

Discovering of Frequent Itemsets with CP-mine Algorithm

Nuansri Denwattana and Yutthana Treewai

Abstract—Efficient algorithms to discover frequent patterns are crucial in data mining research. Several effective data structures, such as two-dimensional arrays, graphs, trees, and tries have been proposed to collect candidate and frequent itemsets. It seems as the tree structure is most extractive to storing itemsets. The outstanding tree has been proposed so far is called FP-tree which is a prefix-tree structure. In this paper, MFP-tree data structure is introduced. It is a modification of FP-tree. A CP-mine algorithm is also proposed in order to mine frequent itemsets from the MFP-tree by using pruning tree and a marked value technique. We have conducted several experiments based on both synthetic and real world datasets with different parameters. The experiments are illustrated the comparison between CP-mine and FP-growth algorithms.

Keywords—CP-mine, FP-tree, Frequent Itemsets, MFP-tree.

I. INTRODUCTION

A procedure so called mining frequent itemsets is a core step of association rules discovering introduced in [1]. By using basket data in supermarket, we can gain valuable information regarding to a relationship of items or products buying together in the form of association rules. Such information can be used to make decisions, for example designing weekly catalog, shelving products in supermarket, customer segmentation, and cross-marketing etc.

Discovering association rules can be performed in two steps (1) find all frequent itemsets and (2) determine association rules by using those frequent itemsets generated from the first step. In a basket database, if we know the support of all frequent itemsets, the generation of association rules is straightforward. For example, if a number of frequent itemsets is n , all 2^n nonempty subsets of those frequent itemsets must be generated. Most existing algorithms, therefore, are focused on mining frequent itemsets. Several algorithms have been proposed to mine frequent itemsets. A classical algorithm is Apriori introduced in [1]. Variations of Apriori were proposed so far such as a hash-based algorithm [12], a dynamic itemset counting technique (DIC) [3], a Partition algorithm [15]. In addition, many research have been developed algorithms using tree structure, such as ITL-mine [6], TreeITL-mine [7], CT-ITL[14], CT-mine [8], H-mine [13] and FP-growth [10]. Lately, an idea of online generation for mining association rules has been introduced in [4, 5, 9, 11].

In this paper, we propose a new algorithm named CP-mine for mining the set of all frequent itemsets directly from

the Marked Frequent Pattern Tree (MFP-tree). A motivation of the idea is from the FP-tree [10]. By using the MFP-tree, the CP-mine algorithm examines frequent itemsets. The main idea of CP-mine algorithm is to improve an efficiency of mining frequent patterns by using pruning tree and marked value techniques.

The organization of this paper is as follows: In Section 2, we put our algorithm in the context of related works, especially the FP-tree structure and the FP-growth algorithm. In Section 3, we give a detail of MFP-tree data structure and the CP-mine algorithm. Several experiments and analysis are discussed in the Section 4. We end with our conclusion in Section 5.

II. RELATED WORK

In this section, we shall briefly discuss of related works starting from an Apriori algorithm to its variations. Since the motivation of our contribution is from the concept of FP-tree, we then discuss the concept of FP-tree structure and FP-growth algorithm.

A. Apriori and Its Variations

Most existing algorithms are related to the Apriori algorithm [2]. The Apriori exploits an observation that all subsets of frequent itemsets are frequent themselves. It is a multi-pass algorithm, where in the k^{th} pass all frequent itemsets of cardinality k are computed. That is Apriori needs up to $c+1$ scans of the database where c is the maximal cardinality of a frequent itemset.

In [15], a Partition algorithm is introduced by scanning only 2 database scans. The general idea is to divide the database into partitions such that each partition fits into main memory. In the first database pass, each partition is loaded into memory and all frequent itemsets with respect to that partition are computed using Apriori. In the next step, all resulting sets of frequent itemsets are merged to find a superset of all frequent itemsets. In the second pass, the actual support of each set in the superset is computed. After removing all small itemsets, the Partition algorithm produces the set of all frequent itemsets.

In contrast to Apriori, the Dynamic Itemset Counting (DIC) algorithm [3] counts itemsets of different cardinality simultaneously. The transaction sequence is partition into blocks. The itemsets are stored in a lattice which is initialized by all singleton sets. While a block is scanned, the number of occurrences of each itemsets in the lattice is adjusted. After a block is processed, an itemsets is added to the lattice is and only if all its subsets are potentially frequent. At the end of the sequence, the algorithm rewinds to the beginning. It terminates when the count of each itemsets in the lattice is determined. Thus after a finite

N. D. and Y. T. are working in the Department of Computer Science, Faculty of Science, Burapha University, Muang, Chonburi 20131 Thailand (phone: 66-38-745900 x. 4116; fax: 66-38-390046; e-mail: {nuansri, 45031922}@buu.ac.th).

numbers of scans, the lattice contains a superset of all frequent itemsets and their frequencies. For suitable block sizes, DIC requires fewer scans than Apriori.

Several algorithms based on tree data structure were proposed to update a previously computed set of frequent itemsets such as Compress Prefixed Trees [8 and 14]. Gopalan et al. has proposed in [6 and 7] an array based data structure called Item-Trans Link. The ITL consists of ItemTable and TransLink. All frequent 1-itemsets and their supports are kept in the ItemTable. TransLink represents each transaction of the database that contains the items of ItemTable and corresponding link of next item occurrence in another transaction. After ITL are constructed, the ITL-mine or TreeITL-mine algorithms are applied to mine frequent itemsets. The improvement of the previous algorithms introduced in [8, and 14] by using compress prefixed tree. This tree can reduce a number of nodes from 2^n to 2^{n-1} nodes by grouping the identical subtrees in the same path and keeping more relevance information. When each transaction is read, corresponding counter for each node will be increased by one. After the compress prefixed tree is created, t_i will be transformed into Compact ITL and the CT-mine algorithm will be applied to find frequent itemsets.

B. FP-tree Structure and FP-growth Algorithm

A FP-growth algorithm is proposed in [10] to solve a bottleneck of the Apriori algorithm. The algorithm used a frequent pattern tree or FP-tree which is an extended prefix-tree structure for storing crucial and compressed information about frequent patterns as shown in Fig 1. The FP-growth algorithm based on FP-tree mines the completed set of frequent itemsets by pattern fragment growth.

```

Algorithm: CreateFP-tree
Input : A database  $DB$  and a minimum support threshold  $\xi$ 
Output : FP-tree

(1) Procedure CreateFP-tree
(2) scan the  $DB$  once to collect the frequent items and their support
    then sort in support descending and create the header table
(3) FP-tree is null
(4) for each transaction  $t_i$  in  $DB$ 
(5)     select and sort the frequent items in  $t_i$  according to the order in
        the header table
(6)     call InsertTree(FP-tree,  $t_i$ )
(7) end for

(8) Procedure InsertTree( $root$ ,  $tran$ )
(9) for each item  $k_i$  in  $tran$ 
(10)  if  $root$  has a child  $N$  that  $N.item\_name = k_i$ 
(11)    increment  $N$ 's count by 1
(12)     $root = N$ 
(13)  else
(14)    create the new node  $k_i$  is the child of  $root$ 
(15)    link the header table to node
(16)  end if
(17) end for

```

Fig. 1 CreateFP-tree Algorithm

Only frequent 1-items are stored in each node of the tree. The FP-tree is applied to restrict generation of large number of candidate sets. Unlike Apriori algorithm that use generation-and-test approach. The FP-growth algorithm is adopted to pattern growth approach to avoid scanning

database for every level of frequent and handle a huge number of candidate sets. By using this approach, there is no need to generate candidate itemsets. The algorithm begins with frequent 1-item that is kept in FP-tree to perform recursive mining as shown in Fig 2. The search technique is a partitioning-based, divide-and-conquer method to increase efficiency of mining.

It can be noticed that only two database scans are needed in FP-growth algorithm. In addition, when mining longer frequent itemsets on large database, the FP-growth algorithm is significantly outperforms the Apriori algorithm. However, when only a small portion of the candidate itemsets becomes frequent itemsets, the FP-growth algorithm is worse than the Apriori algorithm due to the costly FP-tree generation.

```

Algorithm: FP-growth
Input : FP-tree and a minimum support threshold  $\xi$ 
Output : The set of all frequent itemsets

(1) Procedure FP-growth( $Tree$ ,  $\alpha$ )
(2) if  $Tree$  contains a single path  $P$ 
(3)   for each combination of nodes (denoted as  $\beta$ ) in the path  $P$ 
(4)     generate itemset  $\beta \cup \alpha$  with support = minimum support
        of nodes in  $\beta$ 
(5)   end for
(6) else
(7)   for each  $a_i$  in the header table of  $Tree$ 
(8)     generate itemset  $\beta = a_i \cup \alpha$  with support =  $a_i$ .support
(9)     construct  $\beta$ 's conditional pattern base and conditional
        FP-tree  $Tree_\beta$ 
(10)  end for
(11)  if  $Tree_\beta \neq \emptyset$ 
(12)    call FP-growth( $Tree_\beta$ ,  $\beta$ )
(13)  end if
(14) end if

```

Fig. 2 FP-growth Algorithm

III. CP-MINE STRUCTURE AND ALGORITHM

In this section, we shall explain what CP-mine data structure looks like, how the CP-mine structure is constructed and how frequent itemsets are mined from the CP-mine structure.

A. CP-mine Structure

The CP-mine structure consists of two parts: a header table and the MFP-tree.

1. A header table is the list of all frequent 1-itemsets ordered by their frequency in descending order. It contains the item name and the link to the corresponding node in MFP-tree. Its structure is shown in Fig 3(b).

2. The Marked Frequent Pattern Tree (MFP-tree) is a prefix tree such that each node represents the following information: frequent 1-item, its support, marked value, and pointers linking to successive nodes. The MFP-tree structure is shown in Fig 3(c).

B. CP-mine Data Structure Construction

A construction of CP-mine is performed in the following steps. The transaction database is first scanned in order to explore all frequent 1-items and their supports. Secondly, a header table is created by sorting these itemsets regarding to

their frequency in descending order. After the header table is created, the database is read again in order to construct the MFP-tree. During the second database scan, only frequent 1-items in each transaction are sequentially put as one path in the tree corresponding to the header table together with the increment of their supports. In addition, the marked value is initially set up to zero whenever a new node in the tree is created. The algorithm to create the MFP-tree is shown in Fig 4 and the explanation of the algorithm is given in Example 1.

TABLE I
THE TRANSACTION DATABASE

TID	Items	Frequent Items
100	a, c, d, f, g, i, m, p	f, c, a, m, p
200	a, b, c, f, l, m, o	f, c, a, b, m
300	b, f, h, j, o	f, b
400	b, c, k, p, s	c, b, p
500	a, c, e, f, l, m, n, p	f, c, a, m, p

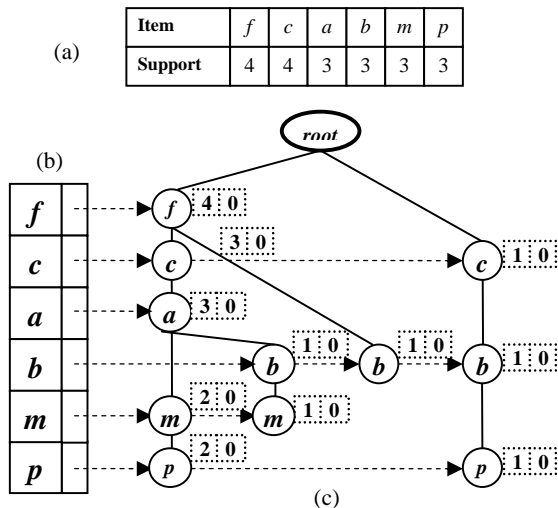


Fig. 3 CP-Mine Data Structure (a) The Frequent 1-Itemsets, (b) The Header Table, (c) The Marked Frequent Pattern Tree

Example 1. Let the Table I be the database used to illustrate our proposed algorithm with 60% (3 transactions) of a minimum support threshold. The third column of the table contains frequent 1-items in each transaction in descending ordered of the support.

Firstly, all transactions in *DB* are read to identify a set of frequent 1-items and their supports as shown in Fig 3(a). This step is expressed in line (2) of Fig 4.

Then, each transaction in the database is secondly scanned. By reading the first transaction, a list of frequent 1-items, $\{f(1), c(1), a(1), m(1), p(1)\}$ (the number in the parenthesis indicates the support), is derived, in which items are ordered in frequency descending order. This step is expressed in line (4) and line (5). The ordering of itemsets is importance because every path of the tree is constructed followed in this order. The list of these items is inserted as a first branch from the root and a support of each item together

with a marked value are set up to 1 and 0 consecutively. Corresponding pointers in the header table are then linked to the node in the tree. These steps are expressed in a procedure InsertTree of Fig 4.

Next, the list of $\{f(1), c(1), a(1), b(1), m(1)\}$, is derived when the second transaction is read. These items are inserted to the tree by a common prefix $\{f, c, a\}$ sharing with existing path $\{f, c, a, m, p\}$. The support of each shared node is increased by 1 and the supports of new nodes are set to 1. The marked values of new nodes, $\{b, m\}$, are set to 0. In addition, node *b* in the tree is linked from item *b* in the header table whereas node *m*, is linked as a sibling node of existing node *m*. We repeatedly apply these steps until the last transaction of the database is performed. Finally, the complete MFP-tree is illustrated in Fig 3(c).

Notice that the construction of MFP-tree and FP-tree is similar, except in the MFP-tree, the marked value is introduced and it will be explored in the next subsection.

Algorithm: CreateMFP-tree
Input : A database *DB* and a minimum support threshold ξ
Output : MFP-tree

- (1) Procedure **CreateMFP-tree**
- (2) scan the *DB* once to collect the frequent items and their support then sort in support descending and create the header table
- (3) **MFP-tree** is null
- (4) **for each** transaction t_i in *DB*
- (5) select and sort the frequent items in t_i according to the order in the header table
- (6) call **InsertTree**(MFP-tree, t_i)
- (7) **end for**
- (8) Procedure **InsertTree**(*root*, *tran*)
- (9) **for each** item k_i in *tran*
- (10) **if** *root* has a child *N* that $N.item_name = k_i$
- (11) increment *N*'s count by 1
- (12) *root* = *N*
- (13) **else**
- (14) create the new node k_i is the child of *root*
- (15) set node **marked value** to zero
- (16) link the header table to node
- (17) **end if**
- (18) **end for**

Fig. 4 CreateMFP-tree Algorithm

C. CP-mine Algorithm

In this subsection, we shall discuss how to mine frequent itemsets from the MFP-tree. We propose a new algorithm, called CP-mine and the detail is given in Fig 5. The algorithm discovers entire frequent itemsets from the tree. There are several different points between FP-growth and CP-mine algorithms. First, CP-mine algorithm does not need to recursively construct new trees during mining frequent itemsets. The CP-mine algorithm determines frequent itemsets from only one tree, that is MFP-tree, whereas the FP-tree has to construct many new (sub)trees. Second, although the MFP-tree is traversed recursively, the algorithm works in the manner of traversing only in the particular subtrees. We use a marked value and a pruning tree to trim irrelevance subtrees and infrequent itemsets. Finally, the CP-mine algorithm mines frequent itemsets starting from the $(n-1)^{th}$ item in the header table and assumes that this item is a

new root. We then examine every path under this root to determine frequent itemsets. Next, the $(n-2)^{th}$ item in the header table is considered. Again, we assume that this node is a new root and every path under this root is recursively determined for frequent itemsets. These steps are repeated until the first item in the header table is performed. The Example 2 is a demonstration of the CP-mine algorithm.

Example 2. Let the MFP-tree from the Example 1 is an input of CP-mine algorithm.

The algorithm starts by considering item m in the header table. We assume that item m is new root. There are 2 subtrees. The first subtree has one child node, that is node p , whereas the second subtree has no child. Therefore, only one path, i.e. $\{p(2)\}$, is determined for the support. This step is expressed in line (3) of Fig 5. As the support of the 2-itemset mp is less than the minimum support, i.e. 3, the itemset mp is then put in the pruning tree as successive nodes under the root shown in Fig 6(a). This step is expressed in line (10) and line (11).

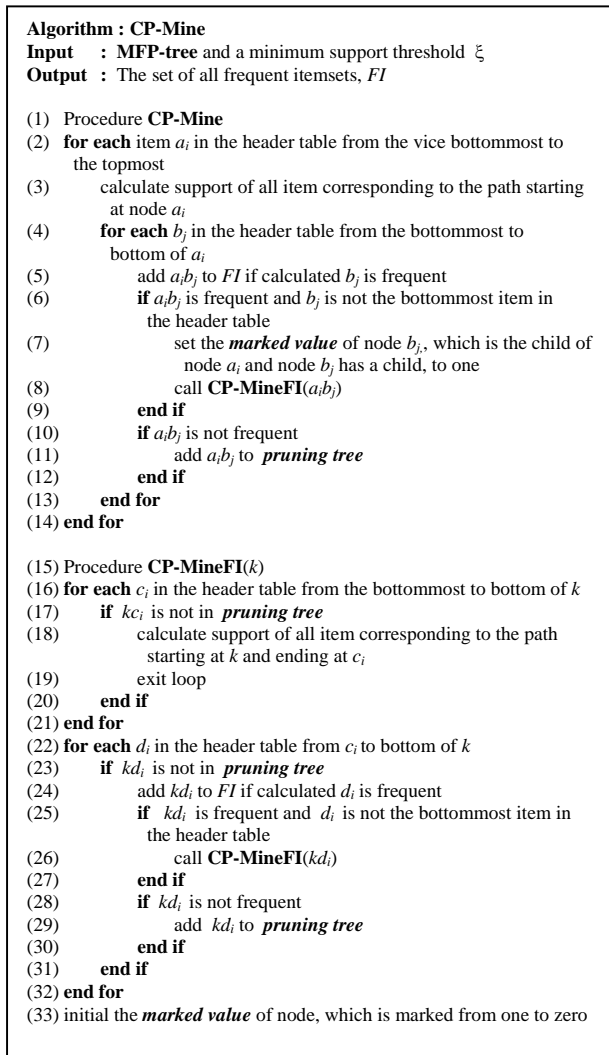


Fig. 5 CP-Mine Algorithm

An item b in the header table is next considered. Again we assume that node b is a root. As there are three nodes labeled b in the tree, the algorithm examines three subtrees. Only two paths are examined, that is $\{m(1)\}$ and $\{p(1)\}$. Since the supports of itemsets bp and bm is less than 3, these itemsets are put in the pruning tree as shown in Fig 6(b).

Next, the item a from the header table is read. There are two paths under root a , $\{m(2), p(2)\}$ and $\{b(1), m(1)\}$. As the support of ap is 2, we then put it in the pruning tree as illustrated in Fig 6(c). In addition, we put itemset am in a set of frequent itemsets because its support is equal to a minimum support. This step is expressed in line (5) of Fig 5. At this point, a marked value of node m , which is a parent of node p , is set to 1 as shown in Fig 7(a). This step is in line (7) of Fig 5. Next, the algorithm recursively works to determine itemsets with a prefix “ am ” as expressed in line (8) of Fig 5. An itemset amp is considered whether it is able to be a candidate itemset. In this step, subsets of amp are traversed in the pruning tree. Starting from the last item of amp , i.e. item p , as itemset pm is a path in the pruning tree, the itemset amp cannot be a candidate itemset. Therefore, itemset amp is not a frequent. The pruning step is expressed in line (17) of Fig 5. At the end of this stage, marked value of every node which is marked to 1 is initialized to 0 as expressed in line (33).

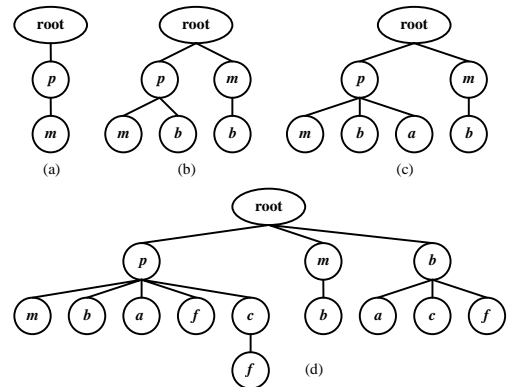


Fig. 6 The Pruning Tree

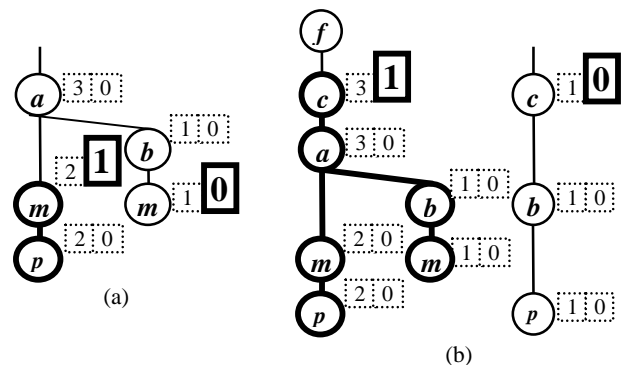


Fig. 7 The Marked Value indicate the Subtrees in MFP-tree

The algorithm continuously performs until the first item, f , is considered. The final pruning tree is shown in Fig 6(d). To illustrate that how marked value is performed in order to

trim some subtrees, let consider itemset fc when the new root is f . As itemsets fc is frequent (its support is 3), the algorithm needs to recursively perform. Therefore, the marked value of item c which is a child node of root f is set to 1. We are considering a prefix c in the tree. There are two subtrees of node c . The algorithm considers only node c whose marked value is 1. Another node c (marked value is 0) can be ignored because it is not a child of node f . Notice that instead of traversing two subtrees starting from item c , the algorithm can only examine in a particular subtree.

Finally, the complete set of frequent itemsets is $\{p(3), m(3), b(3), a(3), am(3), c(4), cp(3), cm(3), ca(3), cam(3), fm(3), fa(3), fam(3), fc(3), fcm(3), fca(3), fcam(3)\}$.

IV. PERFORMANCE STUDY

In this section, we shall start by discussing the datasets and features of our experiments. We then analyze the experimental results of real world datasets. We end up with evaluation of the result from synthetic datasets.

A. Datasets and features

In this section, the performance of CP-mine algorithm is evaluated by comparing with FP-growth. All experiments are performed on a 3.0GHz Pentium IV PC machine with 256 MB main memory and 40 GB hard disk, running MS Windows XP Professional. Both algorithms are implemented by using MS Visual C++ 6.0. All reports of runtime of **CP-mine** algorithm including both constructing **MFP-tree** and mining the complete set of frequent itemsets. The runtime only includes the disk reading time (scan datasets) and CPU time, but excludes disk writing (frequent itemsets output) to reduce the influence of relatively slow speed of disk writing.

We perform experiments on two kinds of datasets, synthetic and real world datasets. The synthetic datasets (T10I4D100K and T40I10D100K) are from Quest IBM as performed in [1] whereas the real world datasets are Pumsb and Pumsb* from Itemset Mining Dataset Repository (<http://fimi.cs.helsinki.fi/data>).

In addition, we divided the experiments into three phases. The first phase is shown the runtime of both algorithms with different datasets when the minimum support is varied. In the second part of our experiment, we compared the memory usage of both algorithms when the minimum support is varied. In the last phase, the runtime of both algorithms are illustrated when the number of transactions is varied.

The Pumsb is a census data that contains 49,046 transactions. There are 74 items in each transaction (from 2,112 different items). It is a very dense dataset in that the number of frequent itemsets grows from 172 to 20,527 and 672,390 when support threshold reduces from 95% to 85% and 75%. The Pumsb* is a relatively less dense dataset than Pumsb. That is in Pumsb*, There are only 50 items in each transaction (from 2,087 different items). Similar to the Pumsb, the number of frequent itemsets is increased when support threshold is decreased (as shown in Fig 10).

The T40I10D100K dataset contains 100,000 transactions and each transaction has up to 40 items. There are 1,000 different items in the dataset and average longest potentially

frequent itemset is with 10 items. It is a less dense dataset. Similarly to T40I10D100K dataset, the T10I4D100K dataset consists of 100,000 transactions and 1,000 different items. Each transaction has up to 10 items and average longest potentially frequent itemset is with 4 items.

B. Experiments base on Real World Datasets

In this subsection, two graphs are shown by using Pumsb and Pumsb* datasets.

The Fig 8 shows the runtime of CP-mine and FP-growth algorithms on Pumsb. Clearly, the CP-mine algorithm is better than the FP-growth algorithm and the gap become larger as the support threshold goes lower. The main reason is CP-mine algorithm creates only tree only time whereas FP-growth needs to construct several trees. Another reason is CP-mine traverses only in particular subtrees.

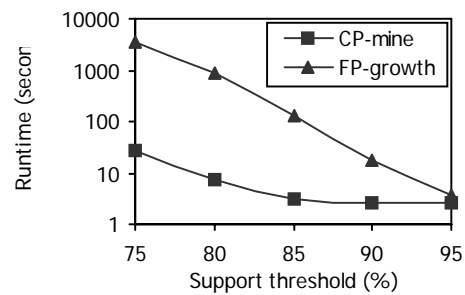


Fig. 8 Runtime on Pumsb

Fig 9 shows the memory usage based on Pumsb dataset. Notice that the CP-mine algorithm consumes less memory resource than FP-growth. It is because CP-mine does not need to create new subtrees during recursively mining frequent itemsets.

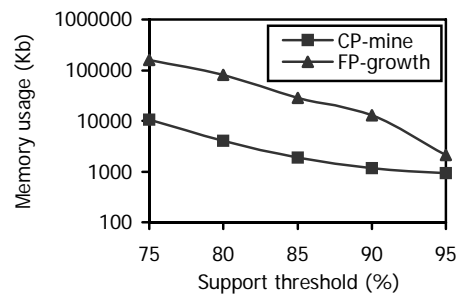


Fig. 9 Memory usage on Pumsb

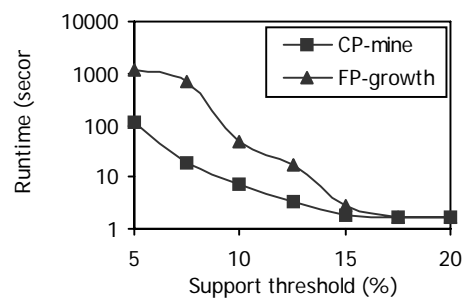


Fig. 10 Runtime on Pumsb*

In Fig 10, based on the Pumsb* dataset, the runtime of CP-mine algorithm is better than FP-growth. Notice that the value of minimum support threshold is lower, the running time of CP-mine algorithm gets closure to FP-growth algorithm. This is because when the minimum support is low, a few itemsets can be pruned.

C. Experiments base on Synthetic Datasets

The graphs in Fig 11 show that CP-mine algorithm is better than FP-growth in every support threshold. The pruning tree of CP-mine algorithm can prune many of candidate itemsets so that the time to traverse the tree is reduce.

In Fig 12, the memory usage of T40I10D100K is shown. Similar to the real world datasets in subsection B, the CP-mine algorithm consumes less memory than the FP-growth algorithm with the high support threshold. However, when the support threshold is low, the memory usage of CP-mine algorithm is closed to the FP-growth algorithm. This is because when the value of minimum support is getting lower, the pruning tree of CP-mine algorithm becomes bigger.

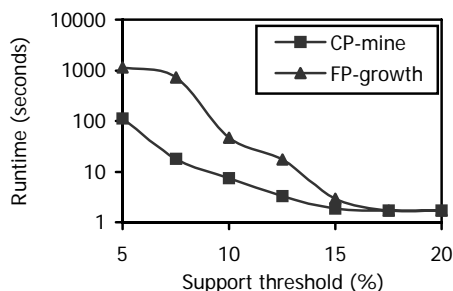


Fig. 11 Runtime on T40I10D100K

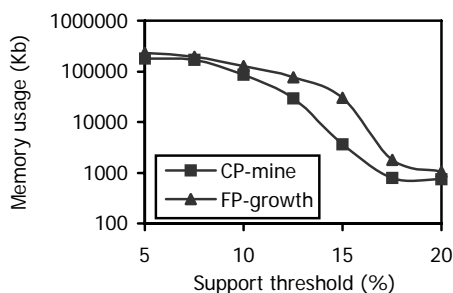


Fig. 12 Memory usage on T40I10D100K

In Fig 13, it confirms that when the transaction length is long (from 10 to 40), the running of both algorithms is also long (comparing between Fig 11 and Fig 13 when the minimum support is 5). The runtime of CP-mine algorithm is better than FP-growth because of in the sparse dataset, many candidate itemsets are pruned.

The Fig 14 shows the performance of both algorithms by using on T10I4 datasets and the minimum support is set to 2.5%. The number of transactions is increased from 100K to 1500K whereas a number of frequent itemsets is fixed. The CP-mine algorithm is better than the FP-growth algorithm in

every size of datasets and the gap of runtime between both algorithms is stability. This is because the runtime of both algorithms is relied to the number of transactions.

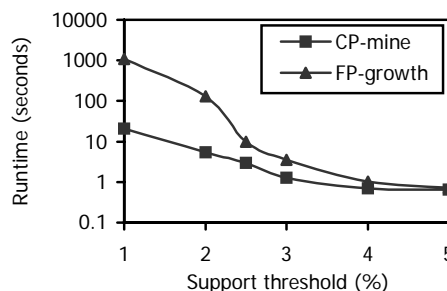


Fig. 13 Runtime on T10I4D100K

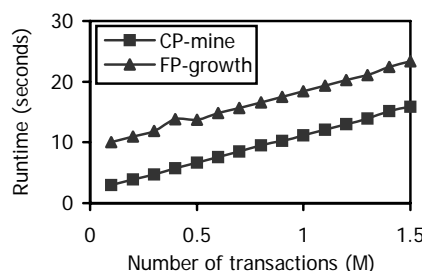


Fig. 14 Runtime on T10I4 with D100K through D1500K at the minimum support threshold of 2.5%

V. CONCLUSIONS

This paper proposes a new technique to improve an efficiency of a procedure so called mining frequent itemsets. A tree structure, called MFP-tree, and a new algorithm, called CP-mine, are introduced. The MFP-tree is constructed in a similar way as FP-tree. Based on MFP-tree, the CP-mine algorithm determines frequent itemsets. In the CP-mine algorithm, several new techniques are applied such as marked value and a pruning tree. By using such techniques, some steps of mining frequent itemsets are trimmed. We have performed varieties of experiments based on real world and synthetic datasets. The results showed that our proposed algorithm is able to overcome the FP-growth algorithm.

REFERENCES

- [1] R. Agrawal, T. Imielinski, and A. Swami, Mining Association Rules between Sets of Items in Large Databases, *Proc. of ACM SIGMOD Conf.*, Washington DC, 1993.
- [2] R. Agrawal and R. Srikant, Fast Algorithms for Mining Association Rules, *Proc. Very Large Databases Conf.*, Dept. 1994.
- [3] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. *In Proceedings of ACM SIGMOD*, page 225-264, 1997.
- [4] Charu C. Aggarwal and Philip S. Yu. Online Generation of Associations Rules. *In 14th International Conference on Data Engineering, ICDE*, Feb, 1998.
- [5] G. Dong, J. Han, L. Lakshmanan, J. Pei, H. Wang, Philip S. Yu, *SIGMOD Workshop on Management and Processing of Data streams*, 2003.
- [6] R. Gopalan, Y.G. Sucahyo, ITL-Mine: Mining Frequent Itemsets More Efficiently, *in Proceedings of 2002 International Conference of Fuzzy Systems and Knowledge Discovery*, Singapore, 2002.

- [7] R. Gopalan, Y.G. Sucahyo, TreeITL-Mine: Mining Frequent Itemsets Using Pattern Growth, Tid Intersection and Prefix Tree, in *Proceeding of 15th Australian Joint Conference on Artificial Intelligence*, Canberra, Australia, LNAL 2557, Springer, 2002.
- [8] R. Gopalan, Y.G. Sucahyo, Fast Frequent Itemset Mining using Compressed Data Representation, in *Proceedings of IASTED International Conference on Databases and Applications (DBA'2003)*, Innsbruck, Austria, Feb 10-13, 2003.
- [9] C. Hidber, Online Association Rule Mining, In *Proc. Of the 1999 ACM SIGMOD international conference on Management of data*, page 145-156, May 31-June 03, 99 Philadelphia, Pennsylvania, USA.
- [10] J. Han, J. Pei, and Y. Yin, Mining Frequent Patterns without Candidate Generation. *Proc. of ACM SIGMOD Conf.*, Dallas, TX, 2000.
- [11] K.K. Loo, I. Tong, B. Kao, D. Cheung, Online Algorithms for Mining Inter-Stream Associations From Large Sensor Networks, In *springer*, May, 2005.
- [12] J. S. Park, M. Chan and P. S. Yu. An Effective Hased-Based Algorithm for Mining Association Rules. In *Proceedings of ACM SIGMOD*, pages 175-186. ACM, May 1995.
- [13] J. Pei J.Han H. Lu S. Nishio S. Tang and D. Yang. H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases. In *ICDM01 International Conference on Data Mining*. ICDM, 2001.
- [14] Y.G. Sucahyo, R. Gopalan, CT-ITL: Efficient Frequent Item Set Mining using a Compressed Prefix Tree with Pattern Growth, in *Proceedings of 14th Australasian Database Conference*, Adelaide, Australia, 2003.
- [15] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21st VLDB Conference*, Zurich, Switzerland, 1995.