Modified FP-Growth: An Efficient Frequent Pattern Mining Approach from FP-Tree

Shafiul Alom Ahmed(✉) and Bhabesh Nath

Tezpur University, Napaam, Tezpur 784028, Assam, India
tezu.shafiul@gmail.com, bnath@tezu.ernet.in

Abstract. Prefix-tree based FP-growth algorithm is a two step process: *construction of frequent pattern tree (FP-tree)* and then *generates the frequent patterns* from the tree. After constructing the FP-tree, if we merely use the conditional FP-trees (CFP-tree) to generate the patterns of frequent items, we may encounter the problem of recursive CFP-tree construction and a huge number of redundant itemset generation. Which also leads to huge search space and massive memory requirement. In this paper, we have proposed a new data structure layout called Modified Conditional FP-tree (MCFP-tree). Moreover, we have proposed a new pattern growth algorithm called *Modified FP-Growth* (MFP-Growth), which uses both top-down and bottom-up approaches to efficiently generate the frequent patterns without recursively constructing the MCFP-tree. During mining phase only one MCFP-tree is maintained in main memory at any instance and immediately deleted or discarded from the memory after performing the mining. From the experimental analysis, it is noticed that the proposed MFP-Growth algorithm requires less memory to construct the MCFP-tree as compared to conditional FP-tree. Moreover, the execution of the MFP-Growth method is found significantly faster than the traditional FP-Growth as it does not generate redundant patterns.

Keywords: Association Rule (AR) · FP-growth · Frequent Pattern (FP) · FP-tree · Pattern Mining (PM) · Data Mining (DM) · Frequent Itemset (FI)

1 Introduction

Frequent pattern (FP or FI) mining is considered as an fundamental problem of DM. It has been extensively exercised in some major DM operations, such as ARM, sequential patterns, classification, max and closed FP and clustering and has applications in many areas such as market-basket analysis, bioinformatics and web mining etc. The problem of mining FIs was first discussed by Agrawal *et al.* in 1993 (Apriori Algorithm) [1]. But the major problem of Apriori algorithm is that it generates huge number of candidate itemsets and also uses multiple database scan. Later on the researchers have discovered many association rule mining techniques with candidate generation approach but most of the algorithms suffers from same problems. In 2000 Han *et al.* [6] proposed an prefix path tree based approach called FP-Growth. This method performs the rule mining in two steps. First it constructs a compressed tree data structure called **FP_tree** using only two database scans. Secondly, it recursively constructs conditional frequent pattern tree (CFP-tree), a special kind of projected data structure to generate the frequent patterns for each individual frequent item of the database. Though the FP-Growth algorithm has been considered as one of the best and fastest frequent pattern generation algorithms; still it has a major disadvantage. For each individual frequent item of FP-tree, FP-growth recursively constructs the conditional FP-trees. Which also leads to huge search space and massive memory requirement.

In this work, an efficient CFP-tree data structure called Modified Conditional FP-tree (MCFP-tree) has been introduced. Unlike conditional FP-tree, the MCFP-tree is constructed in the reverse order of the header table to improve the mining process. Moreover, a new tree-traversal algorithm have been proposed to perform the mining process faster. It is not required to recursively construct the MCFP-tree for a single frequent item of the header table hence improves the performance. We have compared our technique with FP-growth and the experimental result shows that the modified FP-growth outperforms FP-growth.

2 Related Work

FP-tree is a special prefix-tree data structure, used by **FP-Growth** [6] algorithm to efficiently store the dataset information. Though, the performance of FP-Growth is noteworthy with respect to other existing pattern mining approaches, but the algorithm has some crucial drawbacks also. FP-Growth takes a huge amount of time to recursively construct the conditional FP-trees, particularly when the dimensionality of the dataset is high and the minimum support threshold value is low. If the *min-supp* value decreases then performance of FP-Growth also demotes and at some point of time with very low minimum support, it becomes almost similar to Apriori. Therefore, in case of low support threshold, the recursive mining of the CFP-trees, decreases the pattern mining performance unexpectedly. Therefore in 1997 an effective method was proposed by Park et al. [11] called *nonordfp*. It uses an unordered array of pairs (*child node name, child node index*) to map the children, so that the traversing becomes easier, just reading an array sequentially. But, this method works better only if the number of entries for the mapped arrays are very small. Otherwise, since the array needs a sequential memory space it becomes infeasible for huge or dynamic datasets. Many variants of FP-Growth algorithm have been found in literature such as CFP-Growth [12], Improved FP-Growth [8], CT-PRO [13], COFI-tree [3], Inverted Matrix [2], FP-Growth* [4], H-mine [10], Opportunistic Projection [9] and many significant work can be found in [5,7] also. In this work, an efficient method called Modified FP-Growth has been proposed to enhance the frequent pattern mining.

3 Proposed Method: Modified FP-growth

The Modified FP-growth method for mining FPs can be described in three phases. The working principle of MFP-Growth is illustrated with the help of a small dataset D [Table 1] and suppose the support threshold be 20%.

- ***Phase 1. FP-tree Construction:*** The proposed MFP-growth utilizes the conventional FP-tree construction algorithm to construct the FP-tree. Initially, the dataset D is scanned once to fetch the support count of individual item. The frequent items are then added to the header table with respect to their frequency descending order excluding the infrequent items. Then MFP-Growth constructs FP-tree for dataset D as depicted in Fig. 1.

Table 1. Transactional dataset (D)

TID	ITEMS
1	{a, b}
2	{b, c, d}
3	{a, c, d, e}
4	{a, d, e}
5	{a, b, c}
6	{a, b, c, d}
7	{a}
8	{a, b, c}
9	{a, b, d}
10	{b, c, e}

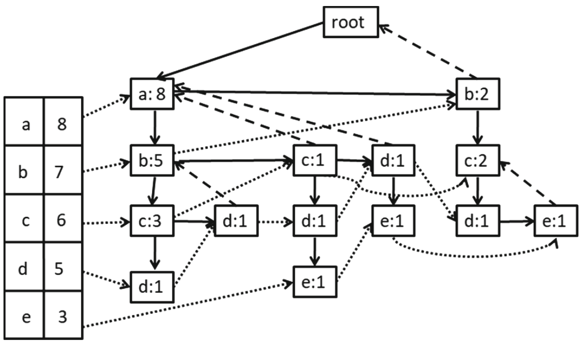


Fig. 1. FP-tree for D.

- ***Phase 2. Construction of MCFP-tree:*** Then the proposed method constructs an effective conditional FP-tree, called MCFP-tree to minimize the mining complexity. Like conditional FP-tree nodes, the MCFP-tree nodes also consists of: ItemLabel, NodeCount represents the number of transactions shared the path, ParentNodeLink, ChildNodeLink and SiblingNodeLink, additionally it contains two other informations namely RealativeItemCount and index. If the support of a node x in a path P_i is P_i(x), then all the node's support along the prefix path from node x to the root(excluding) is considered as P_i(x). The proposed method inserts the prefix path frequent items in the opposite order of conditional FP-tree. Therefore, the Modified CFP-Tree structure consists of a root node labelled with the frequent item for which the CFP-Tree is constructed and it can be described as the reverse CFP-Tree. The complete procedure of MCFP-tree construction illustrated in Algorithm 1.

Algorithm 1. Procedure: Modified_CFP_Tree-Construction(FP-Tree, X)

```
input: Minimum support count (minsup), FP-Tree and the item (X) .
output: MCFP-tree of the item X .
 1: Derive all the prefix paths $P_i$ for item $X$ (excluding) from the FP-tree and find the
    frequency counts of each individual items along the prefix paths of X.
 2: Discard the infrequent items and create a HeaderTable containing the frequent items.
 3: Define the root of the MCFP-tree: root($X$) and set the same item node link.
 4: for each prefix path $P_i$ of item $X$ do
 5:    tempRoot = root;
 6:    for each item $I_k$ in $P_i$ do
 7:       if $I_k$ is frequent and present in the $j^{th}$ position of the header table then
 8:          Call tempRoot = insert-MCFP-tree ( $I_k$, $P_i$(x), tempRoot,$j$ );
 9:       end if
10:    end for
11: end for
12: Procedure: insert-MCFP-tree( $I$, Count, tempRoot, index)
13: if tempRoot has a child_node with label $I$ then increment the frequency count of
    the child_node and HeaderTable[index].Count by adding the frequency count Count
    and set tempRoot = child_node;
14: else create a new child_node of tempRoot with label $I$, RelativeItemCount as 0 and
    set the frequency count of the child_node as Count and set HeaderTable[index].Count
    = Count and tempRoot = child_node ;
15: return(tempRoot); //Return to step 9.
```

- Let us consider, say we are to mine all the frequent patterns for the item '*e*' from the above FP-tree (Fig. 1).
  - First, the algorithm derives the set of all the prefix paths P_i(e) one by one in a bottom-up approach and derives the frequency count of each individual items along the prefix paths. Then the frequent items are inserted in to the header table and the infrequent items are excluded.
  - The algorithm constructs the MCFP-tree with the root node label '*e*' and sets the support of the root(e) as the total support count of item 'e' from the header table of FP-tree. The corresponding MCFP-tree for item '*e*' is shown in Fig. 2a. The corresponding conditional FP-tree for the item '*e*' is shown in Fig. 2b.

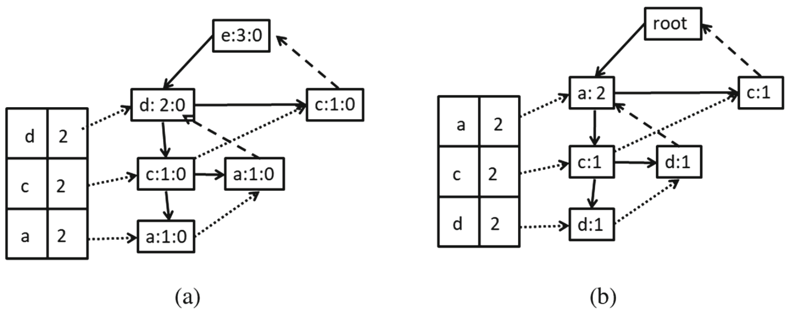


Fig. 2. MCFP-tree and conditional FP-tree for the item 'e'

Algorithm 2: Modified_FP-Growth

```
input: Minimum support count (minsup), Modified_CFP-Tree.
output: Set of frequent patterns.
 1: if $MCFP-tree$ contains a single path $Z$ then
 2:     for each combination $\beta$ of the nodes in $Z$ do
 3:         generate pattern $\beta \cup \gamma$ with support = minimum support count of the item in $\beta$.
 4:     end for
 5:     Define a List[] data structure to store frequent item and its support count.
 6:     List[0]=HeaderTable[0] //stores root node item and its support count.
 7: else
 8:     for i = 2 to n; where n is the size of HeaderTable do
 9:         $temp$ = HeaderTable[i]→SameItemNodeLink;
10:         while temp!=NULL do
11:             for each node $P$ in the prefix path of temp (excluding root) do
12:                 Set $P$ →realCount += temp→Count.
13:             end for
14:             Set Set temp→Count = 0.
15:             temp = temp→SameItemLink;
16:         end while
17:         for j = 1 to i-1 do
18:             temp = HeadetTable[j]→SameItemNodeLink;
19:             List[1]=HeaderTable[j]
20:             while temp!=NULL do
21:                 for each node $P$ in the prefix path of temp do
22:                     Set HeaderTable[$P$ →index]→Count += $P$ →realCount
23:                     $P$ →realCount = 0.
24:                 end for
25:                 temp = temp→SameItemLink;
26:             end while
27:             l=1;
28:             for k=1 to i-1 do
29:                 if HeaderTable[k]→Count ≥ minsup then
30:                     List[l] = HeaderTable[k];
31:                 end if
32:                 HeaderTable[k]→.Count=0;
33:             end for
34:             for each combination $\alpha$ of the nodes in List[] do
35:                 generate pattern $\alpha \cup \gamma$ with support = minimum support count of the item in $\alpha$.
                    //$\alpha = \{List[0], List[1]\}$
36:             end for
37:         end for
38:     end for
39: end if
```

– ***Phase 3. Mining the MCFP-tree:*** After constructing the MCFP-Tree, the next phase is the extraction of frequent patterns from the MCFP-tree. In this section, we have introduced an enhanced FP-growth method called Modified_FP-Growth. FP-Growth algorithm uses bottom-up scanning to recursively reconstruct the conditional FP-trees to generate the FPs for a single FI of the header table. On the other hand Modified_FP-Growth constructs a single for individual item of the header table of FP-tree and employs a bottom-up tree-traversing method to efficiently generate the FIs from the MCFP-tree without recursively constructing the MCFP-trees. The algorithm proposed for mining the FPs from the MCFP-Tree is illustrated in the Algorithm 2.
The procedure for mining the FIs for item 'e' from the MCFP-tree (Fig. 2a) by the proposed method is depicted bellow.

For node 'd', relCount('e') $= 2 \geq$ *minsupp*. Therefore, pattern **{'e', 'd':2}** is generated from Fig. 3. For item 'c', relCount('d') $= 1 \leq$ *minsupp* and relCount('e') $= 2 \geq$ *minsupp* as shown in Fig. 4 and it generates pattern **{'e', 'c':2}**. Similarly, for item 'a', item 'c' is infrequent as shown in Fig. 5. Therefore, it generates **{'e', 'a':2}** and **{'e', 'd', 'a':2}**. After mining all the frequent patterns from a MCFP-tree, the tree is deleted from the main memory to construct the MCFP-tree for other items of the header table.

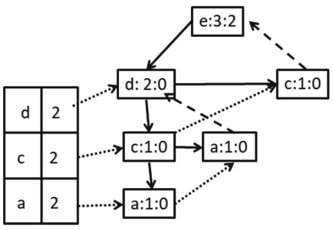


Fig. 3. For item 'd'

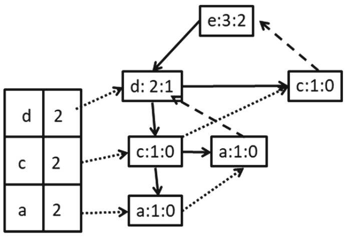


Fig. 4. For item 'c'

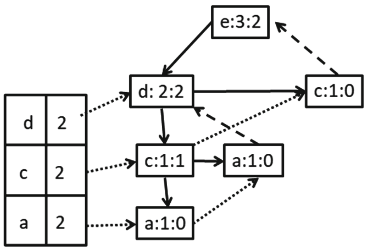


Fig. 5. For item 'a'

4 Performance Analysis

The performance of MCFP-Growth method is evaluated in terms of total time taken to execute the algorithm and memory requirement with respect to the FP-Growth technique. Experiments were performed on a machine with 3.2 GHz Intel i7 processor and 8 GB memory and 64-bit Linux operating system. The algorithms are implemented in C language and executed without running any background process. To analyse the performance of MCFP-Growth, we have used both real and synthetic datasets. To justify the effectiveness of MCFP-Growth algorithm, we have used dense datasets as well as sparse dataset also. The datasets mentioned in Table 2 are collected from UCI and FIMI repository.

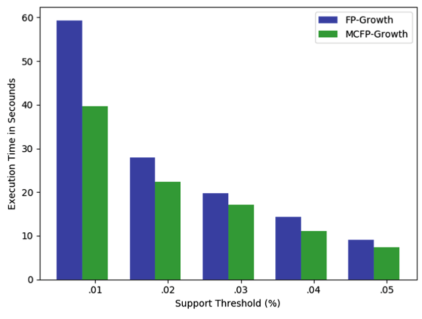
Table 2. Datasets used

Dataset	Category	Average length	Number of transaction	Number of item	Type
T40I10D100k	Synthetic	40	100,000	1000	Sparse
Connect-4	Real	43	67,557	129	Dense

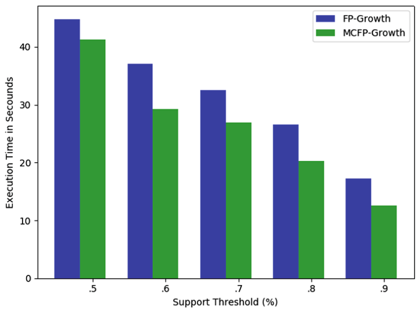
To compare the performance of both the approaches in terms of execution time, three set of experiments have been performed for each support threshold value. Then the execution time for each support threshold is considered as the average execution time of the three experiments for both the approaches. The execution time reported, is the total of MCFP-tree construction and time taken by the Modified_FP-Growth algorithm to mine the MCFP-tree. A comparison of total number of pattern generated and execution time for different datasets of MCFP-Growth and the FP-Growth have been illustrated in Table 3.

Table 3. Execution time

Dataset	Support (in %)	FP-Growth	MCFP-Growth	No. of patterns
		Mining time (secs)	Mining time (secs)	
T40I10D100K	.01	59.33	39.64	65236
	.02	27.90	24.30	2293
	.03	19.74	17.05	793
	.04	14.29	11.04	440
	.05	9.09	7.44	316
Connect-4	.5	44.80	41.26	88316229
	.6	37.12	29.24	21250671
	.7	32.51	26.05	4129839
	.8	26.58	20.04	533975
	.9	17.29	12.54	27127



(a) Dataset : T40I10D100K



(b) Dataset : Connect-4

Fig. 6. Execution time for different thresholds

As depicted in Fig. 6, FP-Growth algorithm invests more time as compared to the proposed method. For each item of the header table of FP-tree, the proposed method constructs a single MCFP-tree. But on the contrary, for each conditional FP-tree, FP-Growth algorithm recursively constructs conditional FP-trees for each frequent item of the header table. Therefore, recursive construction of trees lead to more number of tree node construction. If we consider the Dataset T40I10D100K, for .01% support threshold the header table size is 755. For the 755 header items the proposed method creates total 8736968 number of nodes to construct the 755 MCFP-tree, but for the same support threshold FP-Growth creates 8830100 number of tree nodes. Therefore, for each item of the header table, FP-Growth requires on average 123 more nodes as compared to the proposed method.

5 Conclusion

The major advantage of incorporating the relCount variable to the node structure of the proposed MCFP-tree is that unlike FP-Growth, it is not required to recursively construct the conditional FP-trees and hence reduces the tree construction time. We have also proposed and implemented the MCFP-Growth method for efficiently mining all the FIs in large datasets. The performance analysis shows that it efficiently computes the complete set of FIs and outperforms FP-Growth with respect to execution time and memory requirement.

References

1. Agrawal, R., Imielinski, T., Swami, A.: Tmining association rules between sets of items in large databases. In: ACM SIGMOD International Conference on Management of Data, vol. 22, pp. 207–216 (1993)
2. El-Hajj, M., Zaïane, O.R.: Inverted matrix: efficient discovery of frequent items in large datasets in the context of interactive mining. In: Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 109–118. ACM (2003)
3. El-Hajj, M., Zaïane, O.R.: Non-recursive generation of frequent K-itemsets from frequent pattern tree representations. In: Kambayashi, Y., Mohania, M., Wöß, W. (eds.) DaWaK 2003. LNCS, vol. 2737, pp. 371–380. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45228-7_37
4. Grahne, G., Zhu, J.: Efficiently using prefix-trees in mining frequent itemsets. In: FIMI, vol. 90 (2003)
5. Han, J., Cheng, H., Xin, D., Yan, X.: Frequent pattern mining: current status and future directions. Data Min. Knowl. Disc. **15**(1), 55–86 (2007)
6. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation: a frequent-pattern tree approach. In: Proceedings of ACMSIGMOD, Dallas, TX, pp. 1–12 (2000)
7. Han, J., Pei, J., Yin, Y., Mao, R.: Mining frequent patterns without candidate generation: a frequent-pattern tree approach. Data Min. Knowl. Disc. **8**(1), 53–87 (2004)
8. Lin, K.C., Liao, I.E., Chen, Z.S.: An improved frequent pattern growth method for mining association rules. Expert Syst. Appl. **38**(2011), 5154–5161 (2011)
9. Liu, J., Pan, Y., Wang, K., Han, J.: Mining frequent item sets by opportunistic projection. In: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 229–238. ACM (2002)
10. Pei, J., Han, J., Lu, H., Nishio, S., Tang, S., Yang, D.: H-mine: hyper-structure mining of frequent patterns in large databases. In: Proceedings 2001 IEEE International Conference on Data Mining, pp. 441–448. IEEE (2001)

11. Racz, B.: Nonordfp: an FP-growth variation without rebuilding the FP-tree. In: Proceedings of IEEE ICDM Workshop on Frequent Itemset Mining Implementations (2004)
12. Schlegel, B., Gemulla, R., Lehner, W.: Memory-efficient frequent-itemset mining. In: Proceedings of the 14th International Conference on Extending Database Technology, pp. 461–472. ACM (2011)
13. Sucahyo, Y.G., Gopalan, R.P.: CT-PRO: a bottom-up non recursive frequent itemset mining algorithm using compressed FP-tree data structure. In: FIMI, vol. 4, pp. 212–223 (2004)