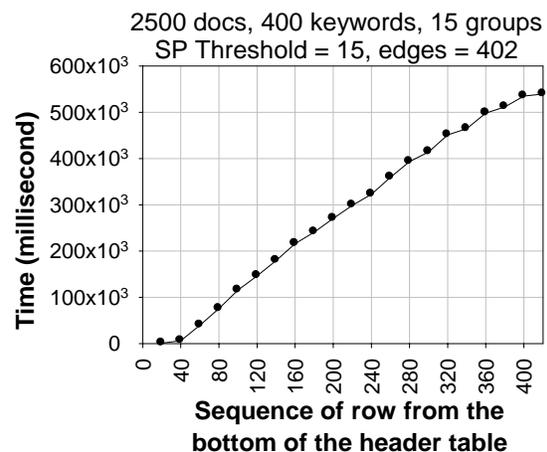
Figure 19: Running time for FP-Mining (2500 documents, SP threshold=15).
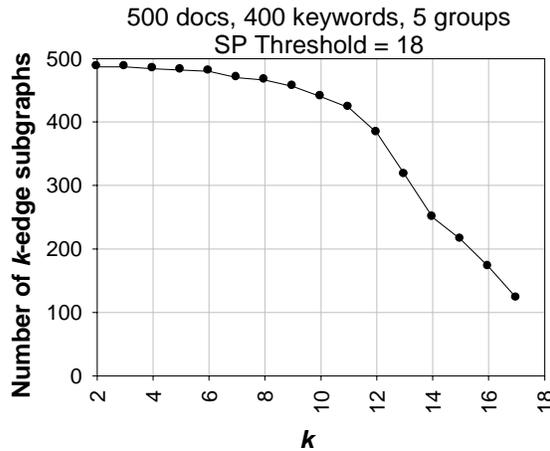
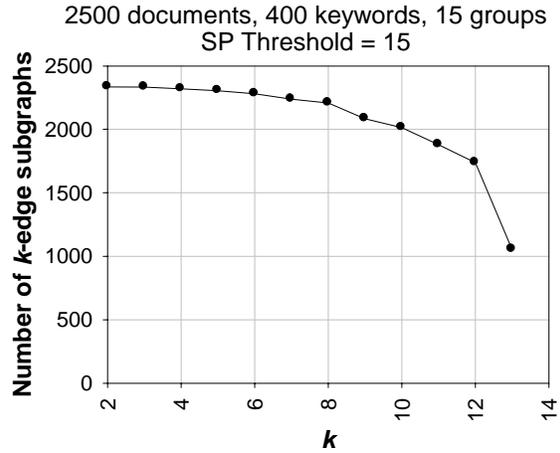Figure 20: Number of *k*-edge subgraphs
(500 documents).

Figure 21: Number of *k*-edge subgraphs
(2500 documents).

edge in the header table. The times plotted in the figures are cumulative. The figures

show that the execution time is almost linear in terms of time required to discover

subgraphs for each edge of the header table. Moreover, at the end the execution time

becomes almost constant. This is more noticeable in Figure 18. After discovering all

subgraphs up to 320 edges from the bottom, the time line becomes almost horizontal.

This happens because edges at the top of the header table have higher chance of pointing

to nodes at the top of the FP-tree. Nodes closer to the top of the FP-tree cause

construction of smaller conditional FP-trees. As a result, the time required to discover the

frequent subgraphs with edges from the top of the header table becomes small.

Figure 20 and Figure 21 show number of frequent *k*-edge subgraphs detected by

the FP-growth approach of Figure 18 and Figure 19 respectively. Both the figures show

that the larger the *k* (size of the subgraphs) the smaller the number of discovered *k*-edge

subgraphs.

Clustering

We used the frequent subgraphs discovered by the FP-growth approach of the previous section to cluster the corresponding document corpora with 500 and 2500 documents. We took different numbers of keywords from each dataset for the clustering. To investigate the accuracy of our clustering, we compared the results with the commonly used frequency-based clustering technique.

Table 12 shows the comparison between our approach and the traditional approach for the dataset with 500 documents. We calculated the average Silhouette coefficient for 5 clusters with different numbers of keywords from the same dataset. In

Table 12: Average SC coefficients for 5 clusters.

# of documents: 500, #of groups =5

| Number of keywords | SC coefficient for 5 clusters | | % Better $\left( \dfrac{X-Y}{X} \times 100 \right)$ |
|---|---|---|---|
| | FP-growth approach ($X$) | Traditional frequency based ($Y$) | |
| 200 | 0.20219064 | 0.054568320 | 73.01 |
| 300 | 0.21034962 | 0.052667454 | 74.96 |
| 400 | 0.22800271 | 0.024996204 | 89.04 |
| 500 | 0.34292123 | 0.021473328 | 93.74 |
| 600 | 0.25682740 | 0.035883773 | 86.03 |

Table 13: Average SC coefficients for 15 clusters.

# of documents: 2500, #of groups =15

| Number of keywords | SC coefficient for 15 clusters | | % Better $\left( \dfrac{X-Y}{X} \times 100 \right)$ |
|---|---|---|---|
| | FP-growth approach ($X$) | Traditional frequency based ($Y$) | |
| 200 | 0.442671 | 0.009031121 | 97.96 |
| 300 | 0.733246 | -0.007208774 | 100.98 |
| 400 | 0.430419 | -0.014922377 | 103.47 |
| 500 | 0.669504 | -0.013861327 | 102.07 |
| 600 | 0.562792 | -0.028308742 | 105.03 |

**500 documents, 5 clusters**

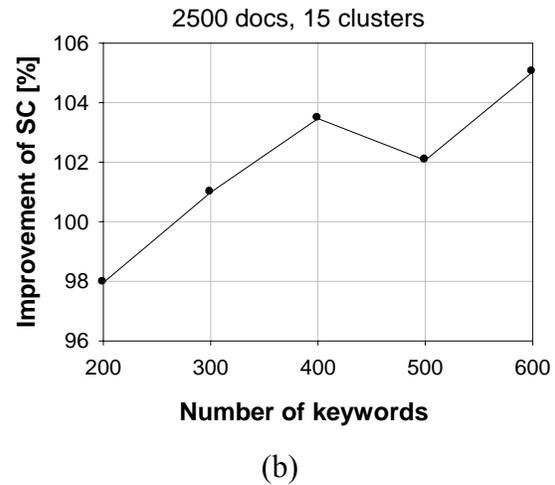**2500 docs, 15 clusters**

(a)

(a)

**500 docs, 5 clusters**

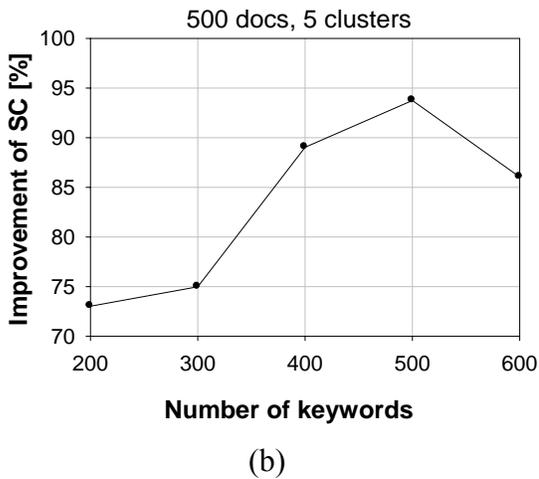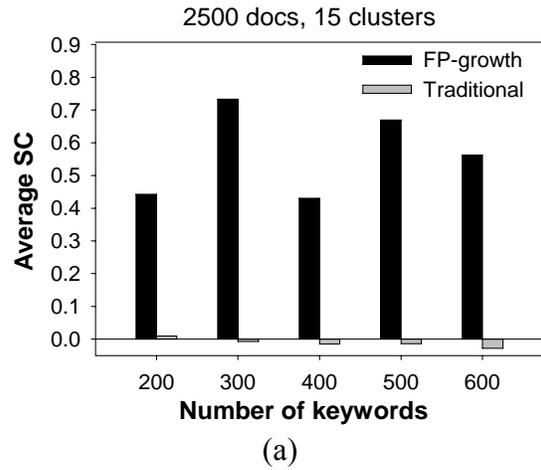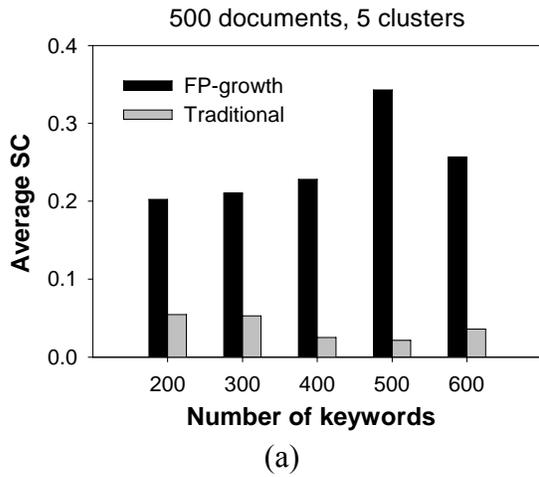**2500 docs, 15 clusters**

(b)

(b)

Figure 22: Comparison of clustering with 500 documents from 5 groups.

Figure 23: Comparison of clustering with 2500 documents from 15 groups.

every case with 500 documents, our approach provided better accuracy than the traditional vector model representation of text documents. We plot the Silhouette coefficients with different numbers of keywords in Figure 22(a) to better visualize the difference between the accuracies of our approach and the traditional approach. Figure 22(b) shows percentage gain of SC that represents a quantitative approach of how good our clustering mechanism is compared to the traditional one.

We show similar experimental results with 2500 documents from 15 groups in Table 13. As the dataset is larger than the previous one, but provided keywords are not sufficient for clustering 2500 documents, the clustering accuracy (average SC) is very low for traditional vector based model. Moreover, the average Silhouette coefficient is negative most of the time indicating inaccurate clustering. In contrast, our approach shows good clustering accuracy even with small numbers of keywords. Table 13 shows that our clustering approach is 105.03% better than the traditional approach when there are 15 clusters in 2500 documents with 600 keywords. Figure 23(a) shows the average SC of our clustering mechanism and the traditional clustering approach with side by side bars. Figure 23(b) presents percent improvement of the average SC value of our approach compared to the traditional clustering. With any number of keywords, our clustering approach is significantly better than the traditional frequency based clustering mechanism.

For the same number of documents, if we increase the number of keywords, the MDG has the tendency to contain more edges. More keywords better clarify the concept
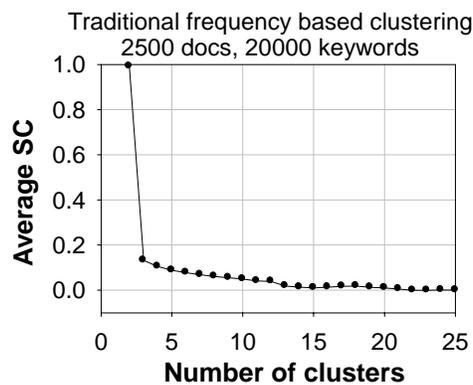


Figure 24: Average SC for traditional keyword frequency based clustering.

of the documents. In such case, the edges in the mid level of the MDG will have more connections between them. As a result, more subgraphs will appear in the middle of the MDG. So, our approach has the tendency to take the advantage of inclusion of keywords by discovering more subgraphs and using them in the clustering. The keyword frequency based traditional approach does not possess this tendency if small amounts of keywords are added. It requires inclusion of thousands of keywords for better clustering. The average SC for different number of clusters using traditional keyword frequency based approach for 2500 documents from 15 groups with 20,000 keywords is shown in Figure 24. With 20,000 keywords, the average SC was improved to only 0.0105 for 15 clusters. The highest average SC for the same dataset using our clustering mechanism was 0.733246 for 300 keywords. Therefore, our clustering mechanism is able to cluster the documents more accurately, even from low dimensional dataset.

CHAPTER 5

CONCLUSION

Document clustering is a very active research area of data mining. Numerous ideas have been explored to find suitable systems that cluster documents more meaningfully. Even so, producing more intuitive and human-like clustering remains a challenging task. In our work, we have attempted to develop a sense-based clustering mechanism by employing a graph mining technique. We have introduced a new way to cluster documents based more on the senses they represent than on the keywords they contain. Traditional document clustering techniques mostly depend on keywords, whereas our technique relies on the keywords' concepts. We have modified the FP-growth algorithm to discover the frequent subgraphs in document-graphs. Our experimental results show that modified FP-growth can be successfully used to find frequent subgraphs, and that the clustering based on the discovered subgraphs performs better than the traditional vector based model.

We started our research by using an association rule mining algorithm for mining frequent subgraphs. The original FP-growth algorithm for a market-basket dataset can efficiently discover the set of all frequent patterns. The traditional FP-growth algorithm does not fit to a non-traditional domain like graphs. When it is directly used in graph-mining, it mostly produces sets of disconnected edges. We have modified the FP-growth approach to ensure that it follows the connectivity constraint for all frequent pattern discovered. As a result, our discovered patterns are always connected subgraphs.

Additionally, we analyzed the single path behavior of the FP-growth approach for better performance. We then focused on the clustering of the documents based on their senses. We represent our documents as graphs generated by a hierarchical organization of the concepts of the related keywords. We believe that the frequent subgraphs discovered by our FP-growth approach reflect the concept of the documents better than the keywords. Thus, we use these frequent subgraphs for document clustering. Experimental results show that our clustering performs more accurately than one of the traditional document clustering mechanisms.

This work can be extended in a number of directions. Mining the FP-tree for larger single path lengths is still computationally expensive. We can generate some optimization techniques for computing the combinations of larger single paths which can improve the FP-growth performance. Additionally, devising an intelligent system to extract the useful edges from the header table remains as another future work.

## REFERENCES

[1]  Ahmed A. Mohamed, *Generating User-Focused, Content-Based Summaries for Multi-Documents Using Document Graphs*.

[2]  C. Borgelt and M.R. Berthold, *Mining Molecular Fragments: Finding Relevant Substructures of Molecules*. In: Proc. IEEE Int'l Conf. on Data Mining ICDM, Japan 2002, Pages: 51–58

[3]  X. Yan and J. Han., *gSpan, Graph–Based Substructure Pattern Mining*. In Proc. IEEE Int'l Conf. on Data Mining ICDM, Maebashi City, Japan, November 2002. Pages: 721–723

[4]  Michael R. Garey and David S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman. 1979, ISBN 0-7167-1045-5. A1.4: GT48, Pages: 202.

[5]  J. Han, X. Yan and Philip S. Yu, *Mining, Indexing, and Similarity Search in Graphs and Complex Structures*. ICDE 2006, Pages: 106

[6]  M. Cohen and E. Gudes, *Diagonally subgraphs pattern mining*. In: Workshop on Research Issues on Data Mining and Knowledge Discovery proceedings, 2004, Pages: 51–58.

[7]  Ping Guo, Xin-Ru Wang and Yan-Rong Kang, *Frequent mining of subgraph structures*. J. Exp. Theor. Artif. Intell., 2006, vol. 18, no. 4, Pages: 513-521.

[8]  J. Han, J. Pei and Y. Yin, *Mining frequent patterns without candidate generation*. In Proc. 2000 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'00), Dallas, TX, Pages: 1–12.

[9]  J. Han and M. Kamber, Data Mining: Concepts and Techniques, 2nd edition, ISBN 9781558609013.

[10]   R. Agrawal and T. Imilienski Swami. *Mining Association Rules between sets of items in large databases*. In Proc. Of  ACM SIGMOD, 1993.


[11]   L. David., *Naive (Bayes) at Forty: The Independence Assumption in Information Retrieval*. Proceedings of ECML-98, 10th European Conference on Machine Learning: 4-15, Chemnitz, DE: Springer Verlag, Heidelberg, DE


[12]   http://people.csail.mit.edu/jrennie/20Newsgroups


[13]   www.cs.cmu.edu/~mccallum/bow


[14]   http://www.wordnet.princeton.edu


[15]   R. Agrawal and R. Srikant., *Fast Algorithms for Mining Association Rules*. In Proc. of the 20th Int'l Conference on Very Large Databases, Santiago, Chile, September 1994.


[16]   D. J. Cook and L. B. Holder. *Substructure Discovery Using Minimum Description Length and Background Knowledge*. In Journal of Artificial Intelligence Research, 1994, vol. 1, Pages 231-255.


[17]   S. Kramer, L. Readt and C. *Helma: Molecular feature mining in HIV data.* In: Proceedings of KDD'01, 2001.


[18]   J. Huan, W. Wang, and J. Prins, *Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism*. In Proc. 2003 Int. Conf. Data Mining (ICDM'03), Mel-bourne, USA, December 2003.


[19]   M. Kuramochi and G. Karypis, *Frequent Subgraph Discovery*. In Proc. 2001 Int. Conf. Data Mining (ICDM'01), San Jose, Canada, November 2001.


[20]   A. Inokuchi, T. Washio, and H. Motoda, *An Apriori-based Algorithm for Mining Frequent Substructures from Graph Data*. In Proc. 2000 European

Conf. Principles and Practice of Knowledge Discovery in Databases (PKDD'00), Lyon, France, September 2000.

[21] J. Han and J. Pei, *Mining Frequent Patterns by Pattern-Growth: Methodology and Implications*, SIGKDD Explorations, 2000, vol. 2, no. 2, Pages: 14-20.

[22] S. Nijssen and J. N. Kok, *The Gaston Tool for Frequent Subgraph Mining*, Proc.of Int'l Workshop on Graph-Based Tools, 2004, vol. 127, no. 1, Pages: 77–87.

[23] X. Yan and J. Han. *CloseGraph: Mining closed frequent graph patterns*, In KDD, 2003

[24] A.-H. Tan., *Text mining: The state of the art and the challenges*. In Proc of the Pacic Asia Conf on Knowledge Discovery and Data Mining PAKDD'99 workshop on Knowledge Discovery from Advanced Databases, 1999, pages 65-70.

[25] Manu A. and Sharma C., *INFOSIFT: Adapting graph mining techniques for document classification*.

[26] G. Salton, A. Wong and C. S. Yang, *A Vector Space Model for Automatic Indexing*, Communications of the ACM, 1975, vol. 18, no. 11, Pages 613–620.

[27] Charles T. Meadow, *Text Information Retrieval Systems*. Academic Press, 1992.

[28] van Rijsbergen, C. J. *Information retrieval*. Butterworths, 1979

[29] Gerard Salton, *Automatic Text Processing*. Addison-Wesley Publishing Company, 1988.

[30] Junji Tomita, Hidekazu Nakawatase and Megumi Ishii, *Graph-based text database for knowledge discovery*. WWW (Alternate Track Papers & Posters) 2004, Pages: 454-455

[31] Manuel Montes-y-Gómez, Alexander F. Gelbukh and Aurelio López-López, *Text Mining at Detail Level Using Conceptual Graphs*. ICCS 2002, Pages: 122-136.

[32] J. F. Sowa and E. C. Way, *Implementing a Semantic Interpreter Using Conceptual Graphs*, IBM J. Research and Development, 1986, vol. 30, Pages: 57-69.

[33] Ahmed A. Mohamed, *Generating User-Focused, Content-Based Summaries for Multi-Documents Using Document Graphs*.

[34] http://www.csc.liv.ac.uk/~frans/KDD/Software/DataSets/newsGroupsTestData.html

[35] M. Shahriar Hossain and Dr. Rafal A. Angryk, *GDClust: A Graph-Based Document Clustering Technique*, IEEE International Conference on Data Mining (ICDM'07), IEEE ICDM Workshop on Mining Graphs and Complex Structures, IEEE Press, USA, 2007. Pages: 417-422.

[36] P. Tan, M. Steinbach and V. Kumar, Introduction to Data Mining, 1st edition, ISBN: 9780321321367, Addison-Wesley 2005.

[37] Kenneth H. Rosen, *Discrete Mathematics and Its Applications*, 2nd edition (NY: McGraw-Hill, 1991), Pages: 284-286

[38] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: an Introduction to Cluster Analysis*, John Wiley & Sons, 1990.

APPENDIX A

AN EXAMPLE OF FP-TREE CONSTRUCTION

In this appendix, we show an example of the construction of the FP-tree with the edges of three document-graphs DG1, DG2 and DG3. The document-graphs and the
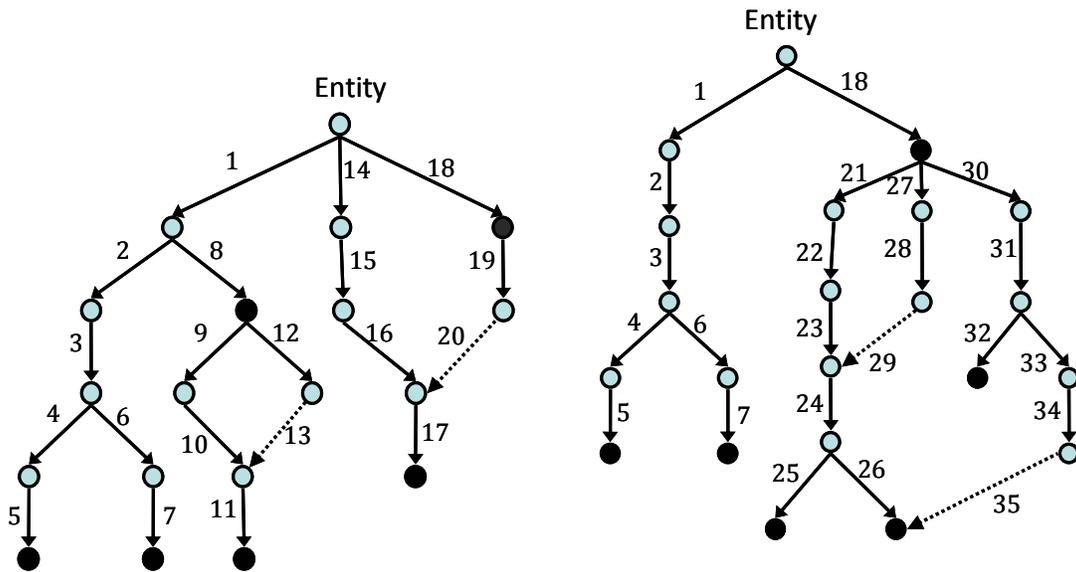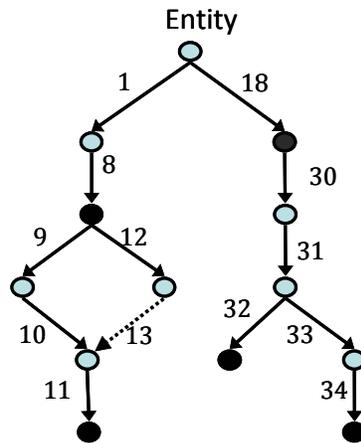


**DFS edge ids of DG 1:**
1, 18, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20.

**(a)**

**DFS edge ids of DG 2:**
1, 18, 2, 3, 4, 5, 6, 7, 30, 31, 32, 33, 34, 21, 22, 23, 24, 25, 26, 27, 28, 29, 35

**(b)**

**DFS edge ids of DG 3:**
1, 18, 8, 9, 10, 11, 12, 13 ,30, 31, 32, 33, 34

**(c)**

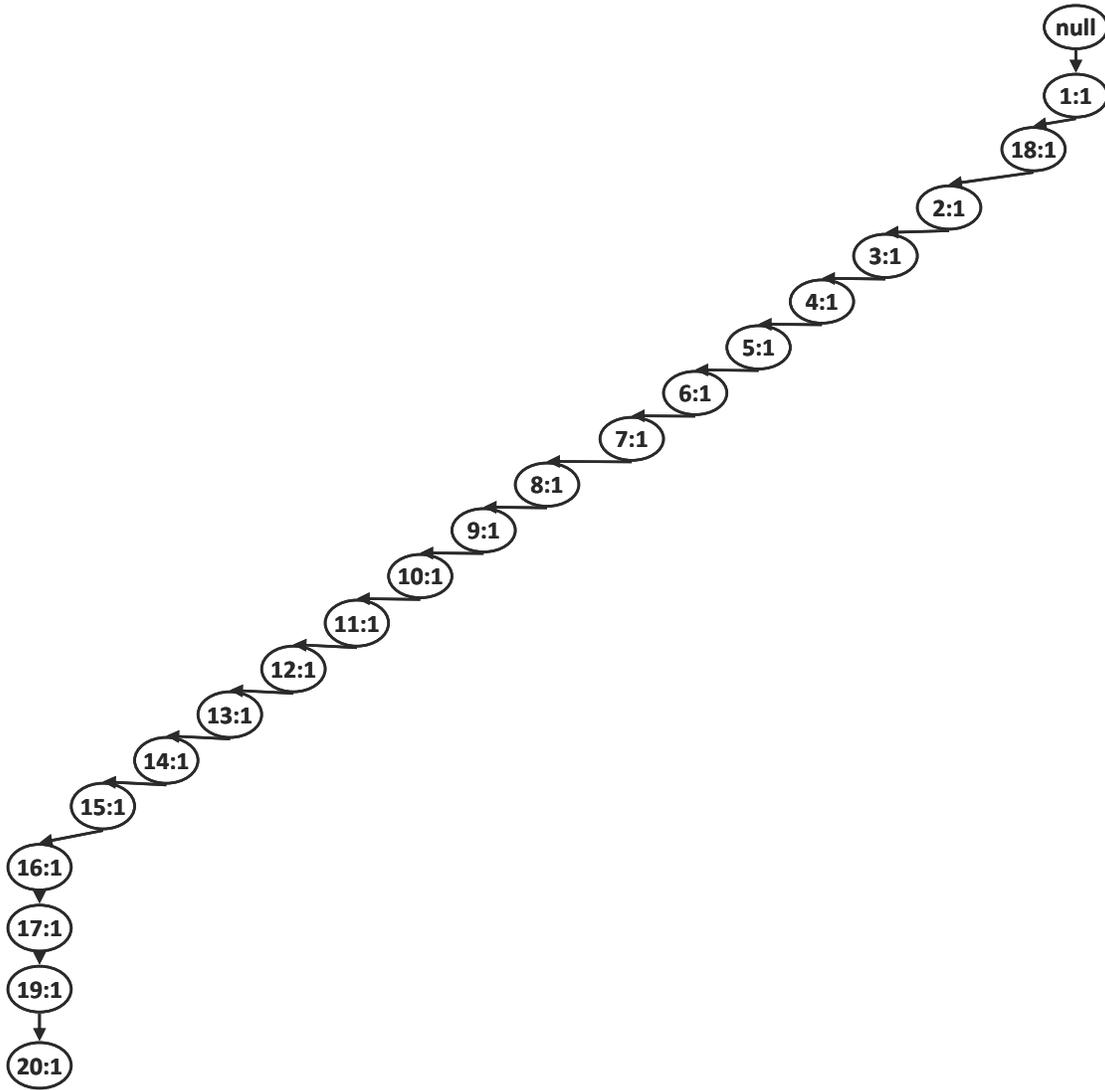Figure 25: Three document-graphs with corresponding DFS edge ids.

Figure 26: FP-tree after inserting the edges of DG1.

corresponding sorted edge-lists are shown in Figure 25 (a), (b) and (c) respectively.

Figure 26 shows the FP-tree after inserting the edges of DG1.

Since DG1 and DG2 contain common edges at the starting of their sorted edge-lists, the insertion of the new nodes in the FP-tree begins after DFS edge id 7. This is shown in Figure 27. The support count is increased accordingly for the already existing edge ids in the FP-tree. The FP-tree nodes containing the common edge ids are marked

by gray color in Figure 27. Now, we insert the edges of DG3 in the FP-tree. The first two edges are already present in the existing FP-tree. So, the insertion of the new nodes begins after DFS edge id 18 in the FP-tree. The final FP-tree after the insertion of all the edges of DG3 is shown in Figure 28. The gray nodes of Figure 28 indicate the already existing edge ids. For simplicity, we have not shown the header table and the node-links in this example.
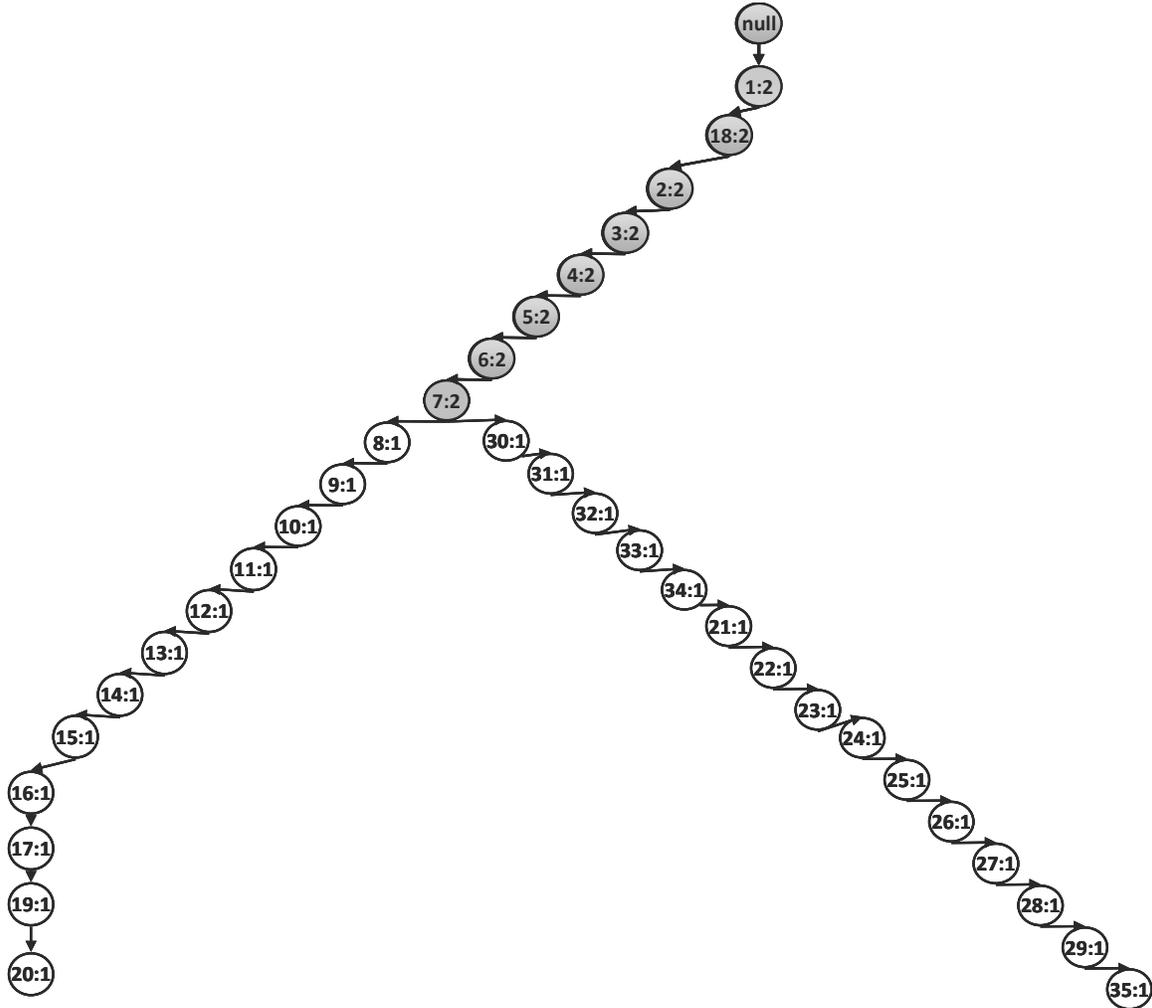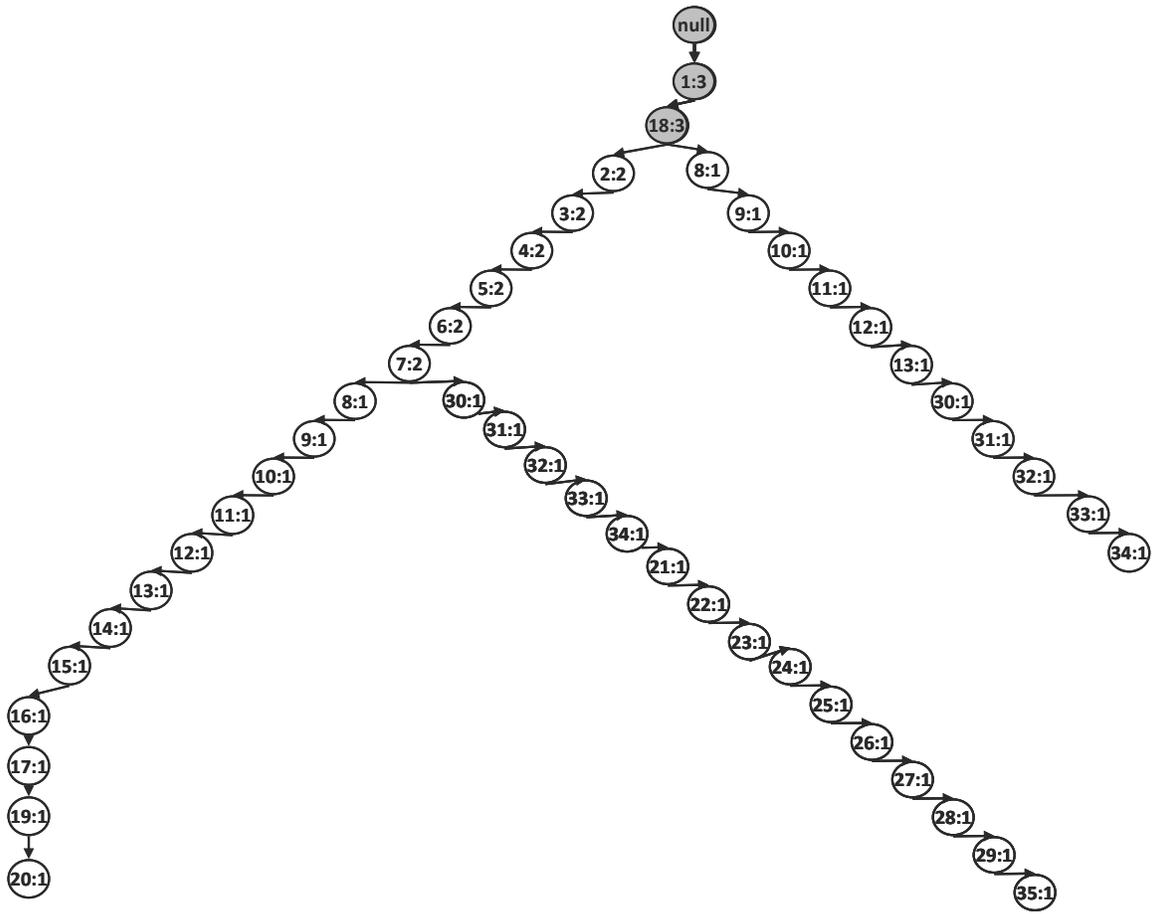


Figure 27: FP-tree after inserting the edges of DG2.

Figure 28: FP-tree after inserting the edges of DG3.