ESE - 589 LEARNING SYSTEMS FOR ENGINEERING SYSTEMS
Stony Brook University

PROJECT 1

IMPLEMENTATION OF FP-GROWTH ALGORITHM

Submitted to

Professor Alex Doboli
Department of Electrical and Computer Engineering

Project Members:

Kaushik G Kulkarni - 111486036
Manoj Kumar Gopala - 111679371

CONTENTS

ABSTRACT

Data mining is used to deal with the huge size of the data stored in the database to extract the desired information and knowledge. It has various techniques for the extraction of data; association rule mining is the most effective data mining technique among them. It discovers hidden or desired pattern from large amount of data. Among the existing techniques the frequent pattern growth (FP growth) algorithm is the most efficient algorithm in finding out the desired association rules. It scans the database only twice for the processing. In this context we have implemented the FP growth algorithm and have evaluated our model on four datasets from UCI machine learning repository.

INTRODUCTION

It may not even be an exaggeration to say that the tasks of frequent itemset mining and association rule induction started one of the popular research area of data mining. The major problem in frequent pattern mining is finding the relationship between the items in a dataset. The enormous research efforts devoted to these tasks have led to a variety of sophisticated and efficient algorithms to find the frequent item sets. Among the best known are Apriori, ECLAT etc.

Nevertheless, there is still much room for improvement. While ECLAT, which is the simplest among all, can be fairly slow on some data sets when compared to some algorithms. Apriori, on the other hand results in large candidate itemset taking too much memory and further, also involves repeated scans of the dataset, proving to be too costly. Hence, a simpler processing algorithm which still maintains efficiency is desirable.

In this project, we introduce the FP-Growth Algorithm for frequent itemset mining which combines the suffix-based exploration with a compressed representation of the database for more efficient routing. This algorithm solves the issues of the Apriori Algorithm since there is no need to generate large candidate dataset and also scan the database just twice and performance wise, is proved to be faster when compared to other algorithms.

PROBLEM STATEMENT

The problem of frequent pattern mining can be stated as, given a database D with transactions T_1, T_2, ….T_N, determine all patterns P that are present in at least a fraction s of the transactions, where s is referred to as the minimum support. We also need to determine a solution to this problem in a time, space, memory efficient manner.

RELATED WORK

Frequent pattern mining has become an important and obvious need in many real-world applications, for example, in market basket analysis, advertisement, medical field, monitoring of patients' routines etc. The aim of frequent pattern mining to recursively search for any occurring relationships or interesting patterns within a dataset, which include association rules, correlations, sequences, episodes, classifiers and many more, of which association rule mining is one of the most important problems. Association rule mining is method to find out association rules that satisfy the predefined minimum support and confidence, the two basic parameters, from a given database. Support of an association rule, defined as the ratio of records that contains X U Y to the total number of records in a database & Confidence is defined as ratio of the number of transactions that contain (X U Y) to the total number of records that contain X, where if the percentage exceeds the threshold of confidence an interesting association rule X => Y can be generated. Several algorithms have been implemented to not only solve these problems, but also reduce the cost of these operations. Some of them are Apriori algorithm, Rapid Association rule mining, Equivalent class Transformation algorithm. Let us discuss each one of them below.

Apriori Algorithm –

This algorithm was proposed by R. Agrawal and R. Srikant for mining frequent itemset for Boolean association rules, based on a concept that previous information of frequent itemset properties are used. It is an iterative approach, where to find (n+1) Itemset, n-Itemset are used. The first step of this algorithm involves finding the set of frequent itemset by scanning the database to have a count for each of them, and those items that have a count more than the minimum support are selected, indicating 1-itemset candidates. In the next step, the database is scanned again to generate 2-itemset candidates, consisting of two items, then again pruned to produce large 2-itemset using Apriori property which states that "*every sub (k-1) – Itemset of frequent k-Itemset must be frequent*". Further, 3-Itemset, 4-Itemset and so on are generated until no frequent Itemset can be found. Apriori algorithm is basic two-step join and prune process.

In order to improve the efficiency of the Apriori Algorithm, several variations of the algorithm are proposed. They include –

- Hash-based Itemset counting – This is based on a concept that if a k-Itemset has a hashing bucket count which is below a certain threshold, then it cannot be frequent.
- Transaction Reduction – This says that a transaction that does not contain any frequent k-itemset is useless in subsequent scans.
- Partitioning – Partitioning concept involves, if any itemset that is potentially frequent in a database, then it must be frequent in at least one of the partitions of the database.
- Sampling - This approach deals with searching frequent itemset instead of whole database, that is, picking a random sample of the given data and then performing search in that sample.
- Dynamic itemset counting – This process includes a principle that says new candidate itemset can be added only when all of their subsets are estimated to be frequent.

Even though, several techniques are presented to improve the Apriori Algorithm, it has 2 major limitations: one is the complex candidate itemset generation process which consumes large

memory and enormous execution time and the second issue is the excessive database scans for candidate generation.

Rapid Association Rule Mining (RARM) –

RARM is a tree-based algorithm that avoids the complex candidate generation process. The tree data structure used is known as *Support-Ordered Trie,* using which RARM accelerates the mining process even at low support count. A *TrieIT* is a set of nodes in a tree that consists of 2 values – Item Label & Item Support. The Support-Ordered *trie* is a sorted ordered *TrieIT* where in the nodes are sorted with respect to their support count with highest support item at the left most nodes and lowest support node at the rightmost position in the tree. To construct a support ordered tree, 1-itemset and 2-itemset are extracted from each transaction, then sorted, and further, Depth First Search is carried out to traverse the tree to generate 1-itemset and 2-itemset and as these are generated, large itemset can be generated from these two itemset using the Apriori algorithm, that will reduce the execution time of candidate generation process.

Even though RARM reduces the complex candidate generation, it has several disadvantages as well. They are –

- RARM is difficult to use in an interactive environment because if the user support threshold is changed, the whole process will have to repeat again.
- It is not applicable for incremental mining, as dataset size is continuously increasing with addition of new transactions, the whole process must be repeated again and again.

Equivalence Class Transformation Algorithm (ECLAT)

ECLAT algorithm uses a vertical database (Items, TransactionID) instead of horizontal data format (TransactionID, Items) as in the cases of Apriori and Rapid Association Rule Mining. This algorithm with set intersection property uses depth First Search or breadth first search for traversal. All the frequent itemset are computed with the intersection of TransactionID & List. For the first scan of Database, a TransactionID list is maintained for each single item, k+1 itemset are generated using the Apriori property, process is continued until no frequent itemset can be found. The general idea of the ECLAT algorithm can be shown in these 5 steps.

- Get the TransactionID – List for every item.
- TransactionID – List of an item {a} is exactly the list of transactions that consists of {a}.
- Intersect the TransactionID – List of {a} with the TransactionID – List of all other items, resulting in TransactionID – List of {a,b}, {a,c} , {a,d} … = {a} – conditional database (if {a} is removed)
- Repeat from first step on {a} – conditional database.
- Repeat for all other items in the itemset.

The major advantage of ECLAT algorithm is that to count the support of k+1 large itemset, there is no need to scan the database because support count information for every item can be obtained from k-itemset. Further, ECLAT also avoids the overhead of generating all the subsets of a transaction and checking them against the candidate hash tree during support counting. One of the

limitation of ECLAT algorithm is that the intermediate TransactionID - List may become too large or the system memory.

Split & Merge Algorithm

The SaM algorithm for data mining is a simplification or extension of the recursive elimination algorithm, where the recursive elimination represents a database by storing one transaction list for each item, the SaM algorithm employs only a single transaction list stored as an array. Further, the array is processed with a split and merge scheme, which computes a conditional database, process it recursively and finally eliminate the split item from the original database. The algorithm processes a given transaction by these 3 simple steps. They are –

- The original database and the count/ item frequencies are loaded.
- Employ Split and merge mechanism to sort the order of transactions and also remove the less frequent items
- Create a new reduced transaction database with the sorted items with respect to their frequency.

The core advantage of the algorithm is its extremely simple data structure and processing scheme, which not only makes it easy to implement, but also easy to execute on external storage, thus rendering it a highly useful method if the data to be mined cannot be loaded on to the main memory.

FP-GROWTH ALGORITHM

Frequent Pattern growth algorithm is an efficient and scalable technique for mining frequent patterns in the field of Data mining, employing a divide & conquer strategy and thus reducing the cost and complexity of frequent pattern mining when compared to other algorithms mentioned in the previous section. The main feature of this algorithm is that it doesn't require generation of candidate itemset for the process of discovering frequent itemset. The idea of the algorithm is that the database which provides the frequent itemset is first compressed, then the compressed database is divided into a set of conditional databases, each associated with a frequent set and then data mining is applied on each database. In order to compress the data, a special data structure called FP-Tree is implemented. The Algorithm is a basic two-step approach. They are –

1. Build a compact data structure called the FP-Tree.
2. Extract Frequent itemset directly from the FP-Tree.

Construction of FP-Tree – The FP-Tree is a compressed representation of the input. As the database is read, each transaction say 't' is mapped to a path in the FP-Tree. Since many different transactions can have several items in common, their path may overlap, with which it is possible to compress the structure.
The FP tree is generated in a simple way. First, A root node is created pointing to null. A transaction 't' is read from the database. The algorithm checks whether the prefix of 't' maps to a path in the FP-Tree. If it maps, then the support count of the corresponding nodes in the tree are incremented, else, new nodes are created with a support count of 1. Additionally, a FP-Tree uses pointers connecting between nodes that have the same items, thus creating a singly linked list. These pointers are used to access individual items faster.

This FP-Tree is used to extract frequent itemset directly, as each node in the tree contains the label of an item along with a counter that shows the number of transactions mapped onto the given path. The algorithm has to scan the database twice.

- Pass 1- To determine the support of each item. The infrequent items are discarded.
- Pass 2- This is done to construct the FP-Tree.

Extracting Frequent itemset – This involves a bottom up strategy where in each prefix subtree is processed recursively to extract the frequent itemset, whose solutions are then merged. The mining procedure starts form each frequent length-1 pattern, then construct its conditional pattern base and thereby construct its conditional FP-Tree by mining recursively. The pattern growth is achieved by concatenating the suffix pattern with the frequent patterns generated from a conditional FP-tree.

Let us consider an example. The minimum support of the items is given to be equal to 3. The table below represents a dataset of a list of items bought from a market in terms of letters.

Transaction ID	List of Items
1	f, a ,c ,d, g, i, m, p
2	a, b, c, f, l, m, o
3	b, f, h, j, o
4	b, c, k, s, p
5	a, f, c, e, l, p, m, n

Table 1: Sample Dataset

Eliminating all the items that do not satisfy minimum support must be discarded and the items must be ordered in the descending of their minimum support count.

The items that satisfy the minimum support count are as follows – f:4, c:4, a:3, b:3, m:3 & p:3
The above table is now as shown below.

Transaction ID	List of Items	Ordered Frequent Items
1	f, a ,c ,d, g, i, m, p	f, c, a, m, p
2	a, b, c, f, l, m, o	f, c, a, b, m
3	b, f, h, j, o	f, b
4	b, c, k, s, p	c, b, p
5	a, f, c, e, l, p, m, n	f, c, a, m, p

Table 2: Ordered Frequent Items

The FP-Tree is now constructed for every row of transaction is shown in the set of figures.

First Transaction

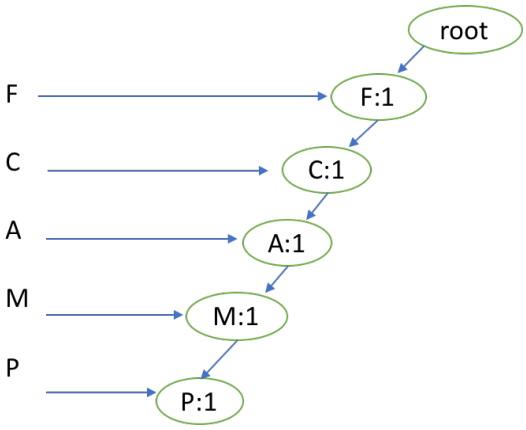


Figure 1: First transaction

Second Transaction

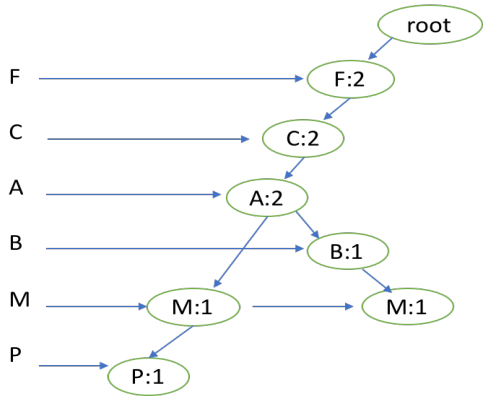


Figure 2: Second Transaction

Third Transaction

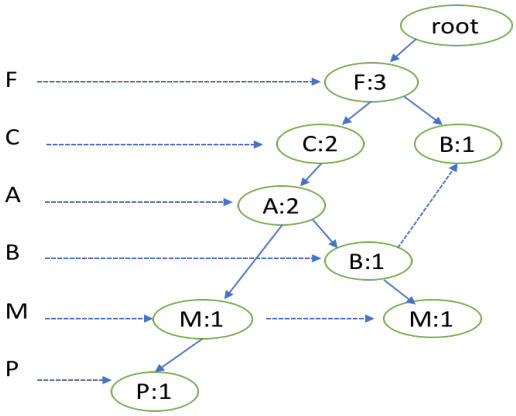


Figure 3: Third Transaction

Fourth Transaction

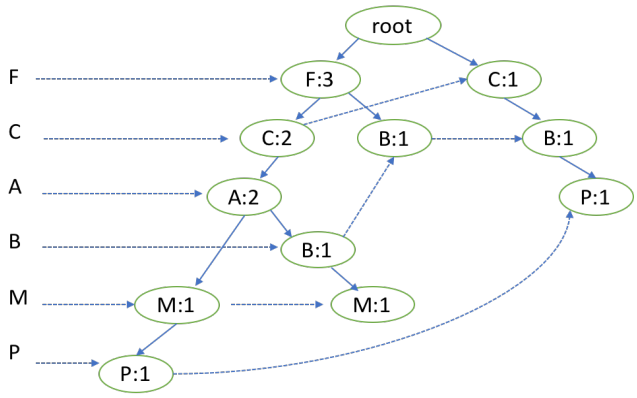


Figure 4: Fourth Transaction

Fifth Transaction

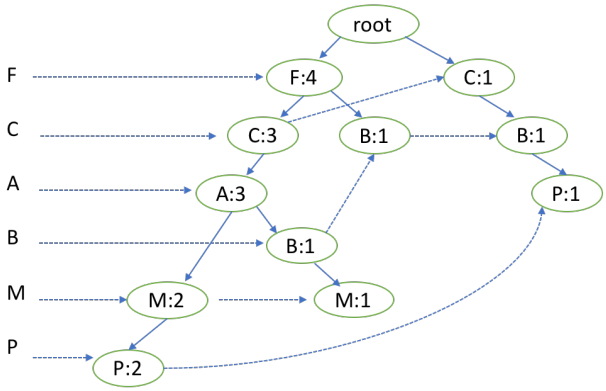


Figure 5: Final FP-Tree

Extracting Frequent Itemset – By following the procedure as mentioned in the algorithm, frequent item mining is done and the conditional pattern base and the subsequently a conditional FP tree is formed and is as shown in the table below.

Item	Conditional Pattern Base	Conditional FP tree
p	(f, c, a, m:2); (c, b:1)	(c:3)/p
m	(f, c, a:2); (f, c, a, b:1)	(f:3, c:3, a:3)/m
b	(f, c, a:1); (f:1); (c:1)	Empty
a	(f, c:3)	(f:3, c:3)/a
c	(f:3)	(f:3)/c
f	Empty	Empty

Table 3: Extracting Frequent itemset

Let us look at the pseudocode of the FP-growth algorithm, given below.

Algorithm Pseudocode:

```
FP-growth(FP-Tree on Frequent Items: FPT, min_sup: s, Current Itemset suffix: P)
Begin
   If FPT is a single path or empty
      For each combination C of nodes in path do
         Report all patterns C U P;
   Else
   For each item I in FPT do
   Begin
      Generate pattern Pi = {i} U P;
      Report pattern Pi as frequent;
      Use pointer-chasing to extract conditional
         Prefix paths after removing infrequent items;
      If(FPTi not equal to Ø) FP-growth(FPTi, Pi, s)
   End
End
```

Thus, we can say that the FP-growth algorithm transforms the problem of finding long frequent patterns into searching for shorter ones in much smaller conditional databases recursively, also reducing the cost of searching and also the overall frequent pattern mining operation. The algorithm also avoids costly, repeated database scans, and proves to be faster than the Apriori algorithm in the order of magnitude. In order achieve much more efficiency, let us discuss some variations of the FP-Growth algorithm. They include –

1. **Parallel FP-Growth** - For a large dataset, the support threshold needs to be set large enough, else the FP-tree would overflow the storage. For web mining tasks, the support threshold needs to be set very low to obtain long-tail itemset. This kind of setting might require unacceptable computational time. While in parallel FP growth, all the steps of FP growth can be parallelized leading to a decrease in computational time. PFP involves 5 steps: sharding, parallel counting, grouping items, parallel FP growth & Aggregating.
2. **Balanced Parallel FP-Growth** – This variation just adds a load balance feature to the Parallel FP growth. Due to this load balancing feature, parallelization and also performance is increased.

IMPLEMENTATION & EVALUATION

1) Dataset: Mushroom

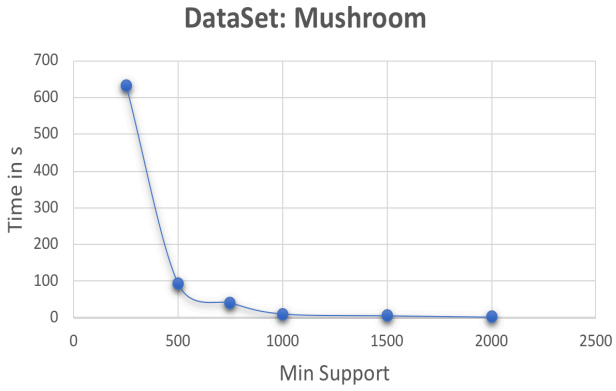
This data set includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family (pp. 500-525). Each species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended. This latter class was combined with the poisonous one. The Guide clearly states that there is no simple rule for determining the edibility of a mushroom; no rule like "leaflets three, let it be" for Poisonous Oak and Ivy.

Data Set Characteristics:	Multivariate	**Number of Instances**	8124
Attribute Characteristics:	Categorical	**Number of Attributes:**	22

The dataset was run on Mac book pro with 16 GB RAM and i7 Processor.

minsupport	Execution time	Memory used(Bytes)
2000	1.054 s	43896832
1500	4.787 s	58904576
1000	9.724 s	92839936
500	39.901 s	240996352
750	90.9166 s	500342784
250	632.70 s	3073363968

Table 4: Dataset Mushroom: Execution parameters



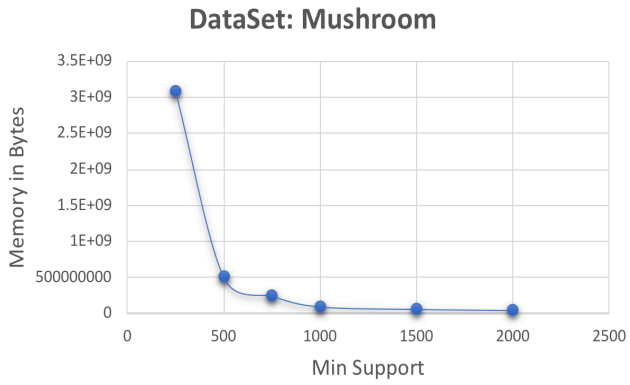


Figure 6: Time in S vs Min Support (Mushroom) *Figure 7: Memory used vs Min Support (Mushroom)*

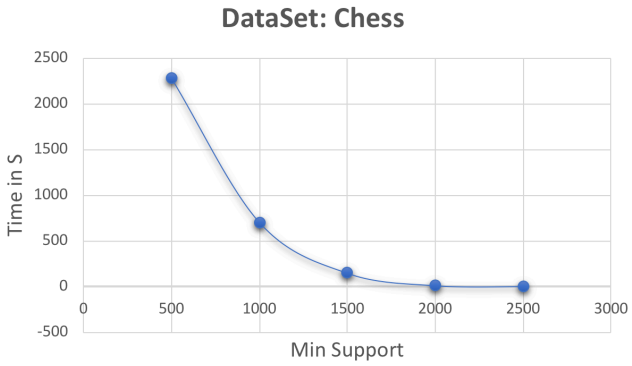
2) **Chess (King-Rook vs. King-Knight) Data Set**

Data Set Characteristics:	Multivariate	**Number of Instances**	3196
Attribute Characteristics:	Categorical	**Number of Attributes:**	36

The dataset was run on Mac book pro with 16 GB RAM and i7 Processor.

minsupport	Execution time in s	Memory used(Bytes)
2500	0.06175 s	23302144
2000	8.9142 s	61214720
1500	144.916 s	485756928
1000	700.32 s	893267977
500	2279.33 s	6442479616

Table 5: Dataset Chess : Execution parameters



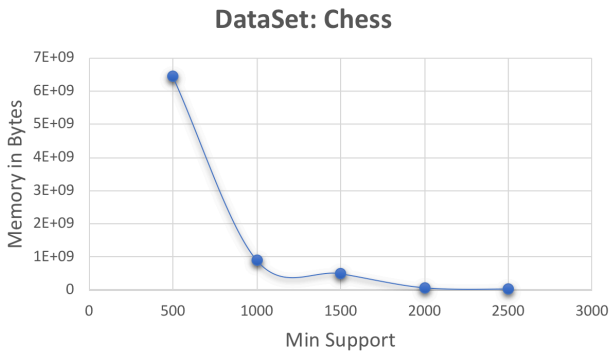


Figure 8: Time in S vs Min Support (Chess) *Figure 9: Memory used vs Min Support (Chess)*

3. Mammographic Mass Data Set

This data set can be used to predict the severity (benign or malignant)
of a mammographic mass lesion from BI-RADS attributes and the patient's age.
It contains a BI-RADS assessment, the patient's age and three BI-RADS attributes
together with the ground truth (the severity field) for 516 benign and
445 malignant masses that have been identified on full field digital mammograms
collected at the Institute of Radiology of the
University Erlangen-Nuremberg between 2003 and 2006.

Data Set Characteristics:	Multivariate	**Number of Instances**	961
Attribute Characteristics:	Categorical	**Number of Attributes:**	6

The dataset was run on Mac book pro with 16 GB RAM and i7 Processor.

minsupport	Execution time	Memory used(Bytes)
100	0.0460 s	5513856
50	0.0728 s	7689262
20	0.09779 s	11000128
10	0.168954 s	13774272
5	0.23497 s	15073280

Table 6: Dataset Mammographic Mass : Execution parameters

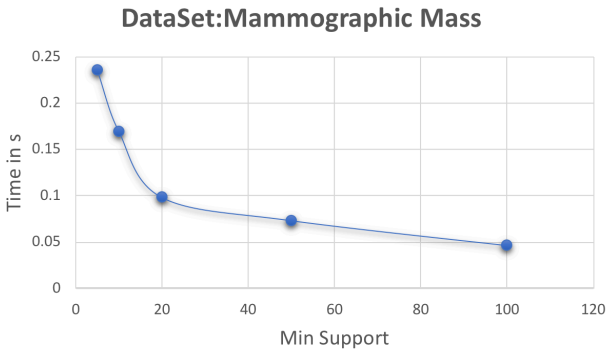


Figure 10: Time in S vs Min Support

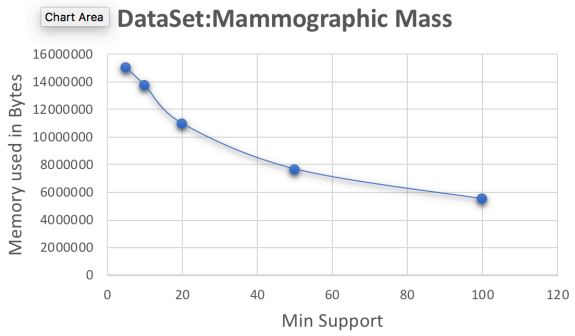


Figure 11: Memory used vs Min Support

4. Letter recognition

The objective is to identify each of a large number of black-and-white rectangular pixel displays as one of the 26 capital letters in the English alphabet. The character images were based on 20 different fonts and each letter within these 20 fonts was randomly distorted to produce a file of 20,000 unique stimuli. Each stimulus was converted into 16 primitive numerical attributes (statistical moments and edge counts) which were then scaled to fit into a range of integer values from 0 through 15. We typically train on the first 16000 items and then use the resulting model to predict the letter category for the remaining 4000. See the article cited above for more details.

Data Set Characteristics:	Multivariate	**Number of Instances**	20000
Attribute Characteristics:	Categorical	**Number of Attributes:**	16

The dataset was run on Mac book pro with 16 GB RAM and i7 Processor.

minsupport	Execution time in s	Memory used(Bytes)
5000	32.2030 s	272379904
3000	37.803 s	285437952
1000	56.99 s	292091648
500	61.59 s	294572032
200	67.03 s	296222720
100	71.27 s	301015040
50	82 s	302870528

Table 7: Dataset Mammographic Letter Recognition: Execution parameters

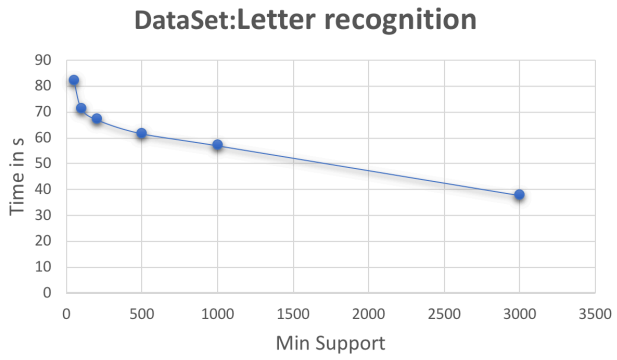


Figure 12: Time in S vs Min Support

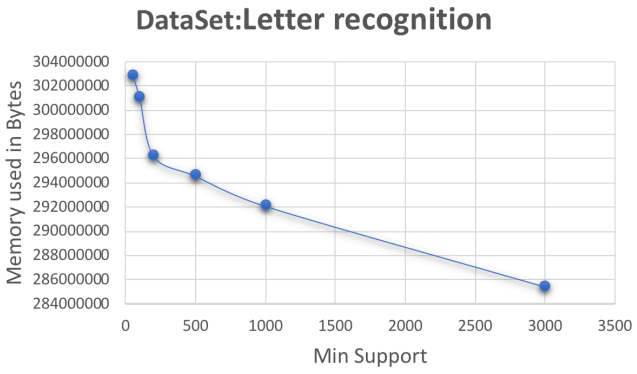


Figure 13: Memory used vs Min Support

As we can see from the experimental results when the Min Support is high the algorithm required less time and memory as most of the frequent patterns will be eliminated as data items won't meet the given Min Support. But as the Min Support is less then more data items meet the Min support condition and the memory and the required time increases exponentially.

Validation of the Algorithm

We validated our Algorithm in 2 steps.

First, we manually calculated the results on a small dataset the dataset which we have explained in the Algorithm section of this report. We used the same dataset as input to our implemented code and the results were accurate.

As we were not able to calculate manually for large datasets we used Apache Spark inbuilt FP-Growth library to validate our results. The results were similar and execution time and memory used was compared for the dataset Mushroom. Execution time was similar to our algorithm but the memory consumption of our algorithm was higher by 10% on average as compared to that of the apache inbuilt Library.

Improvement of the algorithm

Complexity Reduction

The issue with the FP growth algorithm is that it generates a huge number of conditional FP trees. This issue can you resolved by using D-trees. We scan the database one time to generate a D-tree from the database. The D-tree is basically the replica of the database under consideration. After scanning the database and getting the D-tree out of it, we then use the D-tree for further processing. We then scan the D-tree to count the no. of occurrences of each item in the database and record it as frequency of each item. Now here to calculate the frequency we scan the tree and not the database, since reading a transaction from the memory- resident tree structure is faster than scanning it from the disk Thus, using the D-tree and support count as an input we construct a New improved FP tree and a node table which contains only the required nodes and their frequencies.

Execution time Reduction

The time required for generating frequent patterns plays an important role in mining association rules, especially when there exist a large number of patterns and/or long patterns. We can reduce this with the help of multithread FP-growth. Every path in the FP-tree keeps track of an itemset along with its support, and it's known that each starting node generate it's related itemset.

In the Figure-14, taking item 4 and 3 as an example it exists in three branches, below is a list of its related items along it's tree branch path.

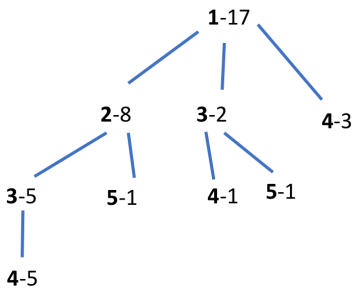


Figure 14: FP-Tree

1. Item 4
   - Branch1: {3,2,1}, supp 5
   - Branch2: {3, 1} , supp 1
   - Branch3: {1} , supp 3 2.

2 Item 3
   - Branch1: {2,1}, supp 5
   - Branch2: {1}, supp 2

So the candidate item sets of 4 are as follows:
- {3,2,1} Sup 5, {3,1} Sup 6, {1} Sup 9
and the candidate item sets of 3 are as follows:
{2,1} Supp 5, {1} Supp 7

The generated itemset are concatenated with its support using (:) are

- Itemset of 4 are as follows: (4):9, (4,1):9, (4,2):5, (4,3):6, (4,1,2):5, (4,1,3):6, (4,2,3):5, (4,1,2,3):5.
- Itemset of 3 are as follows: (3):7, (3,1):7,(3,2):5, (3,2,1):5.

As we see from the example each item generates its own frequent itemset, for example we can generate the itemset for item 4 and item 3 concurrently and then aggregate the results generated by both items in one file. So, we can create a thread to gather the frequent itemset of an item and after that combine the result of the finished thread in one single file.

DISCUSSION

The paper, ***"Linear Programming Based Optimization for Robust Data Modelling in a Distributed Sensing Platform"*** discusses a procedure to construct data models with high robustness by using samples that are acquired from a grid network of embedded sensors with limited resources like bandwidth & buffer memory. The data models are in the form of Ordinary Differential Equations in this paper. The local data models are constructed using lumping state variable, and further, the local models are collected centrally to produce a global data models. The basic purpose of lumping is to reduce communication traffic by combing some of the less important traffic data, thereby minimizing errors due to data loss & time delays. The errors describe the types of inaccuracies that occur during a data model construction using distributed embedded sensing networks. Further, using the error bounds, the lumping levels and the parameters of the networked sensing platform are computed, which further are used to set threshold values by local schemes at the embedded sensing nodes to decide modelling actions. Further, aggregation methods are implemented in a specific path manner, node to node, focused on reducing data traffic in order to lower the power and energy consumption, also avoiding transmission of redundant data.

For this paper, the FP-Growth algorithm can be implemented as follows:

1) A minimum support value must be predefined in this case for the sample embedded sensor data. As the first step of FP-Growth algorithm involves discarding the data less than minimum support, here, the first scan of the data must remove all the sensor nodes which are less than minimum support and place it in a temporary array. Further, sort the values of the array and find the mean/median of the array and store it in a variable. Then, add this value to all the transactions to the left most position in the descending order frequent item column as shown in table no. (--), that is, the least important position for every transaction. Using this table, construct the FP-Tree. Although, using a temporary array and sorting it increases the time and space complexity of the algorithm, we must make sure to use the best possible sorting algorithm.
2) After constructing the FP-tree, extract the mining pattern as usual and since the all unnecessary values that are lumped to avoid communication traffic, will rarely be extracted as frequent pattern.

SOFTWARE IMPLEMENTATION

The Algorithm was implemented in python. The FP-growth algorithm scans the dataset only twice. The basic approach to finding frequent itemset using the FP-growth algorithm is as follows:
1. Build the FP-tree.

2. Mine frequent item-sets from the FP-tree.

General Code Flow:

1) mapping from items to their supports

2) Load the passed-in transactions and count the support that individual items have

3) Remove infrequent items from the item support dictionary.

4) Build our FP-tree. Before any transactions can be added to the tree, they must be stripped of infrequent items and their surviving items must be sorted in decreasing order of frequency.

5) Build a conditional tree and recursively search for frequent itemset within it.

6) Search for frequent itemset, and yield the results

Code flow or conditional paths in tree

1) Import the nodes in the paths into the new tree. Only the counts of the leaf notes matter; the remaining counts will be reconstructed from the leaf counts.

2) Calculate the counts of the non-leaf nodes.

3) Eliminate the nodes for any items that are no longer frequent.

4) Finally, remove the nodes corresponding to the item for which this conditional tree was generated.

Data Structure used:

A trie, also called digital tree and sometimes radix tree or prefix tree (as they can be searched by prefixes), is a kind of search tree—an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings. Unlike a binary search tree, no node in the tree stores the key associated with that node; instead, its position in the tree defines the key with which it is associated. All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string. Values are not necessarily associated with every node. Rather, values tend only to be associated with leaves, and with some inner nodes that correspond to keys of interest. For the space-optimized presentation of prefix tree, see compact prefix tree.

Functions used:

Functions used	Description	Inputs	outputs
find_frequent_itemsets()	Find frequent item-sets in the given transactions using FP-growth. This function returns a generator instead of an eagerly-populated list of items.	transactions , Min Support	Item-sets
add()	Adds a transaction to the tree. If There is already a node in this tree for the current transaction item; reuse it else Create a new point and add it as a child of the point we're currently looking at.	Root node , transaction	Updated tree
update_route()	Add the given node to the route through all nodes for its item	Root Node, point	Updated path
Items()	Generate one 2-tuples for each item represented in the tree. The first element of the tuple is the item itself, and the second element is a generator that will yield the nodes in the tree that belong to the item.	Root Node	2-Tuples
Nodes()	Generates the sequence of nodes that contain the given item	Root Node , item	Sequence of nodes
prefix_paths()	Generates the prefix paths that end with the given item	Root Node , item	path
conditional_tree_from_paths()	Builds a conditional FP-tree from the given prefix paths	Paths , Min Support	Tree

Table 8: Functions

CONCLUSION

This paper presents the overall idea about Frequent Pattern Growth Algorithm for mining frequent patterns for a given database. The study of the FP-Growth algorithm performance also shows that it is efficient and scalable for mining both long and short frequent patterns, also certain variations of FP growth algorithm have proven to improve the performance and reduce the computation time. From the experimental results, the FP-tree algorithm has proven to be fast, reduced costs (both time and space) when compared to other frequent pattern mining algorithms like Apriori, RARM etc. Thus, FP-Growth algorithm provides a better performance in various types of data distributions.

BIBLIOGRAPHY

[1] Frequent Pattern Mining: Current Status & Future Directions, J. Han, H. Cheng, D. Xin, X.Yan, 2007, Data mining and knowledge discovery archive, Vol. 15 Issue 1, pp-55-86

[2] Fast algorithms for mining association rules. R. Agrawal and R. Srikant. In JB Bocca, M. Jarke and C. Zaniolo, editors, Proceedings of national conference on Very Large Data Bases, Morgan Kaufmann, 1994.

[3] Enhancing FP-Growth Performance Using Multi-threading based on Comparative Study Yousef K. Abu Samra, Ashraf Y. A. Maghar. Faculty of Information Technology Islamic University of Gaza, Palestine

[4] International Conference on Advanced Computing Technologies and Applications (ICACTA-2015)An optimized algorithm for association rule mining using FP tree. Meera Narvekara, Shafaque Fatma Sye

[5] Linear Programming – Based Optimization for Robust Data Modelling in a distributed sensing platform. Anurag Umbarkar, Varun Subramanian, Alex Doboli, IEEE Transactions on Computer-Aided Design of Integrated Circuits & systems, Vol. 33, No.10. Oct 2014.

[6] Frequent Pattern Mining, Ed. Charu Aggarwal and Jiawei Han, Springer, 2014.

[7] Frequent Pattern Mining Algorithms for finding associated frequent patterns for data streams: A survey. Shamila Nasreen, Awais Khan, Khurram Shehzad, Usman Naem, Mustansar Ali, 5th international conference on emerging ubiquitous systems and pervasive networks.

[8] Association Rule Mining: A survey, Qiankun Zhao, Sourav S. Bhowmick, National University of Singapore.

[9] C.L Blake and C.J Merz. UCI Repository of Machine Learning. Dept of Information and computer sciences, University of California, Irvine, CA, 1998.