International Journal of Applied Engineering Research ISSN 0973-4562 Volume 13, Number 7 (2018) pp. 5565-5574

2D Array Structure for Pattern Mining

Salah Alghyaline and Tariq Alwada'n

Department of Computer Science, The World Islamic Sciences and Education University, Amman, Jordan.

Abstract

Discovering frequent patterns is a crucial task during association rules mining. To our knowledge, frequent-pattern growth (FP-Growth) method is among the most efficient algorithms for pattern mining. However, many experimental results show that there is excessive memory requirement to build and traverse the conditional FP-trees in FP-Growth. In this paper we present a novel compact 2D array-based algorithm to reduce the memory consumption for FP-growth. Our algorithm efficiently uses the FP-tree data structure in combination with a new compact 2D array data structure to mine frequent itemsets. The advantage of the proposed method is that it does not require building many conditional FP-trees recursively, instead of that it just builds a single compact 2D array and then mines the frequent items through it. The proposed method generates the same frequent itemset compared with FP-growth for all the tested datasets. Our performance study shows that the proposed algorithm significantly reduces the memory consumption compared with FP-growth for many synthetic and real datasets, especially when support thresholds are low.

Keywords: Data mining, mining association rules, frequent itemsets, FP-growth

INTRODUCTION

Data mining is a process of applying different techniques like association rules discovery [1][2], sequential pattern mining [3][4] and clustering [5][6] to extract desired information and knowledge from a large amount of data in a database, for example in market basket analysis, we analyse the past transaction data to improve the quality of decision making. Association rules discovery or associative classification (AC) is a promising approach in data mining that is used to find useful patterns in the database. In data mining, classification and association rule discovery are similar. The purpose of the association rule discovery is finding the relationship between the items in the data set; whereas the goal of the classification is the prediction of class labels [7]. AC has two phases which are rule generation and building the classifier. After the success of Association rule discovery to build accurate classifier in the last few years, AC appeared as a new branch in data mining [8] [9].

In AC approach, the first step is the discovery of all frequent itemsets in the training data set; this step is computationally expensive and it is a challenging problem [9] [10]. Many algorithms were proposed for finding the frequent itemsets [11] [12] [13] [14] [15]. However, the complexity of CPU time and storage of such process is high; also, massive I/O operations are required. It could be enormous with a very low support threshold.

There are two methods for mining frequent item sets: the early approaches which generate candidate item sets like Apriori algorithm [11], the second without generating candidate itemsets like FP-growth algorithm [14]. Apriori algorithm is based in on the idea that all subsets of the frequent item set must be frequent ,it finds the frequent items in levels, each level uses the previous level to generate the new candidate item sets , it removes infrequent item sets from each level , so the number of candidate item sets will be decreased by increasing the number of levels .However Apriori has some shortcomings like it is hard to find the frequent item sets from all possible candidate item sets at each level specially if there is a large data set or low minimum support threshold ,also it scans the database repeatedly and makes matching with large set of candidates, Apriori performs well when the number of candidate itemsets is small [16]. After that many algorithms have been proposed to reduce the number of candidate itemsets, like controlling the number of candidates using direct hashing and pruning techniques [17] or using inverted hashing and pruning [18].

FP-growth algorithm was proposed by Han, it is faster than Apriori algorithm and faster than some recently reported new frequent pattern mining methods. FP-growth mines frequent items without generating candidate set, moreover, it has several advantages like it saves the costly database scan by building a highly compact FP-tree. FP-growth scans the database only two times, and it avoids candidates generation by applying specific pattern growth. [19] proposed a method based of FP-Growth to detect vulnerable lines in large power grid. [20] used FP-Growth to analysis and detect the occurrence of different crime patterns. [21] used FP-Growth to reduce the data being clustered by finding all possible semantic frequent item sets.

FPgrowth* approach [10] uses FP-tree data structure in conjunction with FP-array data structure to reduce the need of traversing FP-tree many times. Building each conditional FP-tree in the original FP-growth requires scanning the FP-trees two times: First, to find out the support of items in the conditional database. Second, for bulling the new conditional FP-tree according to the new order of these frequent items. However, using the FP-array technique by FPgrowth* drastically speeds up the FP-growth method and Only one scan for the FP-tree is needed for each recursive call during mining data from the conditional FP-trees, since it omits the first scan for conditional FP-trees. The constructed array is used to obtain the frequencies of the items that can be accessed in O(1) cost, additionally, building the FP-array for any conditional FP-tree is performed simultaneously while building the upper-level

conditional FP-tree. In summary, FPgrowth* works very well with the sparse and very large datasets and the FP-growth outperforms the FPgrowth* for dense datasets.

Ascending Frequency Ordered Prefix-Tree (AFOPT) [22] is another improvement in FP-growth and suggests that the key factor that influences the performance of generating frequent items in pattern growth approach is the total number of conditional FP-trees which are being built during the process of mining as well as the time and memory for traversing these trees. The order of the items in the conditional database and the way of traversing it significantly affect the overall performance. AFOPT presents a compact Prefix-tree data structure for building the conditional databases by rearranging the frequent items in ascending order. However, the dynamic ascending order will minimize the number of conditional databases compared with FP-growth that adapts arranging the frequent items in descending order, additionally, the AFOPT traverses the conditional databases in a top-down rather than bottom-up scanning. As a result, the cost of traversing the conditional databases will be reduced.

Next section gives more details about mining frequent patterns using FP-growth algorithm. However, many experimental results show that building a conditional FP-trees recursively during frequent patterns mining consumes most of the CPU time [10]. Each conditional FP-tree requires scanning the FP-tree two times. First time, for building the header table to find items frequencies. Second time, for building the conditional FP-tree according to the new order of the frequency after eliminating infrequent items. FP-growth builds many conditional FP-tree recursively until reaching single node; this consumes a lot of CPU time and space especially when dataset is huge and sparse.

This paper proposes a new algorithm based on the well-known algorithm FP-growth. The proposed algorithm avoids scanning the FP-tree many times, also it does not build conditional FP-trees recursively, which in turns, results in reducing the time and memory for scanning and building of the conditional FP-tree. Our algorithm builds FP-tree like FP-growth algorithm, after that it requires only one scan for the FP-tree to construct highly single compact 2D array for the whole mining process. Moreover, the data will be arranged in the 2D array in a way that facilitates searching. The proposed method utilizes the compressed data into the 2D array to generate all the frequent items with less number of calculations and memory space. However, the experimental results which have been applied in many synthetic and real data sets from FIMI repository website[1] show that the proposed method reduces the CPU time and memory usage compared with FP-growth with different percentages. The rest of paper is organized as follows. Section 2 describes the related work about mining frequent patterns. Our algorithm for mining frequent patterns is proposed in section 3. Section 4 shows the experimental results. Finally, we conclude this paper in section 5.

RELATED WORKS

The frequent pattern growth (FP-growth) is based on the FP-tree which is a compact representation of all frequent items in the database; however, mining the compact FP-tree is faster than mining the whole database compared with Apriori approach. FP-growth begins with scanning the database to find the occurrence of each item, after that it will sort the frequent items that exceed the minimum support threshold in descending order inside a header table according to their frequencies. Figure 1 (b) shows the header table for the given transactions Figure 1 (a).

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	{B, C}
8	{A, B, C}
9	{A, B, D}
10	{B, C, E}

(a) Database

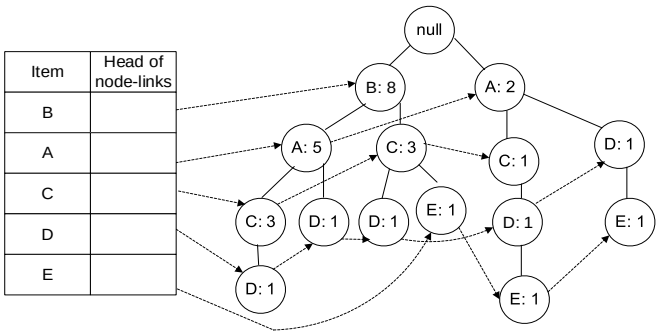
Item	Support
B	8
A	7
C	7
D	5
E	3

(b) Header table

Figure 1. Building the header table with *min_sup = 2*

The second scan results in building the FP-tree as shown in Figure 2 (b), if some transactions share a common prefix with the same order then the shared parts can be merged in the same path during building the FP-tree; however, the shared paths can be increased by sorting the transactions in descending order to facilitate the process of traversing the FP-tree items. The items that have the same label will be linked together with the linked list, and the mentioned header table will mark the beginning of the linked list for each item Figure 2 shows how the header table links all the nodes that have the same label in the FP-tree together.

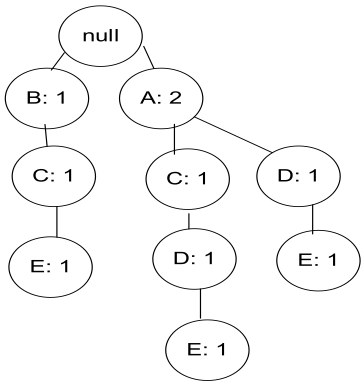
[1] http://fimi.ua.ac.be/data/



(a) Header Table (b) FP-Tree

Figure 2. An example of FP-tree with *min_sup*=2.

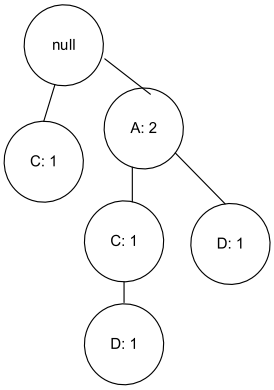
During the FP-growth, building the conditional FP-trees is a crucial task for mining the frequent items. At the beginning, the items in the header table will be visited starting from the bottom of the header table. Then, we will follow the path that is containing the target item such as a_i starting from a_i's head in the FP-tree header table. We determine the new counts of items along these paths, then the new counts are used to build a new header table as it shown in Figure 3 (b). Branches that contains item a_i will be visited one more time and the corresponding item sets in the tree will be inserted according to the new order of the items in the new header table after removing the infrequent items. Note that, we rebuild the conditional FP-tree for each item a_i, because the order of the items in the conditional FP-tree is not like order in the original FP-tree, thus, the items which do not satisfy the minimum support (*min_sup*) will be removed iteratively until we reach the root. Figure3 (c) shows conditional FP-tree which can be obtained by visiting branches along linked list of item *E* . This procedure is applied recursively for the set of paths from the *E*-conditional FP-tree, ending in *D, C* and *A*. However, building the conditional FP-tree for any item will stop when the new FP-tree includes only one single path, then all the subsets will be reduced in this path merged with the corresponding suffix. Table 1 summarized the conditional pattern bases and the conditional FP-trees.



(a) The set of paths ending in E.

Item	Support
A	2
C	2
D	2

(b) New Header table.



(c) Conditional FP-tree for suffix E.

Figure 3. Building conditional FP-tree.

Table 1. Mining patterns by creating Conditional pattern base

Item	Conditional pattern base	Conditional FP-tree
E	{(BC:1), (ACD:1), (AD:1)}	{(A:2), (C:2), (D:2)} | E
D	{(BAC:1), (BA:1), (BC:1), (AC:1), (A:1)}	{(A:4), (B:3), (C:3)} | D
C	{(BA:3), (B:3), (A:1)}	{(A:4), (B:6)} | C
A	{(B:5)}	{(B:5)} | A
B	∅	∅

Finding the frequent patterns through building conditional FP-trees consumes a lot of time and memory space, hence, the whole CPU time and memory space are used for traversing the FP-trees and building the conditional FP-trees. According to some experiments that have been conducted by [10] about 80% of the CPU time is used for traversing the FP-tree. Therefore, the following question will be emerged accordingly; can we reduce the traversing time and memory space of the FP-growth, which in turns, reduces the overall time and memory in mining patterns? Our experiment's results demonstrate the feasibility of using another technique for traversing the FP-tree instead of building the expensive conditional FP-trees. The proposed method added some data structures during building the FP-tree like arrays and array of pointers which makes traversing FP-tree more easier and eliminates the need for building the

conditional FP-trees. Also, the experimental results show that the proposed algorithm reduced the memory consumption and execution time for many data sets. The details of the proposed algorithm will be explained in the next section.

THE PROPOSED METHOD

There are two steps during mining data using FP-growth: building the FP-tree and mining the database through traversing the FP-tree and building the conditional FP-trees recursively. Our approach is sharing the first step with FP-growth and utilizing the overlapping between the different transactions, FP-tree is a compact representation of the original database which makes mining the FP-tree is more efficient than mining the database. However, our method builds a compact 2D array based on FP-tree; the proposed 2D array includes all the required information for mining the entire frequent items. Noted that, from memory side only one 2D array is allocated for the whole process and single scan for the FP-tree is needed, whereas in FP-growth many FP-trees are built and scanned recursively. This will improve the efficiency of mining the frequent patterns because most of the consumed time and memory in FP-growth are for traversing the FP-tree and building many conditional FP-trees. The following sections will describe in details the two main steps for mining frequent patterns using the proposed algorithm.

Constructing the FP-tree:

Building the FP-tree in our algorithm is like building FP-tree using FP-growth method. It starts by scanning the transactions in the given dataset *D* to count the support of each item. The frequent items which denoted as *L1* are the items with support exceeded the minimum support threshold. The items in *L1* will be rearranged in descending order to their supports. The root node for FP-tree is then created and set to "null", however, for each transaction such as *t* in *D*, if $t[l] \in CN(l\text{-}1)$, denote the child node as temporary *l*th node and increase the frequency of this child node by one, where *CN (l-1)* is the set of item-names for child nodes of the *(l-1) th* root. If $t[l] \notin CN(l\text{-}1)$, add a new child node for the *(l-1) th* root; denote this child node as temporary *lth* node with frequency *1* and set the item-name of the child node as *t[l]*.

The following figures explain the main steps for building the FP-tree. Figure 4 (a) represents the database being mined. Whereas the header table which includes the list of all frequent items with length"1" is shown in figure 4 (b). Figure 5 shows the header table which links all the nodes with the same label together, this header table will be used to traverse the FP-tree during frequent pattern generation step. Algorithm 1 summarizes the different steps that are used to build the FP-tree using the proposed method.

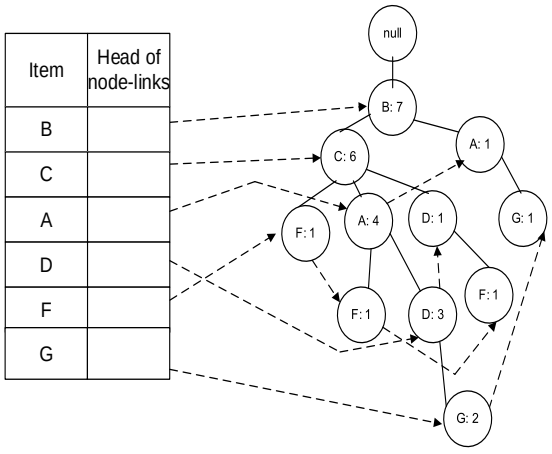
TID	Items
1	{B, C, A, D}
2	{B, A, G}
3	{B, C, D, F}
4	{B, C, A, D, G}
5	{B, C, F}
6	{B, C, A, D, G}
7	{B, C, A, F}

(a) Database

Item	Support
B	7
C	6
A	5
D	4
F	3
G	3

(b) Header table

Figure 4. Building the header table with *min_sup* = 2.



(a) Header table (b) FP-tree

Figure 5. An Example FP-tree (*min_sup*=2) using the proposed algorithm

Algorithm 1: FP-tree Construction

FP-tree Construction (*D, min-sup*)

Input: Transactional data set *D;*

Minimum support *min_sup*.

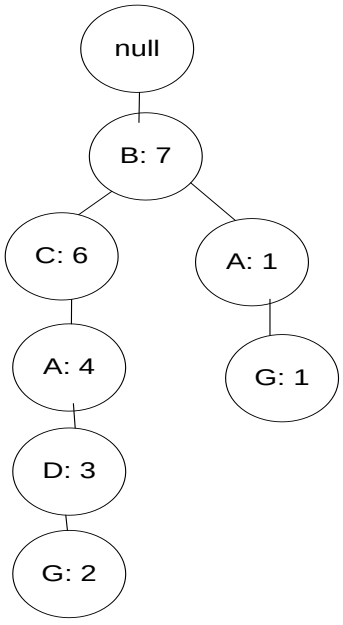
Output: FP-tree

1. Scan the transaction data set *D* to generate the set of frequent items *L1*.
2. Sort the items of each transaction in *D* according to the descending order of *L1*.
3. Create the root of FP-tree *T* and denote it as "null" (0^{th} root)
4. **For** each $t \in D$ and l=1 to $|t|$
   - 4.1. **If** $t[l] \in CN\ (l\text{-}1)$ denote the child node as temporary *l*th node and increase the frequency of this child node by one, where *CN (l-1)* is the set of item-names for child nodes of the *(l-1) th* root.
   - 4.2. **If** $t[l] \notin CN\ (l\text{-}1)$, add a new child node for the (*l-1) th*; denote this child node as temporary lth node with frequency *1* and set the item-name of the child node as *t[l]*. Update $CN\ (l\text{-}\ 1) = CN\ (l\text{-}1) \cup \{t[l]\}$.
5. **Output:** FP-tree

FREQUENT PATTERN GENERATION:

This section explains in details our approach for generating the frequent patters. As we mentioned before the proposed method uses a compact 2D array data structure. Only one 2D array is created for mining the whole frequent patters. After that the 2D array will be traversed to generate the entire frequent items. Using the 2D array will save a lot of memory compared with FP-growth which creates many conditionals FP-trees recursively. Building the conditional FP-trees requires linking many nodes together and each node has associated data like: item name, parent, left's child, right's sibling, count, next. Moreover, building each conditional FP-tree requires scanning the target part of the FP-tree two times.

The following example explains the steps for generating the frequent patterns based on the FP-tree in figure 5. For any frequent item such as a_i in the header table, the transactions that includes a_i can be obtained by following the head node-linkes for a_i as it is shown in figure 5. Let us start mining items bottom-up according to the header table. *G* is a frequent item with support *3* (*G:3*) and two paths can be obtained by following the head of node G: *<B:7, C:6, A:4, D:3, G:2>* and *< B:7, A:1, G:1>*. We identify which string appears together with *G*. The first *G*'s path is*<B:2, C:2, A:2, D:2, G:2>*, whereas the second one is *< B:1, A:1, G:1>*. Our method is not like FP-growth ,it does not construct FP-tree based on this conditional pattern base (which is called *G*'s conditional FP-tree), instead it inserts these paths into 2D array as it is shown in figure 6 (c) .The dimensions of this array as the followings :The number of columns is equal to the length of the header table in Figure 5 (a) (which is "6" columns) ,whereas for the number of rows ,each node in the header table has a set of paths we need to find the node with the maximum number of paths which is node "*F*" . Then the number of rows is equal to the maximum number of paths (which is "3"). These dimensions can be obtained using some counters during building the FP-tree. Noted that only single 2D array is needed. The order of the items in the 2D array is like their order in the main header table. In Figure 6 (c) the third row includes the value "2" for items {*B, C, A, D, G*} and "0" for {*F*}, which can be obtained from the first branch for item *G*. We set *F*'s value to "0" because F's order is less that *G*'s order and does not appear in the first path for *G*. The fourth row includes the value "1" for items {*B, A, G*} and "0" for {*C, D, F*}. Whereas, the fifth row (branch 3) is not used in this case because only two paths can be obtained by following the head of item *G*. The second row of the table in Figure 6 (c) indicates that each item has a specific index that is used for mining the data.



(a) The set of paths ending in G

Item	Support
B	3
A	3
C	2
D	2

(b) Header table for G

Item name	B	C	A	D	F	G
Index	1	2	3	4	5	6
Branch 1	2	2	2	2	0	2
Branch 2	1	0	1	0	0	1
Branch 3	0	0	0	0	0	0

(c) The 2D array for suffix G

Figure 6. Mining Frequent patterns using the proposed method

After building the 2D array, it will be used to generate all the frequent patterns that include the item *G* and the part of FP-tree that includes item *G* will not be scanned anymore, Moreover, no conditional FP-trees will be generated and memorized. The frequent items with suffix *G* will be rearranged according to their frequencies in descending order as the following {*B:3, A:3, C:2, D:2*}. Noted that the content of the 2D array will not be changed during generating the whole frequent items ending with suffix *G*. Only one-dimensional array is used to store the new order of frequent Patterns, instead of changing the 2D array. For explanation purposes, Table 2 shows how the data in the original 2D array will be if we rearrange it according to the new order.

From the *G* column (which is shaded with dark colour) we need to store the regions (list of branches) that include values greater than "0" for item *G*, which are {1, 2} and the total number of locations is "2".

Table 2. Rearranging the data in the 2D array for suffix G

Item name	B	A	C	D	F	G
Branch 1	2	2	2	2	0	2
Branch 2	1	1	0	0	0	1
Branch 3	0	0	0	0	0	0
Total support	3	3	2	2	0	3

Storing the regions is important to reduce the searching area because many regions could have useless data ("0"). Traversing the 2D array will be done in depth order from the less frequent item (which is "*D*") to the most frequent item (which is "*B*") and recursively. The algorithm starts to find the patterns end with "*DG*" as follow: Traversing "*D*" column in the regions {1, 2} to find the new regions in column *D* that have values greater that "0". The result of searching is one region {1} and the total support of this region is "2" as it shown in Table 3.

The proposed approach reduces the search space by keep saving the new regions, and the number of searched regions is shrinking gradually by increasing the length of frequent pattern.

After that the list of frequent items that have order less that *D* will be scanned in the last list of regions to know their supports. Noted that it is not needed to scan all the items that have order less than the target element (which is "*D*" in our example), just the frequent ones {*B, A, C*} in our case will be scanned in the last list of regions {1} to find their frequencies, then rearrange the frequent ones according their frequencies. As we can see in table 3 all the items <*B: 2, A: 2, C: 2*> are frequent and their order will be <*B, A, C*>.

To find the frequent items end with "*CDG*" we need to scan the column "*C*" in the last list of regions which is branch {1} ,the result will be {1} and the total number of regions is "1".

Table 3. Mining Frequent patterns with suffix "*DG*"

Item name	B	A	C	D	F	G
Branch 1	2	2	2	2	0	2
Branch 2	1	1	0	0	0	1
Branch 3	0	0	0	0	0	0
Total support	2	2	2	2	0	3

Table 4. Mining Frequent patterns with suffix "*CG*"

Item name	B	A	C	D	F	G
Branch 1	2	2	2	2	0	2
Branch 2	1	1	0	0	0	1
Branch 3	0	0	0	0	0	0
Total support	2	2	2	2	0	3

In this stage we can notice that there is no any change in the total number of regions and it is "1" like the last stage .In this case it is not necessary to go further more and we just generate the frequent patterns that end with "*DG*" by taking the combinations of the set {*B,A,C*} with the suffix "*DG*". The last operation will save the execution time specially if the database is dense. The algorithm will apply the above the operations to find frequent patterns that end with "*CG*","*AG*" and "*BG*", respectively. Here we will explain another example how to generate frequent patterns that end with "*CG*". First, traversing "*C*" column in the regions {1, 2} to find the new regions in column *C* that have values greater that "0". The result of searching is one region {1} and the total number of regions is "1". Items {*B, A*} will be scanned in the last list of regions {1} to find their frequencies, then rearrange the frequent ones according their frequencies. As we can see in table 4 all the items <*B: 2, A: 2*> are frequent and their order will be <*B, A*>. After scanning column, A to find the frequent items with suffix "*ACG*", we found that the number of regions is like the one in the last step, therefore, we generate the frequent patterns that end with "*CG*" by taking the combinations of the set {*B, A*}

with the suffix "*CG*". Algorithms 2 to 4 show three pieces of pseudocodes that are used to generate the frequent patterns using our approach.

Algorithm 2: 2D Projection

2D_Projection (*FP-tree, H, min-sup*)

Input: FP-tree constructed by Algorithm1;
Minimum support *min-sup.*
Header table for FP-tree *H*
Output: The set of all **frequent** patterns.

1. **Begin**
2. Create 2D array, which is denoted as *2D*, the dimensions of *2D* as follows:
3. Number of columns: the length of the header table.
4. Number or rows: Number of paths for the item with the maximum number of paths.
5. **For** each item *x* in *H*
6. Get the node *n* pointed by the link of *x* in *H*
7. **While** $n \neq$ null
8. Find a path $P{:}x_1, x_2, x_3, x_m$ from the parent node of *n* to the child node of the root;
9. Add a new row to *2D* that includes the value *c* for each item in *P*, where *c* equals to the support of node *n*.
10. Reset a new node *n* to the next node pointed by the link of current node.
11. **End while**
12. Scan *2D* in the rows that have the item *x* with values greater than *0* and add the set of locations to Loc_x.
13. Let L_x be the set of items that have order less than x in *H*
14. **For** each item y in L_x
15. Scan *2D* in Loc_x for the rows that have the item *y* with values greater than *0*, denote these new locations as Loc_{yx}
16. **If** the frquency counter for $Loc_{yx} \geq$ *min-sup* then
17. Freq_L_x= Freq_$L_x \cup$ y
18. **End if**
19. **End for**
20. Rearrange Freq_L_x in descending order.
21. **For** each item y in Freq_L_x
22. Print {*yx*}
23. Suffix= "*x*"
24. **2D_SubProjection** (*2D*, Freq_L_x ,Loc_{yx},*y, suffix-x*)
25. **End For**
26. **End for**
27. **End**

Algorithm 3: 2D SubProjection

2D_SubProjection (*2D, Freq_*L_x, Loc_x,*x, , suffix-x*)

Input:

2D: 2D array that includes paths for one item in the header table.

*Freq_*L_x: the list of frequent items for the parent of item *x*

Loc_x,*x,* : the list of location for the frquent item *x*

suffix-x: the previous frequent items with item *x*

*min-sup: T*he Minimum support threshold.

Output: The set of all **frequent** patterns.

1. **Begin**
2. Let L_x be the set of items that have order less than *x* in *Freq_*L_x
3. Freq_L_x={}
4. **For** each item y in L_x
5. Scan *2D* in Loc_x for the rows that have the item *y* with values greater than *0*, denote these new locations as Loc_{yx}
6. **If** the frquency counter for $Loc_{yx} \geq$ *min-sup* then
7. Freq_L_x= Freq_$L_x \cup$ y
8. **End if**
9. **End for**
10. Rearrange Freq_L_x in descending order.
11. **For** each item y in Freq_L_x
12. **If** | Loc_x |= | Loc_{yx} | then // single path
13. **Single_Path**(L_x ,y, *suffix-x);*
14. **Else if**
15. Print {*yx*$\cup$ *Suffix-x* }
16. *Suffix-y*=y $\cup$*Suffix-x*
17. **2D_SubProjection** (*2D*, Freq_L_x ,Loc_{yx},*y, suffix-y*)
18. **End if**
19. **End For**
20. **End**

Algorithm 4: Frequent items generating from single path.

Single_Path(L_x ,*y*, *suffix-x);*

1. Let *P* be the set of items that have order less than *y* in L_x
2. For each combination (denoted as β) of the items in thelist L_x
3. Do Generate pattern $\beta \cup a$ with support = minimum support of item in *y*;

EXPERIMENTAL RESULTS

In this part we present the performance comparison between the proposed algorithm and FP-growth. All experiments were conducted on Intel ® Core (TM) i3 CPU 2.3 GHz, 4 GB memory using C Programming Language and running on Microsoft windows 7 environment. The synthetic dataset *T10I4D100K* from IBM Almaden research centre and two real datasets *connect* and *Mushroom* from the FIMI repository website were used to make the comparison. Our algorithm saves a lot of memory compared with FP-growth and that is because it does not create conditional FP-trees recursively. FP-growth consumes much memory to save these trees in the main memory.

In figure 7 to 9, we can see that the proposed algorithm is indeed very effective to reduce the memory consumption compared with FP-growth algorithm, especially when the minimum support is small. Also, the amount of saved memory is increased gradually by increasing the size of the dataset. For the all tested datasets the proposed method consumes less memory for the all minimum support thresholds. The main memory usage in FP-growth decreases gradually by increasing the minimum support, this because the number of created conditional FP-trees will be decreased gradually with increasing the minimum support. Whereas using the 2D array approach, we just increase or decrease the size of the 2D array once before starting to mine the database. In our approach after creating the 2D array during mining the data the main memory usage will not be increased. For the dataset *T10I4D100K* the proposed method reduced the memory consumption 28% to 95% compared with FP-growth, whereas it reduced up to 94% for *Mushroom* dataset when the minimum support is equal "300". *connect* dataset is the largest one among these datasets, it is size about 9 MB. Mining operation for *connect* dataset consumes less than 1 MB to mine the data over the all minimum support thresholds. However, FP-growth consumes from 20 MB to 75 MB to mine the same data. This indicates that the proposed method saves more memory by increasing the dataset size and when minimum support thresholds is low, the reason for saving more memory is by increasing the dataset size and decreasing the minimum support FP-growth needs to generate massive number of conditional FP-trees.

However, for the running time the proposed method reduced the computing time for some datasets with different percentages. Figures 10 to 12 show the execution time for the proposed algorithm and FP-growth using the datasets *Mushroom*, *connect* and *T10I4D100K*, respectively, with different minimum support values.

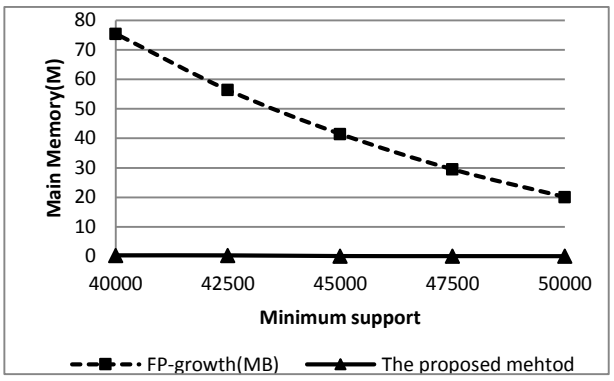


Figure 7. Main memory used by the algorithms on dataset *connect*

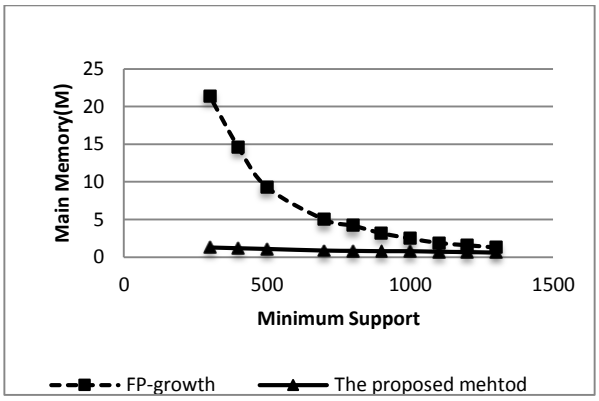


Figure 8. Main memory used by the algorithms on dataset *Mushroom*

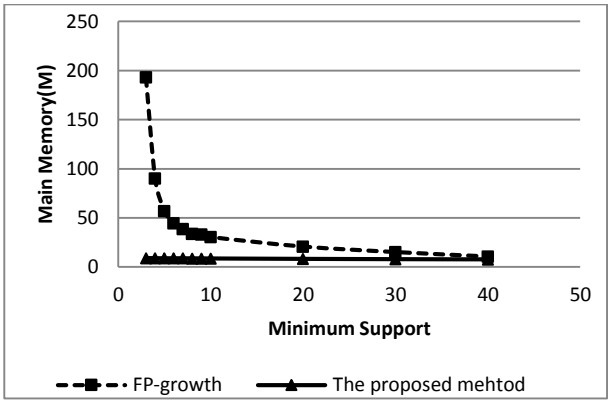


Figure 9. Main memory used by the algorithms on dataset *T10I4D100K*

Please note that the execution time includes also writing the frequent patterns to text file, we use the same output formats like the original FP-growth algorithm without any optimizations on the output. For checking the correctness of the proposed algorithm, we have checked that the proposed

algorithm generates the same number of frequent itemset for all the tested datasets. The solid black line refers to the proposed algorithm whereas the dotted black line refers to FP-growth.

Figure 10 shows the CPU time for running the two algorithms in dataset *Mushroom*, the proposed method running times for the all *min_sup* cases from 300 to 1300 are 4% to 9% less than FP-growth.

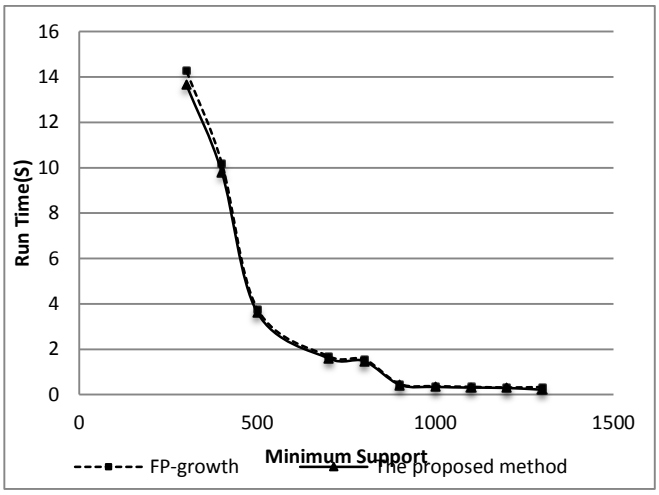


Figure 10. Execution time for the proposed method and FP-growth using the data set *Mushroom*.

In figure 11 our algorithm run time always less that FP-growth and it reduced the CPU time about 5 % for *connect* dataset which is the biggest dataset compared with the others datasets that we used. Whereas it reduced 4%-15% for *T10I4D100K* when the minimum support is less than 7 and the running time is like FP-growth for the other cases of minimum support as it is shown in figure 12.

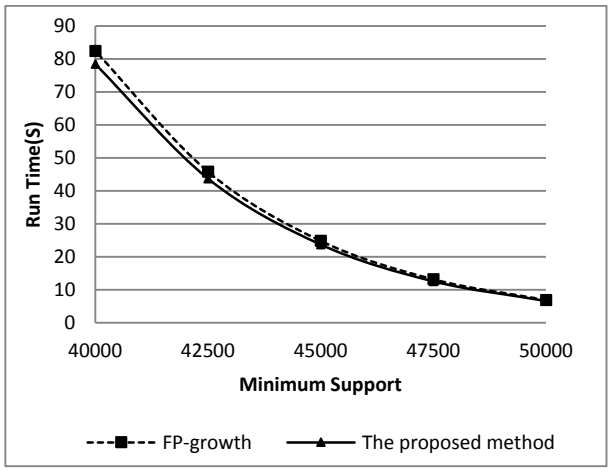


Figure 11. Execution time for the proposed method and FP-growth using *connect* data set.

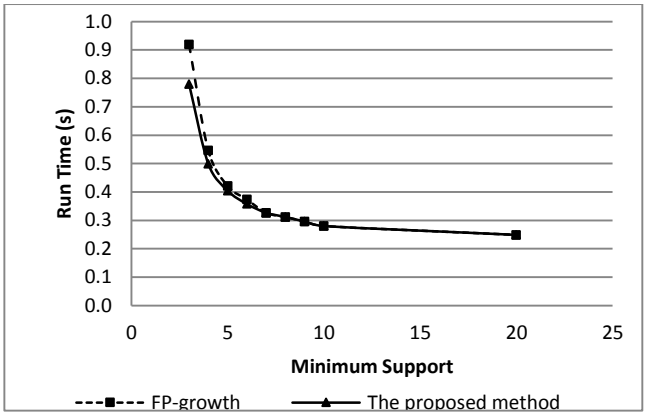


Figure 12. Execution time for the proposed method and FP-growth using T1014D100K data set.

CONCLUSIONS

In this paper we have proposed a novel algorithm to generate the frequent itemsets based on the well-known algorithm FP-growth. The proposed method uses the FP-tree to build a new data structure based on the 2D array to generate the frequent itemsets instead of constructing conditional FP-trees. Many synthetic and real datasets from FIMI repository website were used to test the performance of our algorithm. The experimental results show the memory consumption significantly reduced using our approach compared with FP-growth, also it reduces the running time for some datasets. Our approach builds single 2D array to mine the data instead of recursively generating massive number of conditional FP-tees.

REFERENCES

[1] Agrawal, R., T. Imielinski, and A. Swami (1993). Mining Association Rules between Sets of Items in Large Databases. ACM SIGMOD Record. 22(2), 207-216.

[2] Islam, A.B., T. Chung (2011). Improved Frequent Pattern Approach for Mining Association Rules. International Conference on Information Science and Applications (ICISA), 1 – 8

[3] Agrawal, R., and R. Srikant (1995). Mining sequential patterns. In Data Engineering. ICDE '95 Proceedings of the Eleventh International Conference on Data Engineering, Washington, DC, USA, 3-14.

[4] Yang, Z., Y. Wang and M. Kitsuregawa (2007). LAPIN: Effective Sequential Pattern Mining Algorithms by Last Position Induction. In Advances in Databases: Concepts, Systems and Applications 4443, 1020-1023.

[5] Han, J., M. Kamber, and A. Tung (2001). Spatial clustering methods in data mining: A survey. In Geographic Data Mining and Knowledge Discovery, edited by Harvey J., Harvey J., Taylor and Francis, 1–29.

[6] Hartigan, J. And M. Wong (1979). Algorithm AS136: "A k-means clustering algorithm. Applied Statistics, 28, 100-108.

[7] Thabtah, F. (2007). A review of associative classification mining. The Knowledge Engineering Review, 22(1), 37-65.

[8] Ali, K., S. Manganaris, and R. Srikant (1997). Partial classification using association rules In KDD 97, 115-118.

[9] Liu, B., W. Hsu, and Y. Ma, (1998). Integrating classification and association rule mining. In Proceedings of the International Conference on Knowledge Discovery and Data Mining. New York, NY: AAAI Press, 80–86.

[10] Grahne, G. and J. Zhu (2003). Efficiently Using Prefix-trees in Mining Frequent Itemsets. In: Proceeding of the ICDM'03 international workshop on frequent itemset mining implementations (FIMI'03), Melbourne, FL, 123–132.

[11] Agrawal, R. and R. Srikant (1994). Fast algorithms for mining association rule. In Proceedings of the 20th International Conference on Very Large Data Bases, Morgan Kaufmann, Santiago, Chile, 487–499.

[12] Savasere, A., E. Omiecinski, and S. Navathe (1995). An efficient algorithm for mining association rules in large databases. VLDB '95 Proceedings of the 21th International Conference on Very Large Data Bases, San Francisco, CA, USA, 432-444.

[13] Zaki, M., S. Parthasarathy, M. Ogihara, and W. Li (1997). New algorithms for fast discovery of association rules. In Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining. Menlo Park, CA: AAAI Press, 283–286.

[14] Han, J., Pei ,J. , and Y. Yin (2000). Mining frequent patterns without candidate generation". In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data. Dallas, TX: ACM Press, 1–12.

[15] Sohrabi, M. K., and A. A. Barforoush (2013). Parallel frequent itemset mining using systolic arrays. Knowledge-Based Systems, 37,462–471.

[16] Yen, S., C. Wang, And L. Ouyang (1995). A Search Space Reduced Algorithm for Mining Frequent Patterns", Journal of Information Science and Engineering, 28, 177-191

[17] Park, J., M. Chen, P. Yu (1995). An effective hash-based algorithm for mining association rules. In Proceedings of ACM SIGMOD International Conference on Management of Data, New York, NY, USA 175–186.

[18] Holt, J. and S. Chung (2002). Mining association rules using inverted hashing and pruning. Information Processing Letter 83, 211–220

[19] Shengbo J. , Xueli P., Yi Xiao ,Qiangming Z., Miao L., and Jun Y. (2017). Identification of vulnerable lines in large power grid based on FP-growth algorithm. The Journal of Engineering,13,1862-1866.

[20] Agrawal, S., and V. Sejwar (2017). Frequent Pattern Model for Crime Recognition. International Journal of Computational Intelligence Research, 13(6), 1405-1417.

[21] Pamba, V., E. Sherly, and K Mohan (2017). Automated information retrieval model using FP growth based fuzzy particle swarm optimization. International Journal of Computer Science and Information Technology, 9(1).

[22] Liu, G., H. Lu, Y. Xu and J. X. Yu, (2003). Ascending frequency ordered prefix-tree: efficient mining of frequent patterns. In Database Systems for Advanced Applications, 2003. (DASFAA 2003). Proceedings. Eighth International Conference on .65-72