



ESE 589: LEARNING SYSTEMS

PROJECT 2:  
FREQUENT PATTERN MINING USING  
FP-GROWTH ALGORITHM

GUIDED BY:  
Dr. ALEX DOBOLI

PREPARED BY:

VARUN JAIN (111685202)  
AMOD GANDHE (111678938)  
SHASHANK RAO (111471609)

## **I. Abstract:**

For large databases, the research on improving the mining performance and precision is necessary, so many focuses of today on association rule mining are about new mining theories, algorithms and improvement to old methods. Association rules mining is a function of data mining research domain and arise many researchers interest to design a high efficient algorithm to mine association rules from transaction database. Generally, all the frequent item-sets discovery from the database in the process of association rule mining shares of larger, the price is also spending more. Discovering association rules is a basic problem in data mining. Finding frequent itemset is the most expensive step in association rule discovery. Analyzing a frequent pattern growth (FP-growth) method is efficient and scalable for mining both long and short frequent patterns without candidate generation. In this paper we present an way of implementing one of the efficient and robots pattern mining algorithm FP-Growth.

## **II. Introduction:**

Data mining has long been an active area of research in databases. The day by day decreasing cost and compactness of storage devices has made it possible to store every transaction of a transactional database. This storage solves two problems first they can access the data any times second this data helps them to find relationship among data items. The problem of finding relationship among different data items was first introduced by Agarwal et. Al. The solution to this problem can help to enhance the earnings, optimized storage. There can be different kind of databases available such as, active database, cloud database, embedded database and transactional database etc, but pattern extraction basically deals with a transactional database. A transactional database is a database in which there is no auto commit. There are two layout which are in common use, horizontal layout and the vertical layout. In horizontal layout there are two columns. First represents the transaction id and second represents the items bought in that transaction. It is a divide and conquer mechanism which reduces the size of database recursively by considering only the longest pattern. A frequent pattern is a pattern which occurs in comparatively more transactions. A frequent itemset is an itemset whose support is greater than some user-specified minimum support. FP-Growth is based on these principles which heavily relies on FP-Tree. It is nothing but and tree representation of the database. Further details of the FP-Growth will be discussed ahead in the paper. This paper is sectioned into 'n' parts. Related work, Generic algorithm, proposed solution to the problem, experimental results and finally conclusion.

### III. Related work:

**Naïve Approach for pattern extraction:** Intuitively, pattern extraction or finding out most frequent pattern, must deal with a dataset. Initially when the information size of data was limited, it was somewhat logical to extract all the patterns out of the given dataset. While, as Computer Science field proliferated, one had to deal with large dataset. In case of naïve approach, memory requirement is high also time for searching pattern will be relatively high but considerable for small dataset size. But as size of information increased, calculating and storing all the pattern was difficult and practically impossible. This required introduction of newer algorithm to handle such large datasets. In this section we would concentrate on providing outline for some non-trivial algorithms.

**Apriori based Pattern Extraction:** As it is stated, Frequent -pattern mining plays an essential role in mining associations, correlations, causality plays an important part in data mining task. Acknowledging this there has been vast development in algorithms for pattern extraction. Most of the previous work in this field weather by Agrawal & Srikant or from Mannila et al. or many such authors adopts apriori like approach. Apriori approach revolves around pruning patterns in dataset before they are calculated. This can is supported by an principle: *“If any length  $K$  pattern is not frequent in database then its  $(K+1)$  super pattern can never be frequent”*.

Following this approach definitely gives benefit that, there is no need to calculate each and every candidate in database i.e. ultimately reducing size of candidates generated. Intuitively, this approach has severe restrictions when dataset is very large, or it has long patterns or minimum support threshold is very low. In such cases, Apriori requires more resource as:

- Handling huge number of candidates is in itself is non-trivial task.
- Pattern matching in case of large database by repeated scans is a hectic.

**ECLAT-Equivalence class transformation:** Finding association rule by making use of apriori algorithm is still much more efficient compared to Naïve approach. But as discussed earlier, Apriori causes much more trouble when dealing with large dataset. This is due to its approach to search for pattern in Breadth first manner.

ECLAT (Equivalence CClass Transformation) is considered to be an improvement over Apriori. This algorithm differs from previous algorithm in a way that is constructs a transaction table in a vertical manner. FP-Growth follows Horizontal table approach which will be discussed in detail in next sections. Another major difference than needs to be considered is, ECLAT is depth first search algorithm. This improves the memory requirements for candidate generation.

General comparison between previously mentioned algorithm is made below:

	Apriori	ECLAT	FP-Growth
Dataset consideration	Horizontal	Vertical	Horizontal
Traversing order	Breadth first search	Depth first search	Bottom up approach
Running Time (Low Min_Sup value)	High	Moderate	Low
Number of frequent itemset generated	Equal	Equal	Equal
Memory requirement	Very high	Better than Apriori	Best case

*Table 1: Comparison between different mining algorithm*

Though it seems like FP-Growth outperforms other two algorithms in all aspects, there are some conditions in which Apriori performs better than FP-Growth. One of those condition can be: transactions are smaller in size, dataset is sparse and some other conditions. So, in all it can be concluded, performance of these dataset highly depends on the dataset structure and cannot be decided solely on directions followed by algorithm.

#### **IV. Generic Algorithm:**

The FP-Growth Algorithm employs a partitioning-based, divide-and-conquer method which transforms the problem of finding long frequent patterns into looking for shorter frequent patterns and then concatenating suffixes. It employs least frequent items as the suffix, which offers good selectivity. These methodologies result in dramatic reduction of search costs of the frequent patterns. The FP-Growth algorithm involves the following generic steps:

##### **Construction of Frequent-Pattern tree data structure:**

Consider that  $\{a_1, a_2, \dots, a_i\}$  be a set of items  $I$  which occur in a transaction database TDB, where  $\langle T_1, T_2, \dots, T_n \rangle$  are the  $n$  different transaction ids of TDB. Each transaction id of the TDB has a particular set of items, which is a subset of  $I$ . This means that the initial dataset provided has  $n$  tuples, and each tuple has some (or all) elements from  $I$ . The support (or occurrence frequency) of pattern  $P$  is the number of transactions containing  $P$  in TDB. The pattern  $P$  can be said to be frequent if  $P$ 's support is greater

than a predefined minimum support threshold MST. In order to create the Frequent-pattern tree data structure, the TDB is scanned twice.

During the first scan of TDB, an ordered list of frequent items along with their count is obtained. For example, if the support of the following items -  $(a_2, a_4, a_5)$  – in TDB is less than MST, then these items will not be considered for frequent pattern mining and will not be included in the list of frequent items created during the first scan. This will result in a list such as  $\langle (a_1:x), (a_3:y), (a_6:z), \dots, (a_i:w) \rangle$ , where  $x,y,z,w$  are the supports for the respective items (which are greater than MST). This list is also ordered in frequency-descending order, i.e.,  $(x \geq y \geq z \geq w)$ . This ordering is important since each path of the FP-tree will follow this order.

Following this, the TDB is scanned for a second time. Initially, the root of the FP-tree is created and labeled with “root”. Suppose that the  $T_1$  tuple of TDB consists of the items  $\{a_1, a_2, a_5, a_6, a_7\}$ ;  $T_2$  tuple of TDB consists of the items  $\{a_1, a_3, a_4, a_8, a_{10}\}$ ;  $T_3$  tuple of TDB consists of the items  $\{a_1, a_6, a_8\}$ ;  $T_4$  tuple of TDB consists of the items  $\{a_3, a_{10}\}$ . During the second scan of TDB, each tuple is read and added as a path from the created root of the FP-tree using the ordered list created during the first scan. First the  $T_1$  tuple is read and the items  $(a_1, a_6, a_7)$  are added to the root as  $\langle \text{PATH 1 : root} \rightarrow (a_1:1) \rightarrow (a_6:1) \rightarrow (a_7:1) \rangle$ , where 1 represents the current count of the item and order from the root to the leaf is maintained from greatest count to the least count of the ordered list. Since items  $(a_2, a_5)$  have already been excluded from the ordered list, they are not considered for the creation of the FP-tree. Next  $T_2$  tuple is considered. This will add items  $(a_1, a_3, a_8, a_{10})$  to the root in the following way:  $\langle \text{PATH 2 : root} \rightarrow (a_1:2) \rightarrow (a_3:1) \rightarrow (a_8:1) \rightarrow (a_{10}:1) \rangle$ . Since  $a_1$  item already exists, its count is increased and then the rest of the items are added to a new path. Similarly, reading  $T_3$  tuple results in creation of  $\langle \text{PATH 3 : root} \rightarrow (a_1:3) \rightarrow (a_6:2) \rightarrow (a_8:1) \rangle$  path of the FP-tree and reading  $T_4$  tuple results in creation of  $\langle \text{PATH 4 : root} \rightarrow (a_3:1) \rightarrow (a_{10}:1) \rangle$  path of the FP-tree. Along with the creation of each of these separate paths, links will also be created between same elements present in different paths, i.e., between  $a_3$  and  $a_{10}$  of PATHS 2 and 4 and between  $a_8$  of PATHS 2 and 3. The heads of each of the item is stored in a header list. These headers will point to the occurrence of the item in the tree. This methodology is continued for all transaction ids in the TDB in order to obtain the final FP-tree data structure. The following image shows the FP-tree structure created with the above example of 4 transaction ids (solid line show the path and dotted lines show the lines between the same elements in different paths; the header list is also shown):

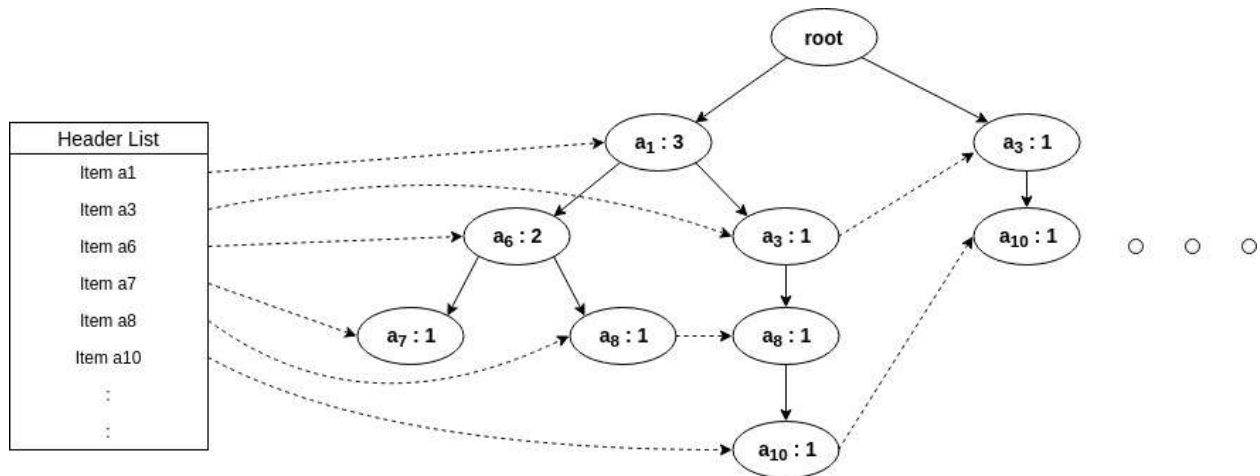


Figure 1: FP-Tree construction

### **Mining frequent patterns from FP-tree data structure:**

Once the FP-tree data structure has been created, frequent-pattern growth algorithm, FP-growth, is used for mining the complete set of frequent patterns from the FP-tree structure. The above example will be used to explain how the frequent mining pattern is obtained. The leaf node is initially considered. For node ( $a_{10}$ ), its immediate frequent pattern is ( $a_{10} : 2$ ) and it has two paths in the FP-tree:  $\langle a_1:3, a_3:1, a_8:1, a_{10}:1 \rangle$  and  $\langle a_3:1, a_{10}:1 \rangle$ . The first path indicates that the pattern ( $a_1, a_3, a_8, a_{10}$ ) appears once in TDB. ( $a_1$ ) itself appears 3 times but it appears only once together with ( $a_{10}$ ). Hence, to study which pattern appears with ( $a_{10}$ ), only ( $a_{10}$ )'s prefix path, i.e.,  $\langle a_1:1, a_3:1, a_8:1, a_{10}:1 \rangle$  or simply  $\langle a_1a_3a_8a_{10}:1 \rangle$ , is considered. The second path indicates that the pattern ( $a_3, a_{10}$ ) appears once in the TDB; in this case, ( $a_{10}$ )'s prefix path is simply  $\langle a_3a_{10}:1 \rangle$ . These two prefix paths of ( $a_{10}$ ),  $\{(a_1a_3a_8:1), (a_3:1)\}$ , form ( $a_{10}$ )'s subpattern-base, which is called ( $a_{10}$ )'s conditional pattern base. Construction of an FP-tree on ( $a_{10}$ )'s conditional pattern base (with MST considered as 2), leads to only one branch ( $a_3:2$ ). Hence only one frequent pattern ( $a_3a_{10}:2$ ) is derived. After this, the search for the frequent patterns associated with ( $a_{10}$ ) terminates.

For the node ( $a_8$ ), its immediate frequent pattern is ( $a_8 : 2$ ) and it has two paths in the FP-tree:  $\langle a_1:3, a_6:2, a_8:1 \rangle$  and  $\langle a_1:3, a_3:1, a_8:1 \rangle$ . ( $a_8$ ) appears together with ( $a_{10}$ ) as well, but there is no need to include ( $a_{10}$ ) here in the analysis since any frequent patterns involving ( $a_{10}$ ) has been analyzed in the previous examination of ( $a_{10}$ ). Similar to the above analysis, ( $a_8$ )'s conditional pattern base is  $\{(a_1a_6:1), (a_1a_3:1)\}$ . Constructing an FP-tree on ( $a_8$ )'s conditional pattern base (with MST considered as 2), leads to only one branch again ( $a_1:2$ ). Hence only one frequent pattern ( $a_1a_8:2$ ) is derived. After this, the search for the frequent patterns associated with ( $a_8$ ) terminates. Applying similar analysis on node ( $a_7$ ), leads to ( $a_7$ )'s conditional pattern base to be  $\{(a_1a_6:1)\}$ . Since this

does not pass the considered MST, there are no frequent patterns associated with ( $a_7$ ) and the search for the frequent patterns associated with ( $a_7$ ) terminates. Similar analysis on node ( $a_6$ ) yields ( $a_6$ )'s conditional pattern base as  $\{(a_1:2)\}$ . Hence, one frequent pattern ( $a_1a_6:2$ ) is derived and the search for the frequent patterns associated with ( $a_6$ ) terminates. Performing similar analysis on node ( $a_3$ ) yields ( $a_3$ )'s conditional pattern base as  $\{(a_1:1)\}$ . Hence, there is only one frequent pattern ( $a_3:2$ ).

This methodology is continued until all the items have frequent patterns associated with them. In case a node has multiple paths associated with them, the node is recursively mined in order to obtain all combinations of frequent patterns associated with that particular node. A single path FP-tree can be mined by outputting all the combinations of the items in the path. Once all the patterns are obtained, the algorithm terminates.

## V. Implementation of the Algorithm

This algorithm suggests its basic use case for Transitional databases. While there is some other type of datasets which are not exactly transactional, to support these kind of dataset, user will be asked to confirm if dataset is transactional or not. If it is, then attributes will not be considered column wise, while for non-transactional databases, we are identifying each column by and unique identifier which will be discusses ahead.

The execution of the algorithm begins at *FP\_Growth.java* file which contains the *main()* method. Initially, the file path containing the dataset is established and first scan of the dataset is performed by calling the *Util.calFreq()* method. This method will get a frequency count of all the items present in the database. We have used *HashMap<String, Integer>* to store this frequency count. The String key of the HashMap contains the item\_attribute identifier, i.e.,  $\langle \text{item} \rangle\_ \langle \text{columnNumber} \rangle$  is used to count the occurrences of  $\langle \text{item} \rangle$  in the particular  $\langle \text{columnNumber} \rangle$ ; the Integer part of the HashMap is used to store the frequency count.

Next, the algorithm begins the second scan of the transaction dataset. *Util.readCSV()* method is used to perform the second scan. This function takes the above created HashMap, file path and minimum support as parameters and returns an object of the type *ArrayList<Attribute>*. Here, Attribute is a object defined in the *Attribute.java* class; within this class file, Attribute object is defined as an ArrayList of String used to hold the different nodes of the transaction database. This *ArrayList<Attribute>* contains the values stored in the database according to frequency, after the application of minimum support on the database, i.e., this object contains the ordered frequent item list.

After obtaining the ordered list, this is passed to the constructor of FP\_Tree object (defined in *FP\_Tree.java*). This class contains two class variables – *root* of *Node* type and *dataset* of *ArrayList<Attribute>* type. The Node type is defined in *Node.java* and it contains the following class parameters -



- *name* of type String denoting the node name
- *count* of type Integer denoting the count of the node in the FP-tree
- *parent* of type Node which is a pointer to the parent node of the current node
- *children* of type ArrayList<Node> which is a list of all children from the current node
- *endOfTuple* of type Boolean which is used to denote the leaf node of the FP-Tree

This Node type is used to create each node of the FP-tree data structure. Initially, the ordered list created is passed to the constructor of *FP\_Tree.java* which will fill the *dataset* class variable defined in *FP\_Tree* class. After this, *FP\_tree.parseTree()* method of *FP\_Tree* object is called which will process each tuple for constructing the FP-Tree. From the *parseTree()* method, *insertNode()* method is called which checks if element is present as a child of the current node (*checkChildPresent()* method), adds node to the header table (*headerTable()* method), and inserts a new node into the FP-Tree or increments existing count (depending on the current transaction tuple being processed). *ParseTree()* method also makes use of *Links.java* objects, which is a *LinkedList<Node>* used to keep track of the same elements present in different paths of the FP-Tree. The *parseTree()* method returns an object of type *Model* (defined in *Model.java*; which contains the root *Node* of the FP-Tree and *HashMap<String, Links>* of the header table). This object returned will be the complete FP-Tree created and is stored in a *Model* object in the *main* method.

Once the FP-Tree has been created, FP-Growth algorithm is performed by calling *fpGrowth()* method of *Algorithm.java* class. Within this method, each node of the FP-Tree starting from the leaf is recursively tested in order to obtain the frequent patterns. *fpGrowth()* method returns an object of type *ArrayList<Result>* which is a hashmap of all frequent patterns present in the transaction database. *Result* object is defined in *Result.java* class and it contains *ArrayList<String>* to hold the frequent patterns and a *String* variable to hold the item value whose different frequent patterns are extracted. This is displayed as the final output to the end user by using *Util.displayPatt()* method.



## VI. Example Illustrating our Implementation:

We have demonstrated a small dataset example to illustrate the way in which our implementation works. Consider the following dataset available with us for a retail store consisting of purchases by 5 different customers:

<u>Items available</u>	<u>Purchase 1</u> <u>(T1)</u>	<u>Purchase 2</u> <u>(T2)</u>	<u>Purchase 3</u> <u>(T3)</u>	<u>Purchase 4</u> <u>(T4)</u>	<u>Purchase 5</u> <u>(T5)</u>
Bread	1	1	1	1	0
Onions	1	0	0	1	0
Gum	1	1	0	1	0
Peppermint	1	1	0	0	0
Fish	1	0	0	0	1
Cigarettes	1	1	1	1	1
Beer	1	0	0	0	1
Hammer	0	1	1	0	0
Chicken	0	1	0	0	0

*Table 2: Non-Transactional Dataset*

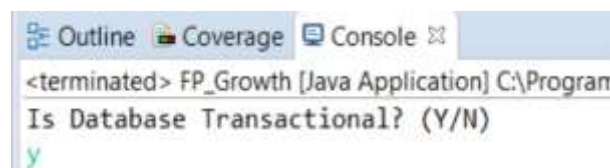
In this dataset, 1 stands for item purchased during the transaction and 0 stands for item not purchased during the transaction. This is the type of datasets available in most of the benchmarks. Using this dataset, we wish to obtain the frequent patterns of the items

which were purchased. However, for better illustration of this example, we have rearranged the above dataset as the following transactional dataset:

<u>Transaction Count</u>	<u>Items purchased during the transaction</u>
T1	Bread, onions, gum, peppermint, fish, cigarettes, beer
T2	Bread, hammer, gum, peppermint, chicken, cigarettes, nails
T3	Bread, hammer, cheese, diapers, tomato, cigarettes, chocolate
T4	Bread, onions, milk, diapers, juice, cigarettes, gum
T5	Beer, knives, milk, cigarettes, chocolate, beer, fish

*Table 3: Transactional Dataset*

Our implementation of the code can handle both types of table described above. Initially when the code is run, user is asked if the dataset being processed is transactional or not. If the user replies no, the first dataset is used for processing frequent patterns and if the user says yes, the second dataset is used for processing. The following figure shows input taken from the user:



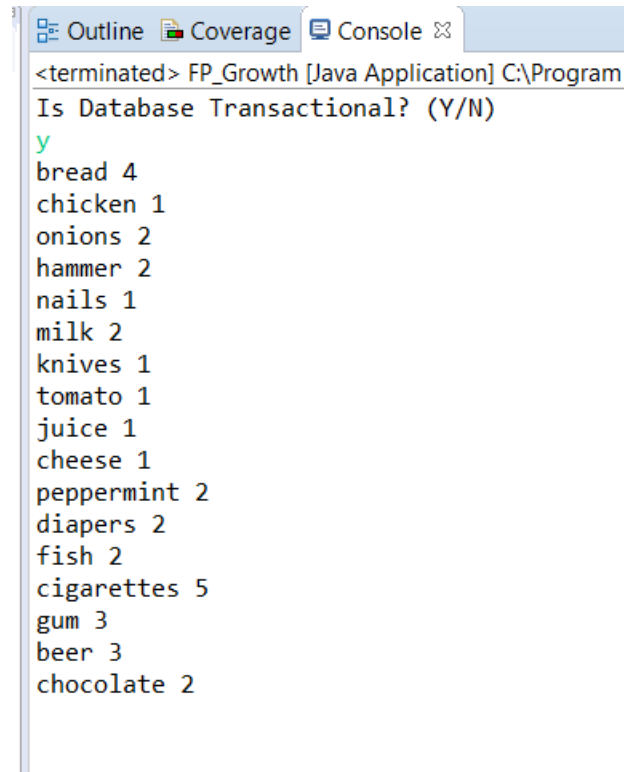
*Figure 2: UI for selecting dataset type*

We have used the transactional dataset to explain this example. The following figure gives an example .csv file created using the above dataset which will be parsed:

bread	onions	gum	peppermin	fish	cigarettes	beer
bread	hammer	gum	peppermin	chicken	cigarettes	nails
bread	hammer	cheese	diapers	tomato	cigarettes	chocolate
bread	onions	milk	diapers	juice	cigarettes	gum
beer	knives	milk	cigarettes	chocolate	beer	fish

*Figure 3: Example .csv file*

In our implementation, this dataset given to us is first parsed and count frequency of each element is extracted by calling the *Util.calFreq()* method. The following figure shows the extracted frequency count:



```
<terminated> FP_Growth [Java Application] C:\Program
Is Database Transactional? (Y/N)
y
bread 4
chicken 1
onions 2
hammer 2
nails 1
milk 2
knives 1
tomato 1
juice 1
cheese 1
peppermint 2
diapers 2
fish 2
cigarettes 5
gum 3
beer 3
chocolate 2
```

Figure 4: Initial frequency count of all items of the dataset

Once the frequency count has been extracted and stored in *HashMap<String, Integer>*, we continue with the creation of FP-Tree. *Util.readCSV()* method is called, which stores the hashmap values in *ArrayList<Attribute>* variable '*dataset*' which is used to create the FP-Tree. *FP\_Tree.parseTree()* method creates the tree and stores tree in a *Model* object. The following figure displays the FP-tree stored in the *Model* object without the header table:

```

Outline Coverage Console
<terminated> FP_Growth [Java Application] C:\Program Files\Java\jre-9.0.1\bin\javaw.exe (Apr 17, 2018, 10:12:46 PM)
Is Database Transactional? (Y/N)
y
root:1
cigarettes:5
bread:4
beer:1
beer:1 gum:2 chocolate:1 chocolate:1
gum:1 peppermint:1 milk:1 diapers:1 milk:1
fish:1 hammer:1 diapers:1 hammer:1 fish:1
peppermint:1 nails:1 onions:1 cheese:1 knives:1
onions:1 chicken:1 juice:1 tomato:1

```

Figure 5: FP-Tree created; Header table not displayed

Once the FP-Tree is created, FP-Growth algorithm is applied on the data structure by calling *Algorithm.fpGrowth()* method. We have set the minimum support at 0, so as to display all the frequent patterns which are obtained. The following figure shows this:

```

Outline Coverage Console
<terminated> FP_Growth [Java Application] C:\Program Files\Java\jre-9.0.1\bin\javaw.exe (Apr 17, 2018, 10:13:31 PM)
Is Database Transactional? (Y/N)
y
Pattern for node: chicken
nails-->hammer-->peppermint-->gum-->bread-->cigarettes-->
Pattern for node: bread
cigarettes-->
Pattern for node: onions
peppermint-->fish-->gum-->beer-->bread-->cigarettes-->diapers-->milk-->gum-->bread-->cigarettes-->
Pattern for node: hammer
peppermint-->gum-->bread-->cigarettes-->diapers-->chocolate-->bread-->cigarettes-->
Pattern for node: nails
hammer-->peppermint-->gum-->bread-->cigarettes-->
Pattern for node: knives
fish-->milk-->chocolate-->beer-->cigarettes-->
Pattern for node: milk
gum-->bread-->cigarettes-->chocolate-->beer-->cigarettes-->
Pattern for node: tomato
cheese-->hammer-->diapers-->chocolate-->bread-->cigarettes-->
Pattern for node: juice
onions-->diapers-->milk-->gum-->bread-->cigarettes-->
Pattern for node: cheese
hammer-->diapers-->chocolate-->bread-->cigarettes-->
Pattern for node: peppermint
fish-->gum-->beer-->bread-->cigarettes-->gum-->bread-->cigarettes-->
Pattern for node: diapers
chocolate-->bread-->cigarettes-->milk-->gum-->bread-->cigarettes-->
Pattern for node: fish
gum-->beer-->bread-->cigarettes-->milk-->chocolate-->beer-->cigarettes-->
Pattern for node: cigarettes
bread-->cigarettes-->beer-->cigarettes-->

Pattern for node: gum
beer-->bread-->cigarettes-->bread-->cigarettes-->
Pattern for node: beer
bread-->cigarettes-->cigarettes-->
Pattern for node: chocolate
bread-->cigarettes-->beer-->cigarettes-->

```

Figure 6: Frequent patterns obtained with minimum support 0

The following figure displays the sorted database based on frequency which is used to extract the frequent patterns:

```

Outline Coverage Console
<terminated> FP_Growth [Java Application] C:\Program Files\Java\jre-9.0.1\bin\javaw.exe (Apr 17, 2018, 10:11:26 PM)
Is Database Transactional? (Y/N)
y
Frequency Based Sorted Database

cigarettes bread beer gum fish peppermint onions
cigarettes bread gum peppermint hammer nails chicken
cigarettes bread chocolate diapers hammer cheese tomato
cigarettes bread gum milk diapers onions juice
cigarettes beer chocolate milk fish knives

```

Figure 7: : Sorted database based on frequency with minimum support 0

Taking an example of the chicken node, which can be seen from the figure, frequent pattern for chicken is displayed as nails->hammer->peppermint->gum->bread->cigarettes which is obtained from the frequency sorted database by listing all items before chicken in a bottom-up approach. On checking our initial transaction dataset, we find that this is correct and chicken does occur with the pattern obtained by our implementation.

The following figure displays the frequent patterns for the same dataset when minimum support is considered as 1:

```

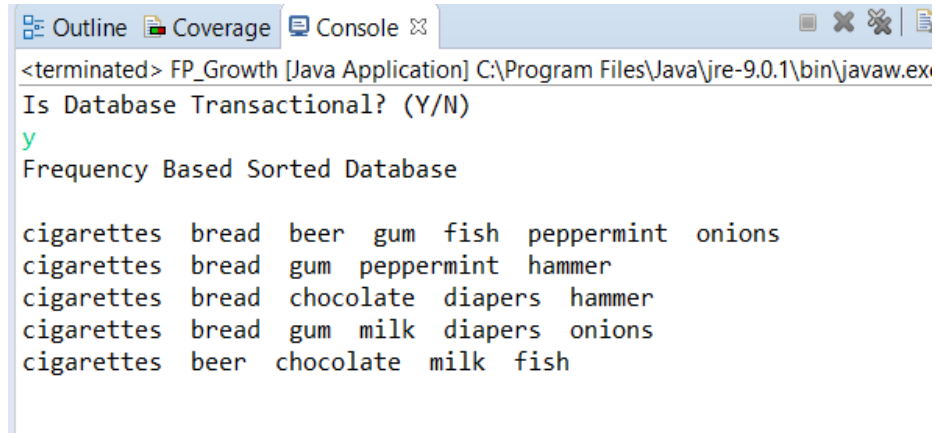
Outline Coverage Console
<terminated> FP_Growth [Java Application] C:\Program Files\Java\jre-9.0.1\bin\javaw.exe (Apr 17, 2018, 10:16:24 PM)
Is Database Transactional? (Y/N)
y
Pattern for node: bread
cigarettes-->
Pattern for node: onions
peppermint-->fish-->gum-->beer-->bread-->cigarettes-->diapers-->milk-->gum-->bread-->cigarettes-->
Pattern for node: hammer
peppermint-->gum-->bread-->cigarettes-->diapers-->chocolate-->bread-->cigarettes-->
Pattern for node: peppermint
fish-->gum-->beer-->bread-->cigarettes-->gum-->bread-->cigarettes-->
Pattern for node: diapers
chocolate-->bread-->cigarettes-->milk-->gum-->bread-->cigarettes-->
Pattern for node: fish
gum-->beer-->bread-->cigarettes-->milk-->chocolate-->beer-->cigarettes-->
Pattern for node: milk
gum-->bread-->cigarettes-->chocolate-->beer-->cigarettes-->
Pattern for node: cigarettes

Pattern for node: gum
beer-->bread-->cigarettes-->bread-->cigarettes-->
Pattern for node: beer
bread-->cigarettes-->cigarettes-->
Pattern for node: chocolate
bread-->cigarettes-->beer-->cigarettes-->

```

Figure 8: Frequent patterns obtained with minimum support 1

And the following figure shows the sorted database based on frequency which is used to extract the frequent patterns:



```

<terminated> FP_Growth [Java Application] C:\Program Files\Java\jre-9.0.1\bin\javaw.exe
Is Database Transactional? (Y/N)
y
Frequency Based Sorted Database

cigarettes bread beer gum fish peppermint onions
cigarettes bread gum peppermint hammer
cigarettes bread chocolate diapers hammer
cigarettes bread gum milk diapers onions
cigarettes beer chocolate milk fish

```

Figure 9: Sorted database based on frequency with minimum support 1

As can be seen in the above figures, the chicken node has now been removed from the frequent pattern set. This is because chicken item occurs only once in the entire transactional dataset and by setting minimum support to 1 filters out its occurrence. Now upon examining the frequent pattern set obtained for the fish node, we find frequent patterns to be with gum->beer->bread->cigarettes and milk->chocolate->beer->cigarettes, which upon checking the initial dataset, is also true.

Hence, upon increasing the minimum support we observe that frequent patterns greater than the minimum support is displayed. In our example, when we increased the minimum support to 1, frequent patterns for items which occurred only once was filtered from our output, while correctly displaying the remaining results.

Thus, we observe that the output results can be validated with the initial given dataset and we can conclude that FP-Growth algorithm has run successfully.

## VII. Experimental results:

Fully functional code for FP-Growth was written in environment for Java provided by Eclipse. Lenovo Yoga 710 with 8GB RAM and 2.70GHz processor speed was used to run all the datasets mentioned below. Primarily we tested accuracy of our code on the

small dataset that has been described previously. Furthermore, to check accuracy and performance we have tested our algorithm of following 4 datasets.

1. Turkiye Student: No. of Attributes: 33, No. of Instances: 5820.
2. Wiki4HE: No. of Attributes: 53, No. of Instances: 951.
3. Dermatology: No. of Attributes: 33, No. of Instances: 336.
4. Zoo: No. of Attributes: 17, No. of Instances: 101.

**a. Frequency based tree traversal**

Up to our understanding gained through implementing FP-Growth algorithm, time to execute code and memory requirement for the process are the main factors that needs to be considered. Hence following our instincts, we have tested FP-Growth for two different minimum support values. Also, we have tested outcomes based on way we access each node while extracting the pattern. For that purpose, we will discuss 3 experimental cases:

1. Accessing each node in header table which is not sorted
2. Accessing each node in header table which is sorted in Ascending order
3. Accessing each node in header table which is sorted in Descending order

Below observations are recorded for minimum support threshold of 10.

<i>Database</i>	<i>Unsorted header table (in ms)</i>	<i>Header table in Descending order (in ms)</i>	<i>Header table in Ascending order (in ms)</i>
Turkiye Student	1361	1435	1508
wiki4HE	777	771	739
Dermatology	203	204	267
zoo	79	120	91

*Table 4: Execution Time*



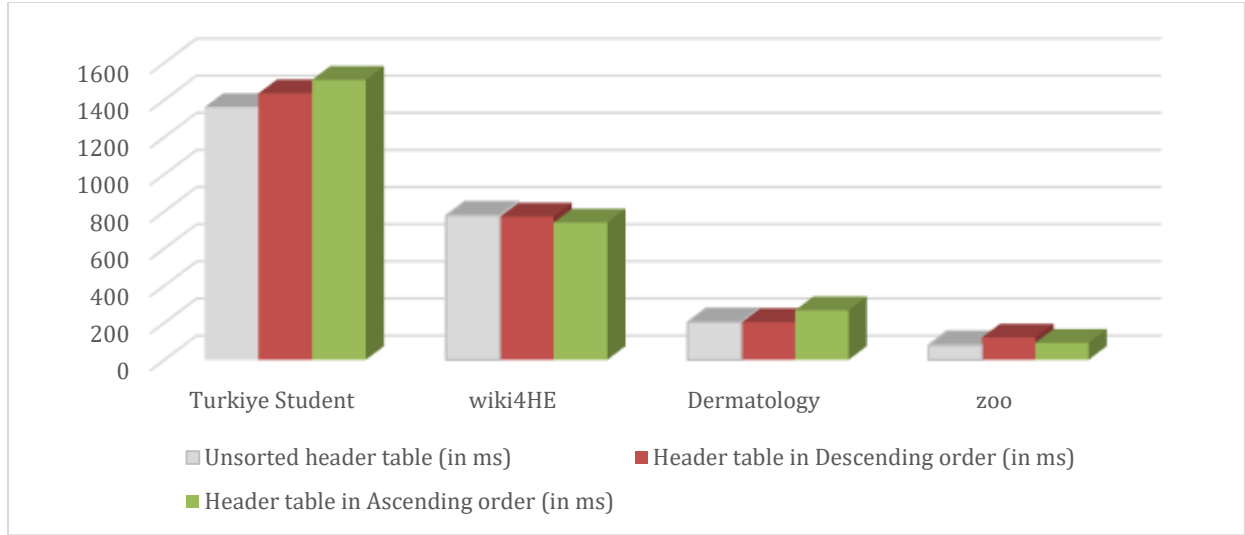


Figure 10: Execution Time graph

Database	For Unsorted header table (in MB)	For Header table in Descending order (in MB)	For Header table in Ascending order (in MB)
Türkiye Student	62.05	64.31	62.4
wiki4HE	52.76	53.2	52.7
Dermatology	4.53	4.55	4.53
zoo	2.09	2.09	2.09

Table 5: Memory Consumed

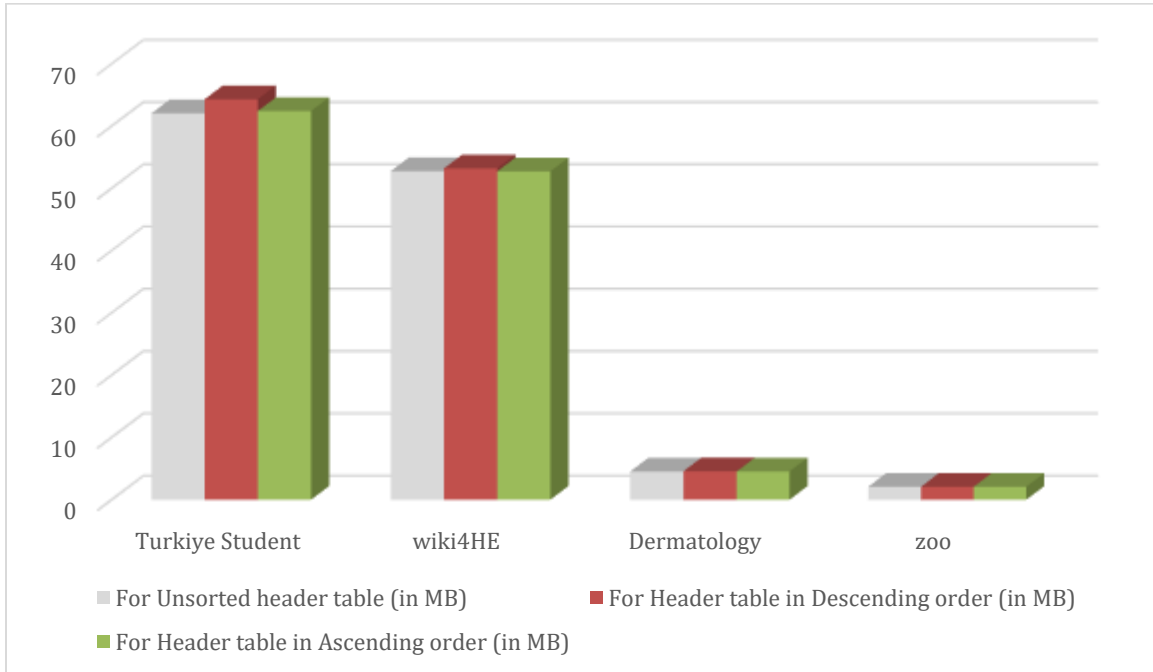


Figure 11: Memory Consumption graph

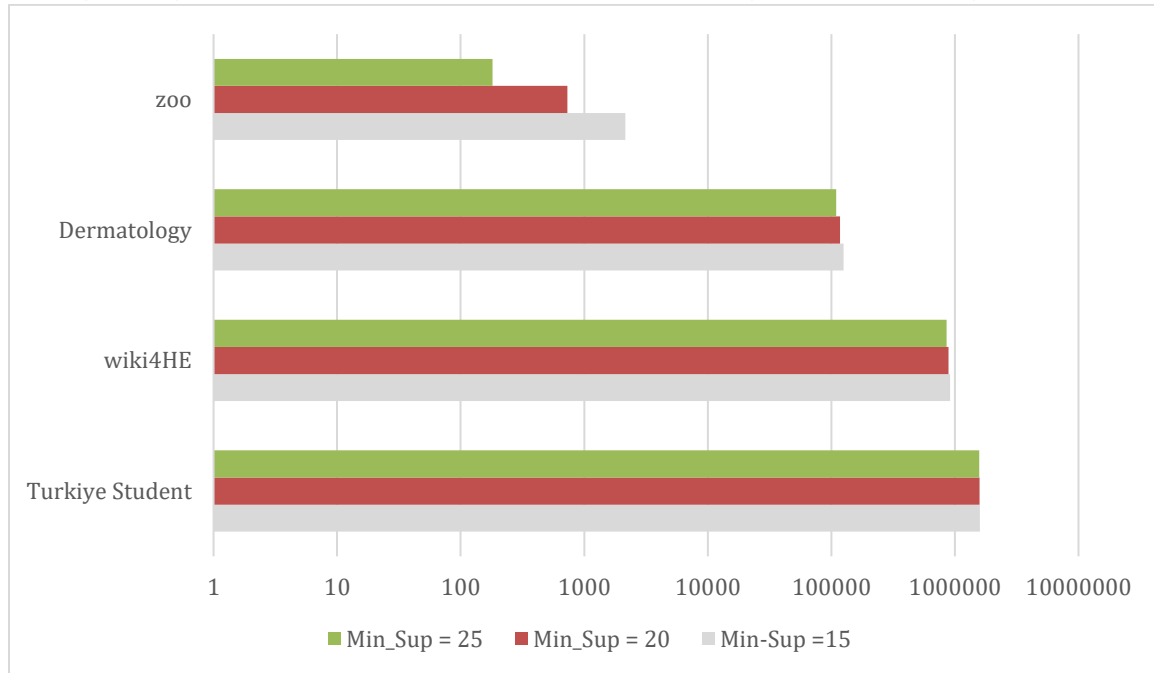
### b. Results based on Minimum Support

Variations in the pattern extracted due to change in minimum support count is another vital factor. As it is possible that, for higher min\_sup, FP-Growth might not work as efficient as it is for lower min\_sup. Here we have noted down total number of pattern generated and average number of pattern per node represented in following format *Total number of patterns/ Average number of patterns*.

<i>Dataset</i>	<i>Min_Sup =15</i>	<i>Min_Sup = 20</i>	<i>Min_Sup = 25</i>
Turkiye Student	1591898/9419	1582918/9366	1571837/9300
wiki4HE	913491/4248	886407/4181	856706/4158
Dermatology	125563/1173	117577/1175	109560/1153
zoo	2153/76	732/29	181/7

*Table 6: Minimum Support Variation*

Below graph gives brief about total number of pattern generated on logarithmic scale.



*Figure 12: Minimum Support comparison*

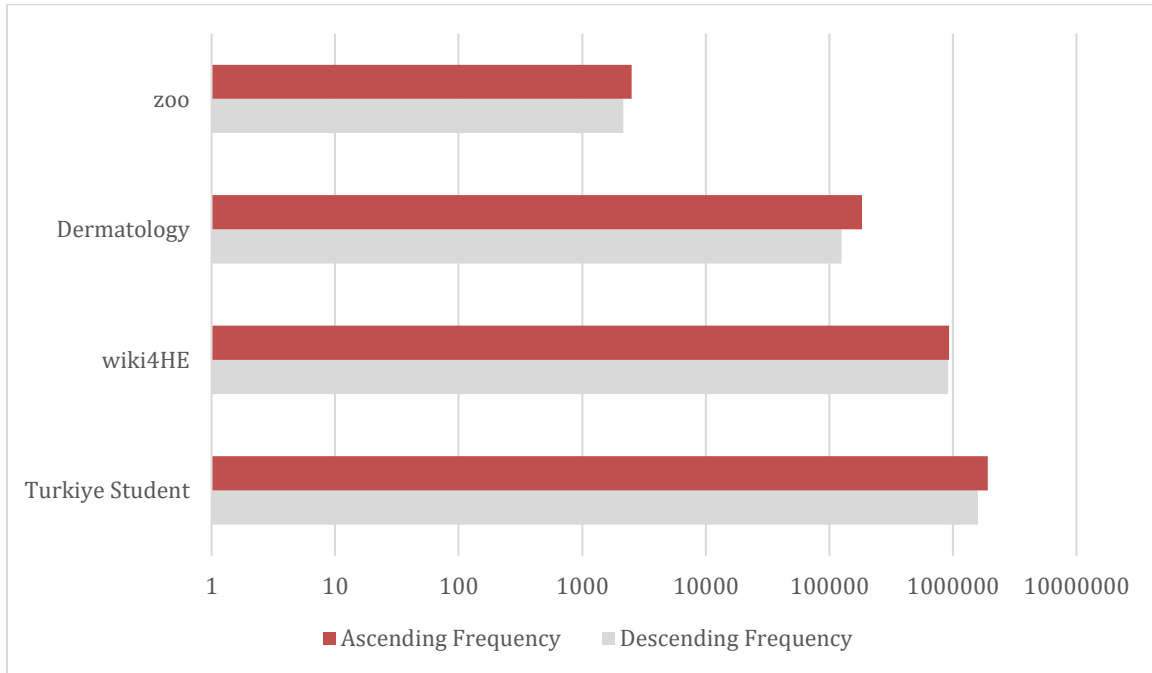
**c. Building Tree on Frequency ordered dataset(Ascending/Descending)**

In case of descending order, there will be lesser children at the higher level leading to more branches being merged early on and hence improving compactness on FP-Tree. Whereas for ascending order, lesser number of branches are merged and hence leading to wider FP-Tree. Following table supports the mentioned observations. All the entries are provided in following format: *Total number of patterns/ Average number of patterns*.

Minumum Support = 15.

<i>Database</i>	<i>Descending Frequency</i>	<i>Ascending Frequency</i>
Turkiye Student	1591898/9419	1918756/11353
wiki4HE	913491/4248	931603/4333
Dermatology	125563/1173	183805/1717
zoo	2153/76	2508/89

*Table 7: Frequency orderd dataset*



*Figure 13: Frequency ordered graph*

## **VIII. Extending FP-Growth for “Linear Programming-Based Optimization for Robust Data Modelling in a Distributed Sensing Platform”**

The paper “Linear Programming-Based Optimization for Robust Data Modelling in a Distributed Sensing Platform” provides an innovative approach to construct robust data models using samples acquired through a grid network of embedded sensing devices with limited resources. The paper has tested the scalability of the model by running simulations by using three network sizes of embedded nodes: 25, 64 and 100. We have considered the 25-node network model for our subsequent explanation on how we propose to implement the FP-Growth algorithm on the dataset.

It is known that all the 25 sensing nodes are connected to one of the nodes, designated as the target point (TP), via predefined path configurations. The TP receives the data packets, extracts temperature information, and builds the resultant thermal map. Loss, delay and error statistics are also extracted from the information received by the TP. The sensor values received from the 25 nodes will contain multiple errors as described in the paper – data loss errors, time delay errors, path-induced errors, and variable lumping errors. Thus, every value received by the TP will be an aggregate of the original sensor data as well as the mentioned errors. As these error values can be extracted from the packets received from each node, we decide not to discard these error values since they would give considerable knowledge regarding the frequent error flow patterns in the network as well as provide insight in improving efficiency.

Considering that each sensor node would have four attributes – the original sensor value, the loss associated with the sensor node, delays in receiving the packet, and errors associated with the values received – our initial dataset that would be created using all the sensor node values will contain  $4 \times 25$  attributes, i.e., each sensor node will have 4 attributes associated with it. The losses, delays and error statistics can be extracted using the equations described in the paper. Thus, our dataset will have 100 items per transaction.

We will consider the following implementation for our obtained dataset. Suppose, at a time  $t$ , the TP node collects data for all 25 sensors. The data required for our dataset will be extracted from these values and stored as a transaction. Now, the same process is repeated at time  $t+1$ ,  $t+2$ , ...,  $t+n$ , where  $n$  is the sampling period we are considering. This will give us  $n$  transactions to work with to perform FP-Growth algorithm. This  $n$  value can be increased or decreased to incorporate a larger/smaller time range of values to work with, thus, effectively increasing or decreasing the dataset size.

After we have obtained these  $n$  transaction (or tuples), we can begin with the execution of our implementation in order to determine frequently occurring patterns. The frequency count of different items (or sensor values) is first determined and stored in `HashMap<String, Integer>`. In order to reduce the large number of different values, we propose to round up the numerical values before storing them in the `HashMap`. Once we have obtained the frequency count, we can proceed by storing the dataset in the `ArrayList` of `Attribute` object. We will consider the utilization rates -  $\alpha$  for discarding sensed values,  $\lambda$  for different lumping levels,  $\beta$  for bandwidth values and `Path` for DCPs (data path configurations) – as Minimum support threshold for our obtained dataset. This will require modifications in the way the minimum support threshold is currently set in our implementation. Using these values, we can construct the FP-Tree and header table required for extracting frequent patterns. Following this, we can extract patterns for different attributes of the dataset.

The results of the sensor network can be calculated and stored for multiple sampling times  $n$  and frequent patterns can be extracted for the same. The results that we obtain can provide us with valuable insights about how the sensor network functions. For example, frequent patterns observed for original sensor values of node A and another node B reflect how the two sensors interact with each other. It can also be used to detect frequent delays in one node due to any attribute of another node. If such patterns are found frequently and start repeating at regular intervals, then it can be safely assumed that the attributes of the dataset are correlated. Errors at a node due to delays or losses at the same or different nodes can all be detected as a part of this frequent pattern extraction. Once such frequent patterns begin to appear, remedial action can be taken to reduce such occurrences. For example, if error in a sensor reading is repeatedly being caused by similar delays or losses in another sensor reading, then steps can be taken to reduce delays or losses in the latter sensor node, so as to reduce errors in the former sensor node. The path losses among data sent from different sensor nodes can be tracked and frequently delayed paths can be modified in order to increase efficiency. The frequent patterns obtained can also be used for predicting how the sensor node values will arrive, in case the sensor readings takes a repeating pattern. Missing values in the sensor readings can also be approximated by looking at the rest of the readings and determining the closest frequent pattern the readings can be a part of. The frequent patterns obtained can also be stored inherently in the memory of the sensor node, in order to increase the speed of data processing.

Our design implementation suffers from a drawback in that we have not considered memory constraints. Our design assumes that memory provided will be sufficient to perform the FP-Growth algorithm while holding the FP-Tree in the memory.

## IX. **Conclusion:**

Structure of FP-Tree gives it some benefits for larger dataset with lower minimum support values. Benefits such as:

- Highly compact data structure of FP-Tree allows to represent huge dataset in considerably compact structure.
- Due to pattern growth model it avoids costly candidate generation process
- Access to each node in sequential pattern makes pattern extraction more efficient.

With lower minimum support values, FP-Growth algorithm becomes more effective compared to dataset with higher minimum support.

Also, access to dataset with descending ordered dataset makes FP-Tree structure more compact and hence efficient.

## X. **Bibliography**

- [1] A. Umbarkar, V. Subramanian, A. Doboli, "Linear Programming-based Optimization for Robust Data Modelling in a Distributed Sensing Platform", IEEE Transactions on CADICS.
- [2] Jiawei Han, Micheline Kamber, "Data Mining Concepts and Techniques", 2<sup>nd</sup> edition.
- [3] Dataset sources
  - <https://archive.ics.uci.edu/ml/datasets/wiki4HE>
  - <https://archive.ics.uci.edu/ml/datasets/Turkiye+Student+Evaluation>
  - <https://archive.ics.uci.edu/ml/datasets/Dermatology>
  - <https://archive.ics.uci.edu/ml/datasets/zoo>

## XI. Appendix

Please find the entire source code attached in the zip folder shared.

### FP-Growth.java

```
import java.util.*;

public class FP_Growth {

    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();
        long beforeUsedMem=Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();

        final String filePath = "D:\\SEM-II\\Learning Systems-ESE589-
Doboli\\Project 2\\zoo.csv";
        //Adult, restraunt, wine, zoo, self
        //tur, wiki, der
        final int minSup = 15;
        final int minSupPatt = 15;
        ArrayList<Result> pattern;
        Model model;
        HashMap<String, Integer> freq ;
        ArrayList<Attribute> dataset;

        if(!Util.ifTransactional()) {
            freq = Util.calFreq(filePath);
            //COUNT FREQUENCY OF EACH ELEMENT
            dataset = Util.readCSV(freq, filePath, minSup);
            // REARRANGE TUPLE IN SORTED ORDER
        }else {
            freq = TransactionalUtil.calFreq(filePath);
            dataset = TransactionalUtil.readCSV(freq, filePath, minSup);
        }

        FP_Tree constructFPTree = new FP_Tree(dataset);
        //CONSTRUCT FP_TREE
        model = constructFPTree.parseTree();
        //CONTAINS COMPLETE FP_TREE AND HEADER TABLE

        Algorithm fpGrowth = new Algorithm(model.root, model.header, freq);
        //FP GROWTH ALGORITHM
        pattern = fpGrowth.fpGrowth(minSupPatt);

        //Util.displayfreq(freq);
        //PRINT FREQUENCY OF EACH NODE
        //Util.display(dataset);
        //PRINT REARRANGED DATASET
```



```

        //Util.displayTree(model.root);
                                //PRINT FP-TREE
        //Util.displayPatt(pattern);
                                //PRINT PATTERN

        int totalPatterns = 0;
        int patternCount = 0;
        for (Result result : pattern) {
            totalPatterns += result.pattern.size();
            patternCount++;
        }

        if(patternCount != 0) {
            System.out.println("tatal patterns:\t" + totalPatterns);
            System.out.println("avg patterns:\t" +
totalPatterns/patternCount);
        }

        long afterUsedMem=Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();
        long endTime = System.currentTimeMillis();
        System.out.println("It took " + (afterUsedMem - beforeUsedMem) +
"Bytes");
                                //PRINT MEMORY CONSUMPTION
        System.out.println("It took " + (endTime - startTime) + "
milliseconds");
                                //PRINT TIME CONSUMPTION
    }
}

```

### Algorithm.java

```

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map.Entry;

public class Algorithm {
    HashMap<String, Links> header;
    Node root;
    HashMap<String, Integer> freq;
    List<Entry<String, Integer>> sorted;

    Algorithm(Node root, HashMap<String, Links> header, HashMap<String,
Integer> freq){
        this.header = header;
        this.root = root;
        this.freq = freq;
        //this.sorted = Util.orderElement(freq);
    }

    public ArrayList<Result> fpGrowth(int min_sup) {
        sorted = Util.orderElement(freq);
    }
}

```

```

        HashMap<String, List<AllPattern>> result = new HashMap<>();

//        for(String str : header.keySet()) {
//            for(int j = 0; j < sorted.size(); j++) {
//                for(int j = sorted.size()-1; j >= 0; j--) {
//                    String str = sorted.get(j).getKey();
//                    //sorted.get(j).getKey()

                    if(header.get(str) != null) {
                        List<Node> link1 = header.get(str).link;

                        AllPattern ap = new AllPattern();

                        List<AllPattern> sdf = new ArrayList<AllPattern>();
                        for(int i = 0; i < link1.size(); i++)
                        {
                            Node current = link1.get(i);
                            ap.cnt = current.count;

                            AllPattern asd = new AllPattern();
                            asd.cnt = current.count;
                            while(current.parent != null) {
                                if(current.parent == root) {
                                    break;
                                }else {

                                asd.pattern.add(current.parent.name);

                                current =
                                current.parent;

                                }
                            }
                            sdf.add(asd);
                        }
                        result.put(str, sdf);
                    }
                }
            }
        }
        return minSupPattern(result, min_sup);
    }

    public ArrayList<Result> minSupPattern(HashMap<String, List<AllPattern>>
result, int min_sup) {
        ArrayList<Result> result1 = new ArrayList<Result>();

        for(String node : result.keySet()) {
            HashMap<String, Integer> freq = new HashMap<String, Integer>();
            for(int i = 0; i < result.get(node).size(); i++) {

```

```

        for(String str : result.get(node).get(i).pattern) {
            if(!freq.containsKey(str)) {
                freq.put(str, 1);
            }else
                freq.put(str, freq.get(str)+1);
        }
    }
    Result patt = new Result();
    //substring(0, node.length() - 4);
    patt.name = node;

    for(int i = 0; i < result.get(node).size(); i++) {
        for(String str : result.get(node).get(i).pattern) {
            if(freq.get(str) >= min_sup) {
                //substring(0, str.length() - 4)
                patt.pattern.add(str);
            }
        }
    }
    result1.add(patt);
}
return result1;
}
}

```

### FP\_Tree.java

```

import java.util.*;

public class FP_Tree {
    Node root = new Node();
    ArrayList<Attribute> dataset;

    FP_Tree(ArrayList<Attribute> dataset){
        this.dataset = dataset;
    }

    //PROCESS EACH TUPLE FOR CONSTRUCTING TREE
    public Model parseTree() {
        HashMap<String, Links> header = new HashMap<String, Links>();
        for(int i = 0; i < dataset.size(); i++) {
            //System.out.println(dataset.get(i).nodes);
            insertNode(dataset.get(i).nodes, root, header);
        }
        //System.out.println("Done");
        return new Model(root, header);
    }

    //INSERT EACH NODE FROM A TUPLE INTO TREE

```

```

    public void insertNode(ArrayList<String> tuple, Node root, HashMap<String,
Links> header) {
        Node current = root;

        for(int i = 0; i < tuple.size(); i++) {
            Node newNode = new Node(tuple.get(i));
            newNode.parent = current;

            int index = checkChildPresent(newNode, current);

            if(index == -1) {
                current.children.add(newNode);
                headerTable(header, newNode);
                current = newNode;
            }else {
                if(index != -1) {
                    current = current.children.get(index);
                    current.count++;
                }
            }
        }
    }

    //CHECK IF ELEMENT IS PRESENT AS A CHILD OF CURRENT NODE
    public static int checkChildPresent(Node newNode, Node current) {
        for(int i = 0; i < current.children.size(); i++)
            if(current.children.get(i).name.equals(newNode.name))
                return i;

        return -1;
    }

    //ADD NODE TO HEADER TABLE
    public static void headerTable(HashMap<String, Links> header, Node node) {
        //HashMap<String, Links> header = new HashMap<String, Links>();

        if(header.containsKey(node.name))
            header.get(node.name).link.add(node);
        else
            header.put(node.name, new Links(node));
    }
}

```

### Node.java

```

import java.util.*;

public class Node {
    String name;
    int count;
    Node parent;
}

```

```

ArrayList<Node> children;
boolean endOfTuple;

Node(String name){
    this.name = name;
    this.count = 1;
    this.endOfTuple = false;
    this.parent = null;
    this.children = new ArrayList<>();
}

Node(){
    this.name = "root";
    this.count = 1;
    this.endOfTuple = false;
    this.parent = null;
    this.children = new ArrayList<>();
}
}

```