EFFICIENTLY MINING FREQUENT ITEMSETS WITH COMPACT FP-TREE

QIN Liang-Xi [1, 2, 3], LUO Ping [1, 2] and SHI Zhong-Zhi [1]

[1] *(Key Lab of Intelligent Information Processing, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080)*

[2] *(Graduate School of Chinese Academy of Sciences, Beijing 100039)*

[3] *(College of Computer and Information Engineering, Guangxi University, Nanning 530004)*

e-mail:{qinlx, luop, shizz}@ics.ict.ac.cn

Abstract: FP-growth algorithm is an efficient algorithm for mining frequent patterns. It scans database only twice and does not need to generate and test the candidate sets that is quite time consuming. The efficiency of the FP-growth algorithm outperforms previously developed algorithms. But, it must recursively generate huge number of conditional FP-trees that requires much more memory and costs more time.

In this paper, we present an algorithm, CFPmine, that is inspired by several previous works. CFPmine algorithm combines several advantages of existing techniques. One is using constrained subtrees of a compact FP-tree to mine frequent pattern, so that it is doesn't need to construct conditional FP-trees in the mining process. Second is using an array-based technique to reduce the traverse time to the CFP-tree. And an unified memeory management is also implemented in the algorithm. The experimental evaluation shows that CFPmine algorithm is a high performance algorithm. It outperforms Apriori, Eclat and FP-growth and requires less memory than FP-growth.

Key words: association rule; frequent patterns; compact FP-tree

1. INTRODUCTION

Since the association rule mining problem was introduced in [2] by Agrawal et al., finding frequent patterns is always a crucial step in association rules mining. In addition to association rules, frequent pattern is also used in mining correlations, causality, sequential patterns, episodes, multidimensional patterns, maximal patterns, partial periodicity, emerging patterns and many other important data mining tasks. So, the efficiency of a frequent pattern mining approach has a great influence to the performance of the algorithms used in these data mining tasks. The following is a formal statement of frequent patterns (or itemsets) mining problem:

Let $I = \{i_1, i_2, ..., i_m\}$ be a set of literals, called items. Let database D be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. Each transaction has a unique identifier, TID. Let X be a set of items. A transaction T is said to contain X if and only if $X \subseteq T$. The support of a itemset X is the probability of transactions in database D that contain X. X is frequent if the support of X is no less than a user defined support threshold. We are interested in finding the complete frequent itemsets.

Agrawal, et al presented Apriori algorithm in [3], in which a prior knowledge about frequent itemsets was used. The prior knowledge is that if an itemset is not frequent, then all of its supersets can never be frequent. Most of previous studies adopted an Apriori-like candidate set generation-and-test approach, such as DHP algorithm[8], Partition algorithm[9], Sampling algorithm[10] and DIC algorithm[4]. The Apriori-like algorithms suffer two problems: (1) It is costly to generate huge number of candidate sets for long patterns. (2) It is quite time consuming to repeatedly scan the database to count the support of candidate itemsets to decide which one is a frequent pattern.

Zaki et al. presented Eclat Algorithm in [11], in which they use an itemset clustering technique, efficient lattice traverse technique and vertical database layout in the mining process. Agarwal et al. proposed TreeProjection algorithm in [1], in which they represent itemsets as nodes of a lexicographic tree and use matrices to count the support of frequent itemsets. Han, et al, developed the FP-growth algorithm [7] that is based on frequent pattern tree. This algorithm avoids time consuming operations such as repeated database scans and generation of candidate set. The efficiency of the FP-growth algorithm is about an order of magnitude faster than the Apriori algorithm and outperforms TreeProjection algorithm. But, there still exist some aspects in FP-growth algorithm that can be improved. For example, it needs to recursively generate huge number of conditional FP-trees that consumes much more memory and more time. It has appeared several improved FP-

growth algorithms based on original one. Fan and Li [5] presented a constrained subtree based approach to avoid the generation of huge number of conditional FP-trees recursively in the mining process. They also reduced the fields in each FP-tree node. Their approach has better time and space scalability than FP-growth algorithm. Grahne and Zhu [6] proposed an array-based technique to reduce the time cost in FP-tree traverse. They also implemented their own memory management for allocating and deallocating tree nodes.

In this work, we adopts several advanced techniques which is inspired by [5] and [6]. One is using constrained subtrees of a compact FP-tree(CFP-tree) in the mining process, so that it doesn't need to construct conditional FP-trees. Second is using an array-based technique to reduce the traverse time to CFP-tree.

The remaining of the paper is organized as follows. The detailed description of CFP-tree based approach is given in Section 2. The CFP-tree mining algorithm is described in Section 3. In Section 4, we will give the experimental results. And in Section 5, we will give our conclusion and the idea about our future work.

2. COMPACT FP-TREE AND ITS CONSTRUCTION

The FP-growth algorithm [7] uses a data structure called the FP-tree. The FP-tree is a compact representation of frequency information in a transaction database. There are 6 fields in a FP-tree node. They are item-name, count, parent-link, child-link, sibling-link and next-link (a pointer to next node that has same item-name). However, child-link and sibling-link are only used in the FP-tree constructing process, parent-link and next-link are only used in the mining process. So we can reduce the number of fields by joining child-link with parent-link as cp-link which is first pointing to its child-node and after construction pointing to its parent node, and joining sibling-link with next-link as sn-link which is first pointing to its sibling-node and finally pointing to next node.

The compact FP-tree (CFP-tree) has similar structure as FP-tree. They also have several differences:

(1) Each node in CFP-tree has 4 fields, item-no (which is the sequential number of an item in frequent 1-itemsets according frequency descending order), count, cp-link and sn-link. Therefore, CFP-tree requires only 2/3 memory spaces of FP-tree.

(2) FP-tree is bi-directional, but CFP-tree is single directional. After the construction, CFP-tree only exists paths from leaves to the root.

The CFP-tree is constructed as follows.

Algorithm 1. (CFP-tree construction)

Input: A transaction database DB and support threshold minsup.

Output: Its CFP-tree.

Method: CFP-tree is constructed in following steps.

(1) Scan the transaction database DB once. Collect the set of frequent items F and their supports. Sort F in support descending order as L, the list of frequent items.

(2) Create the root of an FP-tree, T, and label it as "null". For each transaction in DB do the following.
Select the frequent items and replace them with their order in L, and sort them as Is. Let the sorted Is be [p|P], where p is the first element and P is the remaining list. Call insert tree([p|P], T).

The function of insert_tree(p|P, T) is performed as follows.

(1) If T has no child or can not find a child which its item-no=p, then create a new node N. N.item-no=p, N.count=1, N.cp-link=T; Insert N before the first node which item-no is greater than p.
If T has a child N such that N.item-no=p, then increment N.count by 1.

(2) If P is not empty, then call insert_tree(P, N) recursively.

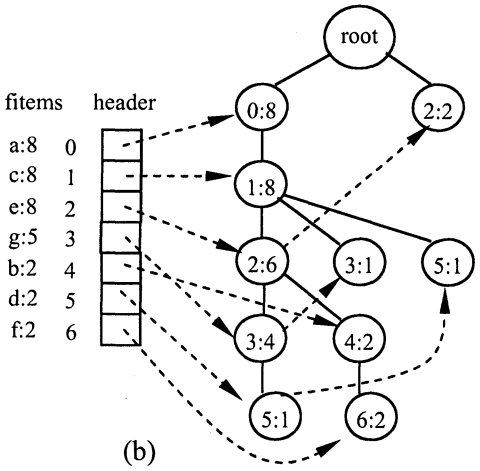
After the construction of CFP-tree, we should change the sn-link from sibling-link to next-link and reverse the cp-link. The processing procedure is as follows: Traverse the tree from the root. Add current node CN to the link of header[CN.item-no] as the last node. If CN has no child or all of its children and siblings have been processed then let CN.cp-link=CN's parent, else process its children and its siblings recursively.

Figure 1 (a) shows an example of a database and Figure 1 (b) is the CFP-tree for that database.

tid items

1	abcefo
2	acg
3	ei
4	acdeg
5	acegl
6	ej
7	abcefp
8	acd
9	acegm
10	acegn

(a)



(b)

Figure 1. An Example of CFP-tree (minsup=20%)

3. CFP-TREE BASED MINING ALGORITHM

In the mining process, FP-growth algorithm must recursively generate huge number of conditional FP-trees that requires much more memory and costs more time. In [5], an approach that needn't to generate conditional FP-trees is proposed. It uses constrained subtrees to mine frequent pattern directly.

3.1 Constrained subtree

Definition 1. The semi-order relation "$\prec$" of two patterns (itemsets) is defined as follows: Let $\{a_1, a_2, ..., a_m\}$ and $\{b_1, b_2, ..., b_n\}$ be two patterns. $\{a_1, a_2, ..., a_m\} \prec \{b_1, b_2, ..., b_n\}$ if and only if exist $1 \le i \le \min(m, n)$, while $1 \le j < i$, $a_j = b_j$, and $a_i \prec b_i$; or while $1 \le j \le m$, $a_j = b_j$, and $m<n$.

Definition 2. Let $i_1 \prec i_2 \prec ... \prec i_k$ be item orders. N is a node in the CFP-tree. P is a sub-path from root to N. We say P is constrained by the itemset $\{i_1, i_2, ..., i_k\}$, if exist a N's descendant node M, such that $i_1, i_2, ..., i_k$ appear in the sub-path from N to M, and i_1 is the item-no of N's child, i_k = M.item-no. N is called end node of sub-path P. Node M's count c is called the base count of constrained sub-path P.

Definition 3. In a CFP-tree, all of the sub-path constrained by itemset $\{i_1, i_2, ..., i_k\}$ make up of a subtree, it is called a subtree constrained by $\{i_1, i_2, ..., i_k\}$, nominated as ST($i_k, ..., i_2, i_1$).

ST($i_k, ..., i_2, i_1$) can be represented by an array, let it be EndArray, that every element of it has two fields: end-ptr (point to the end node) and base-count(store the base count of the constrained sub-path). The frequent items and count of ST($i_k, ..., i_2, i_1$) can be represented by ST($i_k, ..., i_2, i_1$).fitem[] and ST($i_k, ..., i_2, i_1$).count[] respectively.

Figure 2 shows the constrained subtree ST(3) of the CFP-tree in Figure 1 (b).

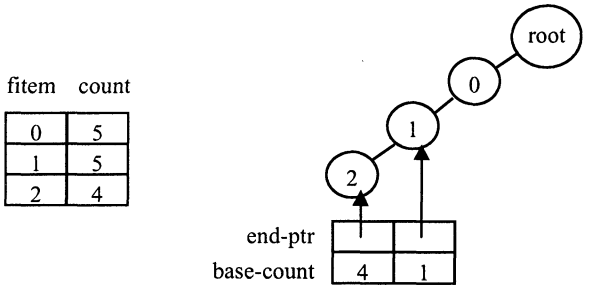
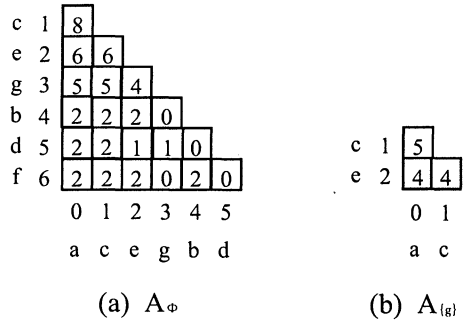


Figure 2. The constrained subtree ST(3)

3.2 An array technique

The main work done in the mining process of constrained subtree-based approach is recursively traversing a constrained subtree and generate a new subtree. For each frequent item k in constrained subtree ST(X), two traverses of ST(X) are needed for generate frequent items and counts of ST(X, k). Can the traversal time be reduced? In [6], an array-based technique was proposed to reduce the traversal time, this technique just supplies a gap of [5]. The following example will explain the idea. In Figure 3 (a), A_Φ is generated from database in Figure 1 (a) while building CFP-tree. Let the minimum support is 20%, after first scan of the database, we sort the frequent items as {a:8, c:8, e:8, g:5, b:2, d:2, f:2}. Let the itemset in support descending order be F1 (let the order of first item be 0).



(a) A_Φ (b) $A_{\{g\}}$

Figure 3. Two array examples

In A_Φ, each cell is a counter of a 2-itemset. Because c's order is 1, a's order is 0, cell $A_\Phi[1, 0]$ is the counter for itemset {c, a}, and so forth. During the second scan, the frequent items in each transaction are selected and replaced with their order in F1. Sort them as Is. When we insert Is into CFP-tree, at the same time $A_\Phi[i, j]$ is incremented by 1 if {i, j} is contained in Is. For example, the first transaction, {a, b, c, e, f} is selected and replaced with {0, 4, 1, 2, 6}. After sorted, the set becomes {0, 1, 2, 4, 6}, so $A_\Phi[1,0]$, $A_\Phi[2,0]$, $A_\Phi[2,1]$, $A_\Phi[4,0]$, $A_\Phi[4,1]$, $A_\Phi[4,2]$, $A_\Phi[6,0]$, $A_\Phi[6,1]$, $A_\Phi[6,2]$, $A_\Phi[6,4]$ are all incremented by 1. After second scan, array keeps the counts of all pairs of frequent items, as shown in Figure 2 (a).

While generating the constrained subtree of some item k, instead of traversing CFP-tree, now we can get the frequent items of subtree from the array A_Φ. For example, by checking the third line in the table for A_Φ, frequent items a, c, e and their counts can be obtained from g's constrained subtree ST(2).

It is just the same as CFP-tree, during the generation of a new constrained subtree ST(X, k), the array $A_{X\cup\{k\}}$ is filled. For example, the cells of array $A_{\{g\}}$ is shown in Figure 2 (b).

3.3 CFPmine Algorithm

From the analysis of above, we can give a new algorithm CFPmine based on constrained subtree and array-based technique. The following is the pseudocode of CFPmine. CFPmine is a main procedure, it output frequent 1-itemset and generate constrained subtree that has only one constraint item, then call mine procedure to generate frequent itemsets that have more than one item. The most work of mining is done by mine procedure.

```
Procedure CFPmine(T)
Input: A CFP-tree T
Output: The complete set of FI's corresponding to T
Method:
(1) patlen=1;
(2) for (k=flen-1; k>=0; k--) { // flen is the length of frequent itemset
(3)     pat[0]=fitem[k];
(4)     output { pat[0]} with support count[k];
(5)     generate ST(k).EndArray[];
(6)     mine(ST(k));
    }

Procedure mine(ST(i_k,...,i_2,i_1))
{
(1) generate ST( i_k,...,i_2,i_1 ).fitem[] and ST( i_k,...,i_2,i_1 ).count[], let the
    length be listlen;
(2) if (listlen==0 ) then return;
(3) if (listlen==1) then {pat[patlen]= ST( i_k,...,i_2,i_1 ).fitem[0];
      output pat with support ST( i_k,...,i_2,i_1 ).count[0]; return}
(4) if ST( i_k,...,i_2,i_1 ) has only single path then
    { output pat ∪ all the combination of ST( i_k,...,i_2,i_1 ).fitem[];
      return; }
(5) patlen++;
(6) for (k=listlen-1; k>=0; k--) {
(7)     generate array;
(8)     generate ST( i_k,...,i_2,i_1,k ).EndArray[];
(9)     if ST( i_k,...,i_2,i_1,k ).EndArray[] is not NULL then
          mine(ST( i_k,...,i_2,i_1,k ));
(10) }
(11) patlen--;
}
```

In mine procedure, line 1 generate frequent itemset in the constrained subtree ST($i_k,...,i_2,i_1$). Line 2~3 process the condition while listlen is 0 or 1. Line 4 process the condition while constrained subtree has only single path. Line 6~10 generate new array and constrained subtree, then mine the new subtree.

Because the CFP-tree could have millions of nodes, thus, it takes plenty of time for allocating and deallocating the nodes. Just like [6], we also implement unified memory management for CFPmine algorithm. In the recursively mining process, the memory used in it is not frequently allocated and freed. It allocating a large chunk before the first recursion, and when a recursion ends, it doesn't really free the memory, only changes the available size of chunk. If the chunk is used up, it allocates more memory. And it frees the memory only when all the recursions have been finished.

4. EXPERIMENTAL EVALUATION

In this section, we present a performance comparison of CFPmine algorithm with Apriori, Eclat and FP-growth. The experiments were done on 1Ghz Pentium III PC with 384MB main memery, running Windows 2000 professional.The source codes of Apriori and Eclat are provided by Christian Borgelt [12]. Thereinto, Apriori algorithm uses many optimizing techniques to improve the efficiency, such as it doesn't scan the database for many times and it uses a transaction tree to count the support of candidate sets. So, the Apiori here is much faster than the original one. The source code of FP-growth is provided by Bart Goethals [13]. All of these three algorithms are run in Cygwin environment. CFPmine algorithm is coded in Microsoft Visual C++ 6.0. All the times in the Figures refer to the running time from read data to the frequent patterns have been mined, but excluding the result writing time.

We ran all algorithms on two datasets. T25I20D100K, a synthetic dataset, is from IBM Almaden Research Center [14]. It has 100k transactions, 10k different items, and the average transaction length is 25. Connect-4 is a dense dataset. It is from UCI Machine Learning Repository [15]. It has 67557 transactions, 130 different items, and the average transaction length is 43. All of the algorithms get the same frequent itemsets for the same dataset under the same minimum support threshold.

Figure 4 gives the running time of the four algorithms on dataset T25I20D100K. It shows the performance is that CFPmine $\succ$ Apriori $\succ$ FP-growth $\succ$ Eclat. Thereinto, CFPmine is above five times faster than FP-growth. Figure 5 gives the running time of the four algorithms on dataset Connect-4 (The time plots in Figure 5 are on a logarithmic scale). It shows

the performance is that CFPmine ≻ FP-growth ≻ Eclat ≻ Apriori (while support > 65%, Apriori ≻ Eclat). Thereinto, CFPmine is a magnitude faster than FP-growth.

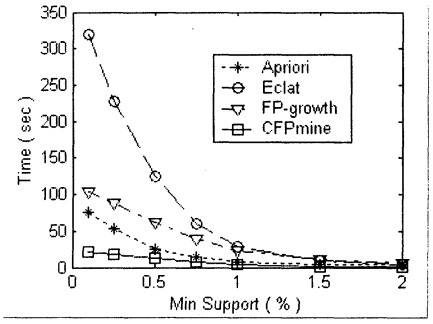


Figure 4. [Running time on T25I20D100K]

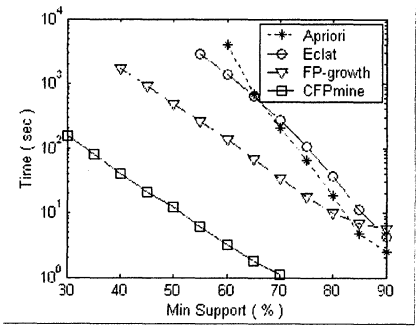


Figure 5. [Running time on Connect-4]

Discussion: From the figures, we can see that CFPmine is the most efficient algorithm in four algorithms. It also shows that the techniques used in CFPmine is effective. CFPmine algorithm has higher performance than FP-growth on the scale of either time or space. Apriori, Eclat and FP-growth have different efficiency on different dataset. It is because that: (1) The characters of two datasets is different. T25I20D100K is a sparse dataset, and Connect-4 is a dense dataset. (2) Each algorithm takes different data structure and mining strategy. The optimized Apriori is more efficient than FP-growth on sparse dataset, but FP-growth is more efficient than Apriori on dense dataset. It is because that Apriori uses a breadth-first strategy in the mining process, when the dataset is dense, the number of combinations grows fast makes it become less efficient. But the high compressed tree structure and the divide-and-conquer strategy make CFPmine and FP-growth efficient on dense datasets. Eclat uses a vertical representation of dataset. It is low performance on sparse dataset, and is relatively efficient only while the dataset is dense and has little items.

5. CONCLUSIONS

We have introduced a new algorithm, CFPmine. It is inspired by several previous works. CFPmine algorithm combines several advantages of existing techniques. One is using constrained subtrees of a compact FP-tree to mine frequent pattern, so that it is doesn't need to construct conditional FP-trees in the mining process. Second is using an array-based technique to reduce the traverse time to the CFP-tree. And an unified memeory management is also

implemented in the algorithm. The experimental evaluation shows that CFPmine algorithm is a high performance algorithm. It outperforms Apriori, Eclat and FP-growth and requires less memory than FP-growth.

ACKNOWLEDGEMENTS

This work is supported by the National Natural Science Foundation of China.(Grant No. 90104021,60173017)

REFERENCES

1. Agarwal R C, Aggarwal C C, and Prasad V V V. A Tree Projection Algorithm for Generation of Frequent Itemsets. Journal of Parallel and Distributed Computing, 2001.
2. Agrawal R, Imielinski T, Swami A. Mining association rules between sets of items in large database. In Proc of 1993 ACM SIGMOD Conf on Management of Data, 207~216, Washington DC, May 1993.
3. Agrawal R, Srikant R. Fast algorithms for mining association rules. In Proc of the 20th Int'l Conf on Very Large DataBases (VLDB'94). 487~499. Santiago, Chile, Sept. 1994.
4. Brin S, Motwani R, Ullman J D, and Tsur S. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2):255, 1997
5. FAN Ming, LI Chuan. Mining frequent patterns in an FP-tree without conditional FP-tree generation(In Chinese). Journal of computer research and development, 40(8):1216~1222. 2003.
6. Grahne G, Zhu J. Efficiently using prefix-trees in mining frequent itemsets. In: First Workshop on Frequent Itemset Mining Implementation (FIMI'03). Melbourne, FL
7. Han J, Pei J, and Yin Y. Mining Frequent Patterns without Candidate Generation. In Proc of 2000 ACM-SIGMOD Int'l Conf on Management of Data (SIGMOD'00). 1~12. Dallas, TX, 2000.
8. Park J S, Chen M-S and Yu P S. An Effective Hash-based Algorithm for Mining Association Rules. In: Proc of 1995 ACM-SIGMOD int'l Conf on Management of Data (SIGMOD'95). San Jose, CA, 1995. 175~186.
9. Savasere A, Omiecinski E, Navathe S. An efficient Algorithm for Mining Association Rules in Large Databases, In Proc of 21st Int'l Conf on Very Large Databases (VLDB'95), pages 432~443. Zurich, Switzerland, Sept. 1995.
10. Toivonen H. Sampling Large Databases for Association Rules. In Proc of 22nd Int'l Conf on Very Large Databases (VLDB'96). pages134~145. Bombay, India, Sept. 1996.
11. Zaki M, Parthasarathy S, Ogihara M, and Li W. New algorithms for fast discovery of association rules. In Heckerman D, Mannila H, Pregibon D, and Uthurusamy R eds, Proc of the Third International Conference on Knowledge Discovery and Data Mining (KDD-97), page 283. AAAI Press, 1997. http://citeseer.ist.psu.edu/zaki97new.html
12. http://fuzzy.cs.uni-magdeburg.de/~borgelt/
13. http://www.cs.helsinki.fi/u/goethals/
14. http://www.almaden.ibm.com/software quest/Resources/datasets/syndata.html
15. http://www.ics.uci.edu/~mlearn/MLRepository.html