Journal of
Marine Science and Technology

Volume 24 | Issue 2 Article 12

EFFICIENTLY MINING FREQUENT ITEMSETS IN TRANSACTIONAL DATABASES

Salah Alghyaline
Department of Computer Science and Engineering, National Taiwan Ocean University, Keelung, Taiwan,
salahshaman2007@gmail.com

Jun-Wei Hsieh
Department of Computer Science and Engineering, National Taiwan Ocean University, Keelung, Taiwan.

Jim Z. C Lai
Department of Computer Science and Engineering, National Taiwan Ocean University, Keelung, Taiwan.

Recommended Citation
Alghyaline, Salah; Hsieh, Jun-Wei; and Lai, Jim Z. C (2016) "EFFICIENTLY MINING FREQUENT ITEMSETS IN TRANSACTIONAL DATABASES," *Journal of Marine Science and Technology*: Vol. 24: Iss. 2, Article 12.
DOI: 10.6119/JMST-015-0709-1
Available at: https://jmstt.ntou.edu.tw/journal/vol24/iss2/12

Journal of Marine Science and Technology, Vol. 24, No. 2, pp. 184-191 (2016)
DOI: 10.6119/JMST-015-0709-1

EFFICIENTLY MINING FREQUENT ITEMSETS IN TRANSACTIONAL DATABASES

Salah Alghyaline, Jun-Wei Hsieh, and Jim Z. C. Lai

Key words: data mining, frequent pattern, frequent itemsets, FP-growth.

ABSTRACT

Discovering frequent itemsets is an essential task in association rules mining and it is considered to be computationally expensive. To find the frequent itemsets, the algorithm of frequent pattern growth (FP-growth) is one of the best algorithms for mining frequent patterns. However, many experimental results have shown that building conditional FP-trees during mining data using this FP-growth method will consume most of CPU time. In addition, it requires a lot of space to save the FP-trees. This paper presents a new approach for mining frequent item sets from a transactional database without building the conditional FP-trees. Thus, lots of computing time and memory space can be saved. Experimental results indicate that our method can reduce lots of running time and memory usage based on the datasets obtained from the FIMI repository website.

I. INTRODUCTION

Association rules mining (ARM) suggested by (Agrawal et al., 1993) is one of main pillars of data mining and knowledge discovery (Bashir and Baig, 2006). It discovers the correlations among items in transactional database that reveals interesting relations between items. It can be applied to many applications such as Web usage mining, intrusion detection, production, bioinformatics, market basket analysis, and so on. In general, ARM includes two essential steps: (1) finding the frequent itemsets in the training dataset; (2) generating rules based on the discovered itemsets. The first step requires extensive computation time and storage and thus becomes a challenging problem in ARM (Liu et al., 1998). Many algorithms were proposed to find various frequent itemsets to reduce CPU time and memory consumption (Agrawal and Srikant, 1994; Savasere et al., 1995; Zaki et al., 1997; Han et al., 2000; Leung et al., 2008; Sohrabi and Barforoush, 2013). Because it requires more computations and memory, the first step is relatively more hard and important than the second one, (Thabtah, 2007; Schlegel et al., 2011).

In the literature, many frequent itemsets mining algorithms have been proposed during last two decades. Among them, four most popular mining algorithms are Apriori (Agrawal and Srikant, 1994), FP-growth (Han et al., 2000), FPgrowth* (Grahne, 2003), and Ascending Frequency Ordered Prefix-Tree (AFOPT) (Lu et al., 2003), respectively. Apriori algorithm tries to mine frequent items at different stages in which each one represents a different length of the itemsets. The first stage scans the whole dataset to count the frequency of each item and gets a list of frequent items with length "1" (1- items). After that, the list of frequent items of stage one will be used to generate the list of candidates for stage two and then infrequent items will be pruned. However, another scan for the dataset is required again to check the frequencies for each candidate to obtain the frequent items with length "2" (2-items). The algorithm moves from one stage to another until no frequent items are existed. The bottleneck of Apriori algorithm is that it needs to scan the whole dataset many times and thus consumes a lot of memory and CPU time especially if the minimum support is low (Zaki et al., 2004).

As to FP-growth method, it is faster than Apriori algorithm according to many other evaluation reports (Han et al., 2000), because it does not generate any candidates and scans the database only twice. In FP-growth, the first scan is to find the frequent items with length "1" based on a given minimum support, and then sorts the frequent items in ascending order according to their supports. The second scan builds the FP-tree which is a prefix tree structure for sorting compressed and useful information about frequent patterns. After that, without mining the whole database, it needs to mine only the FP-tree. The last step is mining the frequent patterns using a pattern growth method which builds various conditional FP-trees recursively. Because FP-tree is much smaller than the original database, the FP-growth method is more effective than Apriori algorithm.

The FPgrowth* approach uses FP-tree structure in conjunction with the FP-array structure to reduce the number of iterations for traversing the FP-tree. Building each conditional FP-tree in the original FP-growth requires scanning the FP-trees

Paper submitted 12/15/14; revised 06/30/15; accepted 07/09/15. Author for correspondence: Salah Alghyaline (e-mail: salahshaman2007@gmail.com).
Department of Computer Science and Engineering, National Taiwan Ocean University, Keelung, Taiwan.

twice: the first scan is to find out the support of items in the conditional database and the second scan is to build a new conditional FP-tree according to the new ordered frequent items. However, the FPgrowth* method uses the FP-array technique to omits the first scan for building the conditional FP-trees and thus drastically speeds up its efficiency. The constructed array can be accessed in O (1). Additionally, building the FP array for building any conditional FP-tree is performed simultaneously while building the upper-level of the conditional FP-tree. In summary, FPgrowth* works very well for sparse and very large datasets but the FP-growth outperforms the FPgrowth* for dense datasets.

CT-PRO (Sucahyo and Gopalan, 2004) proposes a new data structure called CFP-Tree instead of FP-tree for mining data. Depending on the dataset characteristics, the CFP-Tree makes the FP-tree highly compacted and thus can use only half of the nodes to mine data. The scheme uses a bottom-up method to traverse the CFP-Tree to generate frequent patterns in a non-recursive approach.

Another improvement in FP-growth is called Ascending Frequency Ordered Prefix-Tree (AFOPT). Three key factors to influence the performance of generating frequent items are: the total number of conditional FP-trees, time, and memory for traversing trees, the order of the items in the conditional database, and the way of traversing it. AFOPT rearranges the frequent items in ascending order to present a compact Prefix-tree data structure for data mining. The dynamically ascending order will minimize the number of conditional databases. Additionally, AFOPT traverses the conditional databases in a top-down rather than a bottom-up order. As a result, the cost of traversing the conditional databases will be significantly reduced.

Many evaluation results have shown that building a conditional FP-trees recursively to mine frequent patterns will consume most of CPU time. Each conditional FP-tree requires scanning the FP-tree twice. The first scan needs to build the header table to find items frequencies; while the second scan builds the conditional FP-tree according to the new order of item frequencies after eliminating infrequent items. The FP-growth method will build many conditional FP-trees recursively until reaching single node and thus consumes a lot of CPU time and space especially when dataset is huge and sparse.

This paper proposes a new algorithm called FP-growth+ for extracting the frequent items without building conditional FP-trees. The proposed algorithm improves the inefficiency of FP-growth method; it first scans the database to find all frequent items and the second scan is for inserting the frequent items from each transaction into the FP-tree as a branch. This method uses a new data structure for traversing the FP-tree based on set of arrays instead of building the conditional FP-trees. Experimental results have been conducted on many

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

Fig. 1. Header table for *min_sup* = 2.

synthetic and real data sets (from FIMI repository website[1]) and show that the proposed method can significantly reduce the usages of CPU time and memory.

The rest of paper is organized as follows: Section 2 describes details of FP-growth algorithm in mining frequent patterns. Our "FP-growth+" algorithm for mining frequent patterns is proposed in section 3. Section 4 shows the experimental results. Finally, we conclude this paper in section 5.

II. FP-GROWTH ALGORITHM

The FP-tree is a compact representation for mining various frequent items from databases. The FP-growth algorithm uses it to mine data more efficiently than the Apriori approach. Firstly it scans the database to find the occurrence of each item, then sorts and gets higher frequent items if they exceed the minimum support threshold in descending order and then to be recorded inside a header table according to their frequencies. Fig. 1(b) shows the header table for the given transactions in Fig. 1(a). After that, it uses the second scan to build the FP-tree. As shown in Fig. 2(b), if some transactions share a common prefix with the same order, all the shared parts are then merged in the same path to construct the FP-tree. By sorting the transactions in a descending order, the shared paths are used to facilitate the process of traversing the FP-tree items. The items with the same labels will be connected together via a linked list. The beginning of this linked list is marked in the above mentioned header table. Fig. 2 shows how the header table links together all the nodes having the same label in the FP-tree.

A crucial task in the FP-growth algorithm is building the conditional FP-trees for mining frequent items. At the beginning, all the items in the header table will be visited from the bottom of the header table. Then, we will follow the path containing the target item such as a_i starting from a_i's head in

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

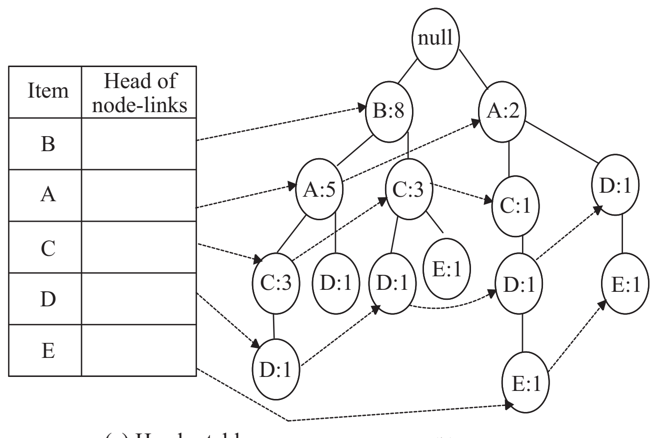
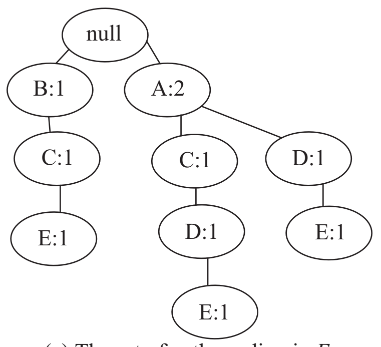


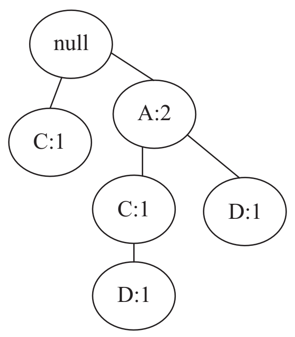
Fig. 2. An example of FP-tree with *min_sup* = 2.



(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*

Fig. 3. Building conditional FP-tree.

the FP-tree header table to determine the new counts of items to build a new header table as shown in Fig. 3(b). Branches that contains item a_i will be visited one more times to find its corresponding item set in the tree and insert them into the conditional FP-tree according to its new order in the new header table after removing all infrequent items. Note that, we rebiuld the conditional FP-tree because the order of the items is not like the original FP-tree order; thus, if the items do not statisfy the minimum support (*min_sup*), they will be removed iterativly until we reach the root. Fig. 3(c) shows an example of conditional FP-tree which is obtained by visting branches along the linked list beginning from *E* in the FP-tree shown in Fig. 2. This procedure is recursively applied for the set of all possible paths in the E-conditional FP-tree, ending at *D*, *C*, and *A*. The taks of building the conditonal FP-tree for any items will stop in case that the new FP-tree includes only one single path. Then all of the subsets will be reduced along this path and then merged together with the corresponding suffex. Table 1 summarizes the set of conditional patterns and the conditional FP-trees.

Table 1. Mining patterns by creating conditional pattern base.

Item	Conditional patterns	Conditional FP-tree
E	{(BC:1), (ACD:1), (AD:1)}	{(A:2), (C:2), (D:1)}| 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 frequent patterns by building and traversing the conditional FP-trees will consumes a lot of CPU time and memory. According to the experiments conducted by (Grahne, 2003), almost 80% of CPU time is spent for traversing the FP-tree. If the traversing time of the FP-tree can be reduced, the overall time in mining patterns can be much speeded up. To achieving this goal, a new traveling technique will be proposed without building the time-expensive conditional FP-trees. The proposed method takes advantages of arrays and their pointers to make traversing the FP-tree easier and more efficient during the process of building the FP-tree. Details of this algorithm will be explained in the next section.

III. THE PROPOSED METHOD

There are two steps during mining data using the FP-growth algorithm: (1) building the FP-tree, and (2) traversing the FP-tree for mining the database. Our approach shares the first step with FP-growth and utilizes the overlapping information between two transactions. We use a new method for traversing the FP-tree instead of building conditional FP-trees. This can make mining the FP-tree more efficient than mining the database. In the following sections, we will describe the two steps of our proposed method in more details.

1. FP-tree Constructing

The proposed method starts by scanning transactions in a given dataset to count the support of each item. If a frequent

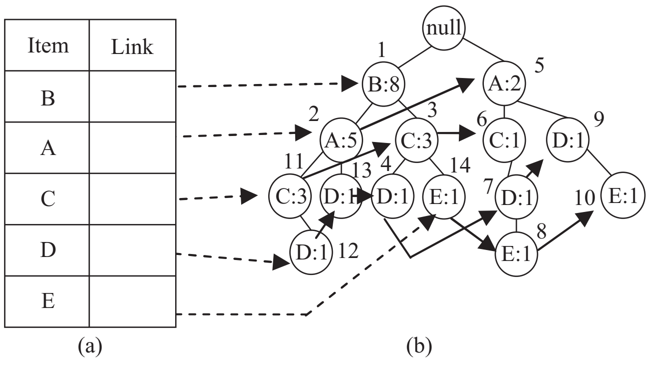


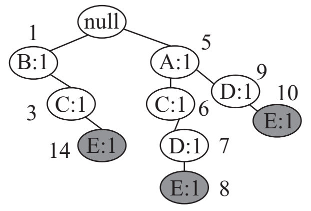
Fig. 4. FP-tree (*min_sup* = 2) using the proposed algorithm.

item exceeds the minimum support threshold, it will be used to build the FP-tree. Then, according to the supports, the items will be sorted in a descending order. After that, the FP-tree can be constructed using these sorted transactions. While building the FP-tree for each item (node), two operations are performed. For the first one, each node will associate with a specific identifier called *node-id* to indicate the node location in the FP-tree. For the second operation, an array of pointers, called *node-head*, is used to record the pointer of each node in the FP-tree and is indexed by the *node-id*. This header table can link all the items with the same labels together. The two data structures can make traversing the FP-tree easier without building the conditional FP-trees. Because only an additional attribute (*node-id*) is added to each node in the FP-tree by using an 1-D array to keep all locations for different nodes, the new data structure does not consume much memory and CPU time.

Fig. 4 shows an example to illustrate how the FP-tree is built via our proposed method from the set of transactions shown in Fig. 1. An Identifier number (*node-id*) is tagged to each node in the second scan for building the FP-tree. The order of each *node-id* in the FP-tree is arranged by the order when it is inserted into the FP-tree. Its value starts from "1" and is increased when a new node is inserted into the FP-tree. Clearly, the total number of locations is equal to the total number of nodes in the FP-tree. Algorithm 1 summarizes details of our proposed method to build the FP-tree.

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 D to generate the set L1 of frequent items.
2. Sort the items of each transaction in D according to the
   descending order of L1.
3. Create counter with name node-id and initialize it with 0.
4. Create array of pointers with name node-head.
5. Create the root of FP-tree T and denote it as "null" (0th root)
6. For each $t \in D$ and $l = 1$ to $|t|$
     If $t[l] \in CN(l-1)$,
       a) Denote the child node as temporary lth node;
       b) 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 $t[l] \notin CN(l-1)$;
       (a) Denote this child node as temporary lth node with
           frequency = 1;
       (b) Set the item name of the child node as t[l];
       (c) Set the id for this node as node_id + = 1, and
           node-head[node-id] points to this child node;
       (d) Update $CN(l-1) = CN(l-1) \bigcup \{t[l]\} CN$.
```



(a) The set of paths ending in E.

item	support
C	2
D	2
A	2

(b) Header table for E

item	Locations list
C	{3:1,6:1}
D	{7:1,9:1}
A	{5:2}

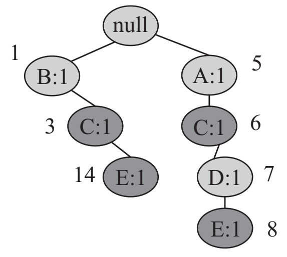
(c) Frequent items ending in E.

Fig. 5. Constructing the locations list for item E.

2. Frequent Pattern Generation

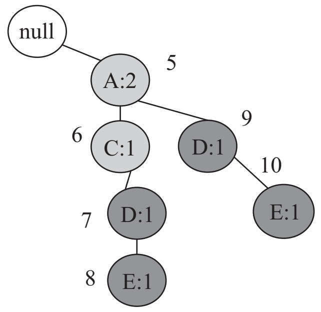
After building the FP-tree, the next task is to mine frequent patterns from it. The first step is to trace the linked list of the item a_i in the header table *H* and then visit the paths that include item ai in the FP-tree to count its visited times. Fig. 5(a) shows an example that the set of all paths include item *E*. After scanning the paths, three frequent items can be found, i.e., nodes *C*, *A*, *D*. Item *B* is ignored because it is not frequent. Fig. 5(b) shows the new header table *New_H* for frequent items ending with *E*. Another scan on the FP-tree is performed to find the locations of frequent items in *New_H* according to their supports. If the item has more than one children with the label *E*, its support will be the sum of its children supports. For example, item *A* in location "5" has two children; each one with frequency "1". Thus, the support of *A* in this case is "2".

After scanning the FP-tree, we can get the suffixes {*CE*, *AE*, *DE* }. Fig. 6 shows the set of paths that include these suffixes. Their locations and frequencies are recorded as the sets *CE* {3:1, 6:1}, *DE* {7:1, 9:1}, *AE* {5:2}, whereas {3, 6} are the set of locations for suffix *CE,* and {1, 1} represents the frequencies for their locations. After that, our approach uses the list of locations to generate all the frequent items that end with *CE*, *AE* and *DE,* recursively. To obtain the frequent items that end



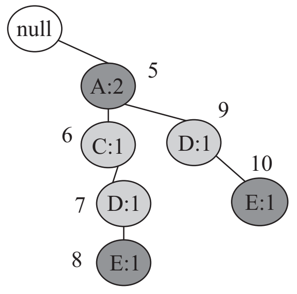
Item	support
B	1
A	1

(a) The set of paths ending in CE with the header table.



item	support
A	2
C	1

(b) The set of paths ending in DE with the header table.



item	support

(c) The set of paths ending in AE with the header table

Fig. 6. Generating frequent patters with suffix E.

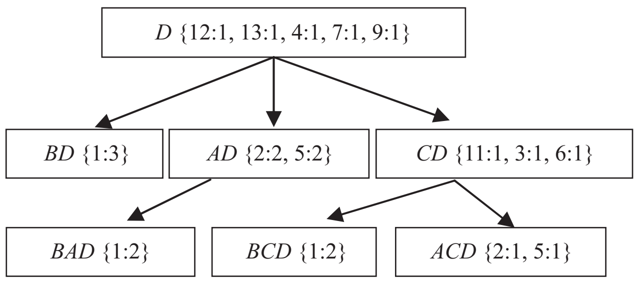


Fig. 7. Finding frequent patterns from Item D.

with *CE*, the paths that include the item *CE* from the locations {3, 6) are used to determine their new counts along these paths. Fig. 6(a) shows the patterns {*ACE*, *BCE*} with frequency "1" which is less than the minimum support. However, suffix *DE* has one frequent item, i.e., *ADE*. Finally, for suffix *AE* we stop calculating the frequent items because its parent is null.

Fig. 7 explains another example of how can we recursively find the frequent patterns from the item *D* using the proposed method. The first set of frequent items with length "2" are obtained by following the linked list of item *D* starting from the main header table and the results are {*BD*, *AD*, *CD*}. The remaining set with length greater than "2" can also be generated. For example, the locations {11, 3, 6} are used to find all frequent patterns that end with *CD* as shown in Fig. 7. Algorithm 2 to Algorithm 4 show three pseudocodes of our approach used to generate the frequent patters.

Algorithm 2: FP-growth+ phase one.

```
FP-growth+_ phase1(FP-tree, H , min-sup)
Input: FP-tree constructed by Algorithm 1.
       H is the Header_table for FP-tree.
       min-sup is the minimum support threshold.
Output: The complete set of frequent patterns.
1.  Begin
2.    For each item x in H do
3.       Get the node n pointed by the link of x in H;
4.       While n ≠ null
5.          Find a path x1, x2, ..., xm from the parent
              node of n to the child node of the root;
6.          Count the frequency for each item z along
              this path
7.          Reset a new node n to the next node pointed
              by the link of current node n .
8.       End while
9.       For each counted item z
10.        If the frequency count of z ≥ min-sup then
11.           Insert z into a frequent item list $F_x$.
12.           Add the set of locations for item z to the
13.                list $L_x[z]$
14.           Print {zx}
15.        End if
16.      End for
17.    suffix = "x"
18.    FP-growth+_ phase2 ($F_x$, $L_x$, suffix);
19. End for
20. End
```

Algorithm 3: FP-growth + Phase two.

```
Procedure FP-growth+ Phase2 ($F_x$, $L_x$, suffix-x)
Input:
     $F_x$: the list of frequent items traced from node x;
     $L_x$ is the list of locations for the frequent items in $F_x$;
     Suffix-x is the previous Frequent items with item x.
     min-sup is the minimum support threshold.
Output: Full set of frequent patterns.
1. Begin
2. For each element z in list $F_x$
3.    If z exists in one location in the tree
4.        Let P is the single path for z and
                $a = z \cup$ Suffix-x.
5.        Single_Path (P, a);
6.    Else if z exists in many locations
7.      Use the list of locations for z ($L_x[z]$) to
```

```
            traverse FP-tree;
8.        Find the frequent list of items from z nodes;
9.        Denote this list as $F^Z$ and their locations as $L_Z$.
10.           For each frequent item such as y in $F_Z$
11.               Print {$yz \cup suffix\text{-}x$}
12.           End for
13.               $suffix\text{-}y = z \cup suffix\text{-}x$
14.           FP-growth+_ phase2 ($F_Z$, $L_Z$, suffix-y);}
15.       ENDIF
16.     End for
17.  End
```

Algorithm 4: Frequent items generating from single path.

```
Single_Path (Tree, a)
1. Let P be the single prefix-path part of Tree;
2. For each combination (denoted as $\beta$) of the nodes in the
   path P
3. Do
     Generate pattern $\beta \cup a$ with support = minimum support
     of nodes in $\beta$;
```

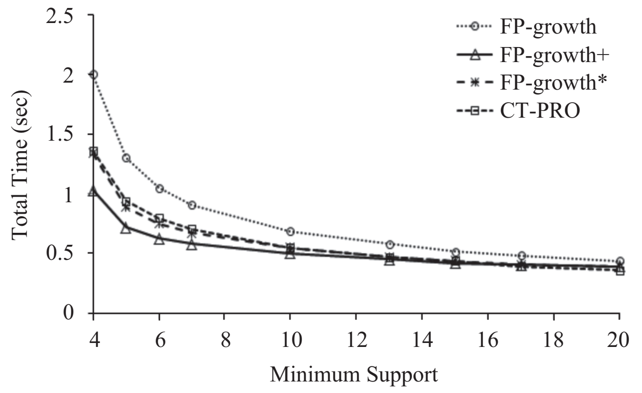


Fig. 8. Execution time on dataset *retail.*

IV. EXPERIMENTAL RESULTS

This section will present the performance comparison between the proposed algorithm and other algorithms. All experiments were conducted on Intel® Core (TM) i5 CPU 1.6 GHz and 4 GB memory using C programming language and they were all running on Microsoft windows 8 environments. To make the comparisons, the synthetic dataset *T10I4D100K* taken from IBM Almaden Research Centre and three real datasets c*onnect*, c*hess* and *retail* taken from FIMI repository website were adopted. The source codes for the algorithms used for comparison were downloaded from FIMI repository website. Fig. 8 to Fig. 10 show the comparisons of execution time among the proposed algorithm, FP-growth, FP-growth* and CT-PRO algorithms on the above datasets with different minimum supports. We used the same output formats like the original FP-growth algorithm without any optimizations on the output. The solid black line refers to the proposed algorithm whereas the dotted black lines refer to other algorithms. Fig. 8 shows that the computing time for the proposed algorithm in dataset *retail.* From Fig. 8, clearly the computing time for the proposed algorithm is less than the execution time for FP-growth. When the *min_sup* is set to 4, the proposed method gains 49% improvement (compared with the FP-growth method). This percentage decreases gradually when the value of *min_sup* increases until the *min_sup* is equal to 20. Clearly, the proposed algorithm outperforms the FP-growth method in all cases especially when the *min_sup* is small. In addition, FP-growth+ has the best running time compared with other algorithms, i.e., FP-growth* and CT-PRO algorithms when *min_sup* is from 2 to 8. If *min_sup* is larger than 8, FP-growth+ will have the same running time like FP-growth* and CT-PRO algorithms.

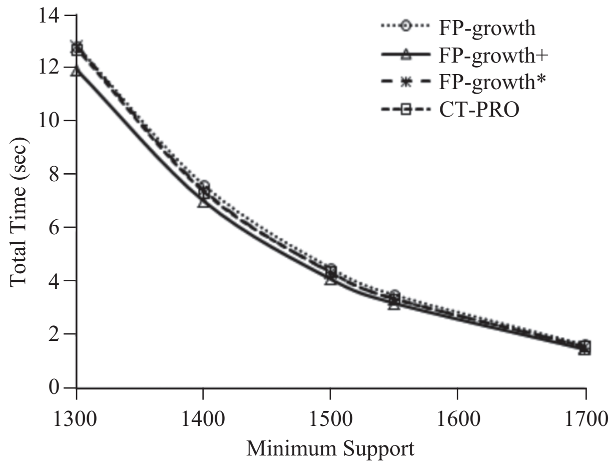


Fig. 9. Execution time on dataset *chess.*

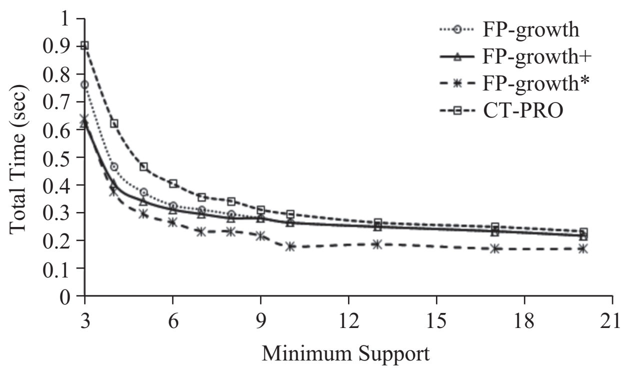


Fig. 10. Execution time using *T1014D100K* data set.

Fig. 9 shows the comparisons about CPU time consuming among the four algorithms in dataset *chess*. We see that FP-growth+ has the best running time. The running time improvements for all *min_sup* cases are 7%-10% than the FP-growth method.

Fig. 10 shows the comparison for the *T10I4D100K* dataset. The "FP-growth+" algorithm reduced CPU time up to 18%

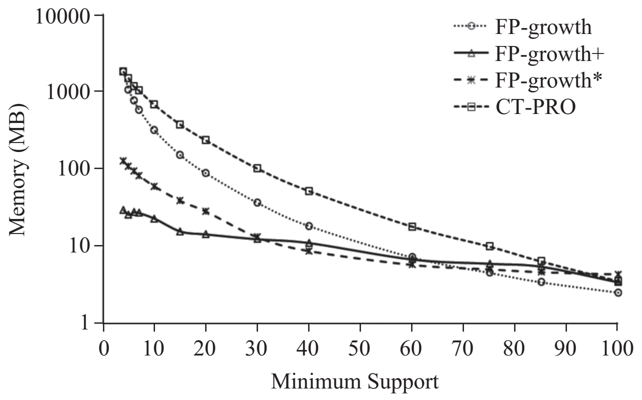


Fig. 11. Main memory usages on dataset "*retail*".

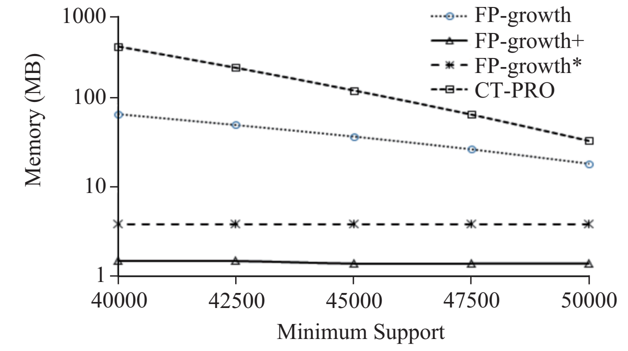


Fig. 12. Main memory usages on dataset "*Connect*".

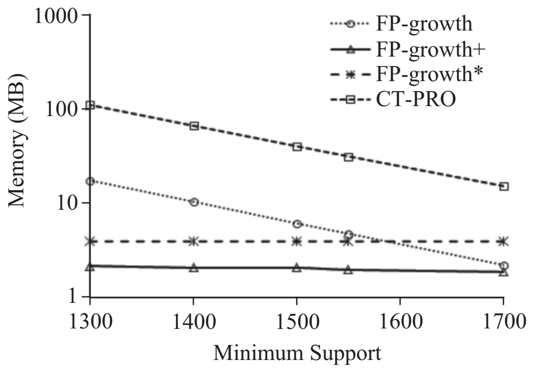


Fig. 13. Main memory usages on dataset "*chess*".

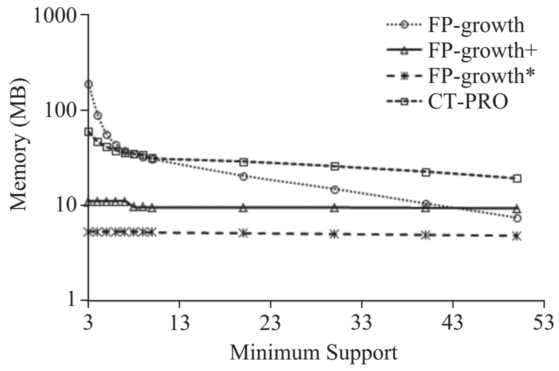


Fig. 14. Main memory usages on the dataset "*T10I4D100K*".

compared with FP-growth when the minimum support equals "4". After *min_sup* exceeds "8", the FP-growth and "FP-growth+" algorithms have similar running time. Comparing with the CT-PRO algorithm, our algorithm has the best running time for all cases of minimum supports. The FP-growth* has the best running time compared with other approaches.

In addition to time consuming, the memory usages of the four algorithms were also conducted. Their comparison results are shown in figures from Fig. 11 to Fig. 14. It is clear that the proposed algorithm is indeed much more effective in reducing the memory consumption than other three algorithms especially when the minimum support is small. In addition, the amount of memory saving increases clearly when the size of dataset increases. For all the tested datasets, the proposed method consumes less memory for all minimum support thresholds. If the minimum support is higher, the FP-growth consumes less memory because the number of conditional FP-trees to be created decreases gradually when the minimum support increases. It means that the main memory usage in FP-growth decreases gradually by increasing the minimum support. Our proposed approach uses arrays for traversing the FP-tree. The array structure can be frequently reused without allocating additional memory space by overwriting their contents many times. Compared with the FP-growth, the memory usage of our proposed method is not increased too much during the mining process. For the dataset *T10I4D100K*, the proposed method can reduce the memory consumption from 10% to 94%. Fig. 13 shows that the FP-growth+ is the best among all compared algorithms. The FP-growth+ method reduced up to 88% for *chess* dataset when the minimum support is "1300". In *retail* dataset, our algorithm is the best when the minimum support threshold is low. When the minimum support is larger than 100, the four algorithms consume less than 5 MB. The FP-growth+ consumes memory less than the FP-growth algorithm with the amounts 98%, 66% and 6% when the supports are "4", "30", and "60", respectively. The *Connect* dataset is the largest one among these datasets. Fig. 12 shows the memory usages when mining the *connect* dataset over all minimum support thresholds. For the FP-growth method, it consumes from 20 MB to 75 MB. This indicates that the proposed method saves more memory by increasing the dataset size, especially when the values of minimum support are low.

From the above experiments, we conclude that the proposed algorithm is indeed more effective and efficient to reduce the running time and memory usage than other three methods on most popular datasets in data mining. Also, the proposed algorithm uses less memory because it does not use linked list data structure for building conditional FP-trees. Our array technique is more efficient than using the tree technique because each node in the conditional FP-tree has many attributes associated with it. Additional advantage is the direct access for the data in the array compared with tree (usually takes constant time O (1)).

V. CONCLUSIONS

We have introduced a novel algorithm to find the frequent patterns based on the well-known algorithm FP-growth. The proposed method uses a new data structure based on the arrays to generate the frequent item sets instead of constructing conditional FP-trees. We applied our experiments in many synthetic and real datasets from FIMI repository website. Experimental results show the success of our algorithm to reduce the running time and the memory usage because it uses a set of arrays for traversing the FP-tree instead of recursively generate mass number of conditional FP-trees.

REFERENCES

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.

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.

Bashir, S. and A. Baig (2006). Ramp: High Performance Frequent Itemset Mining with Efficient Bit-vector Projection Technique. Advances in Knowledge Discovery and Data Mining, 3918, 504-50.

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.

Han, J., J. Pei 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.

Leung, C., M. Mateo and D. Brajczuk (2008). A Tree-Based Approach for Frequent Pattern Mining from Uncertain Data. Advances in Knowledge Discovery and Data Mining Lecture Notes in Computer Science 5012, 653-661.

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.

Liu, G., H. Lu, Y. Xu and J. X. Yu (2003). Ascending frequency ordered prefix-tree: efficient mining of frequent patterns. Proceedings of Eighth International Conference on Database systems for advanced applications, 65-72.

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.

Schlegel, B., R. Gemulla and W. Lehner (2011). Memory-Efficient Frequent-Itemset Mining", EDBT/ICDT '11 Proceedings of the 14th International Conference on Extending Database Technology, New York, NY, USA, 461-472.

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

Sucahyo, Y. G. and R. P. Gopalan (2004). CT-PRO: A Bottom-Up Non Recursive Frequent Itemset Mining Algorithm Using Compressed FP-Tree Data Structure. In FIMI. 4, 212-223.

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

Zaki, M. J. (2004). Mining non-redundant association rules. Data mining and knowledge discovery 9(3), 223-248.

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.