Efficiently Using Prefix-trees in Mining Frequent Itemsets

Gösta Grahne and Jianfei Zhu
Concordia University
Montreal, Canada
{grahne, j_zhu}@cs.concordia.ca

Abstract

Efficient algorithms for mining frequent itemsets are crucial for mining association rules. Methods for mining frequent itemsets and for iceberg data cube computation have been implemented using a prefix-tree structure, known as an FP-tree, for storing compressed information about frequent itemsets. Numerous experimental results have demonstrated that these algorithms perform extremely well. In this paper we present a novel array-based technique that greatly reduces the need to traverse FP-trees, thus obtaining significantly improved performance for FP-tree based algorithms. Our technique works especially well for sparse datasets.

Furthermore, we present new algorithms for a number of common data mining problems. Our algorithms use the FP-tree data structure in combination with our array technique efficiently, and incorporates various optimization techniques. We also present experimental results which show that our methods outperform not only the existing methods that use the FP-tree structure, but also all existing available algorithms in all the common data mining problems.

1. Introduction

A fundamental problem for mining association rules is to mine frequent itemsets (FI's). In a transaction database, if we know the support of all frequent itemsets, the association rules generation is straightforward. However, when a transaction database contains large number of large frequent itemsets, mining *all* frequent itemsets might not be a good idea. As an example, if there is a frequent itemset with size ℓ, then all 2^ℓ nonempty subsets of the itemset have to be generated. Thus, a lot of work is focused on discovering only all the *maximal frequent itemsets* (MFI's). Unfortunately, mining only MFI's has the following deficiency. From an MFI and its support s, we know that all its subsets are frequent and the support of any of its subset is not less than s, but we do not know the exact value of the support. To solve this problem, another type of a frequent itemset, the *Closed Frequent Itemset* (CFI), has been proposed. In most cases, though, the number of CFI's is greater than the number of MFI's, but still far less than the number of FI's.

In this work we mine FI's, MFI's and CFI's by efficiently using the FP-tree, the data structure that was first introduced in [6]. The FP-tree has been shown to be one of the most efficient data structures for mining frequent patterns and for "iceberg" data cube computations [6, 7, 9, 8].

The most important contribution of our work is a novel technique that uses an array to greatly improve the performance of the algorithms operating on FP-trees. We first demonstrate that the use of our array-based technique drastically speeds up the FP-growth method, since it now needs to scan each FP-tree only once for each recursive call emanating from it. We then use this technique and give a new algorithm FPmax*, which extends our previous algorithm FPmax, for mining maximal frequent itemsets. In FPmax*, we use a variant of the FP-tree structure for subset testing, and give number of optimizations that further reduce the running time. We also present an algorithm, FPclose, for mining closed frequent itemsets. FPclose uses yet another variation of the FP-tree structure for checking the closedness of frequent itemsets.

Finally, we present experimental results that demonstrate the fact that all of our FP-algorithms outperform previously known algorithms practically always.

The remaining of the paper is organized as follows. In Section 2, we briefly review the FP-growth method, and present our novel array technique that results in the greatly improved method FPgrowth*. Section 3 gives algorithm FPmax*, which is an extension of our previous algorithm FPmax, for mining MFI's. Here we also introduce our approach of subset testing needed in mining MFI's and CFI's. In Section 4 we give algorithm FPclose, for mining CFI's. Experimental results are given in Section 5. Section 6 concludes, and outlines directions of future research.

2. Discovering FI's

2.1. The FP-tree and FP-growth method

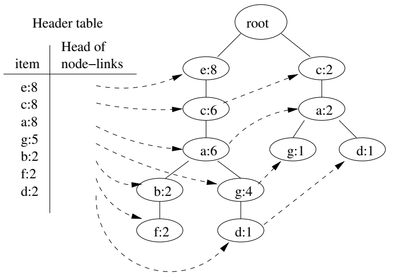
The FP-growth method by Han *et al.* [6] uses a data structure called the FP-tree (Frequent Pattern tree). The FP-tree is a compact representation of all relevant frequency information in a database. Every branch of the FP-tree represents a frequent itemset, and the nodes along the branches are stored in decreasing order of frequency of the corresponding items, with leaves representing the least frequent items. Compression is achieved by building the tree in such a way that overlapping itemsets share prefixes of the corresponding branches.

The FP-tree has a header table associated with it. Single items and their counts are stored in the header table in decreasing order of their frequency. The entry for an item also contains the head of a list that links all the corresponding nodes of the FP-tree.

Compared with Apriori [1] and its variants which need several database scans, the FP-growth method only needs two database scans when mining all frequent itemsets. The first scan counts the number of occurrences of each item. The second scan constructs the initial FP-tree which contains all frequency information of the original dataset. Mining the database then becomes mining the FP-tree.

a b c e f o
a c g
e i
a c d e g
a c e g l
e j
a b c e f p
a c d
a c e g m
a c e g n

(a)



(b)

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

To construct the FP-tree, first find all frequent items by an initial scan of the database. Then insert these items in the header table, in decreasing order of their count. In the next (and last) scan, as each transaction is scanned, the set of frequent items in it are inserted into the FP-tree as a branch. If an itemset shares a prefix with an itemset already in the tree, the new itemset will share a prefix of the branch representing that itemset. In addition, a counter is associated with each node in the tree. The counter stores the number of transactions containing the itemset represented by the path from the root to the node in question. This counter is updated during the second scan, when a transaction causes the insertion of a new branch. Figure 1 (a) shows an example of a database and Figure 1 (b) the FP-tree for that database. Note that there may be more than one node corresponding to an item in the FP-tree. The frequency of any one item i is the sum of the count associated with all nodes representing i, and the frequency of an itemset equals the sum of the counts of the least frequent item in it, restricted to those branches that contain the itemset. For instance, from Figure 1 (b) we can see that the frequency of the itemset $\{c, a, g\}$ is 5.

Thus the constructed FP-tree contains all frequency information of the database. Mining the database becomes mining the FP-tree. The FP-growth method relies on the following principle: if X and Y are two itemsets, the count of itemset $X \cup Y$ in the database is exactly that of Y in the restriction of the database to those transactions containing X. This restriction of the database is called the *conditional pattern base* of X, and the FP-tree constructed from the conditional pattern base is called X's *conditional FP-tree*, which we denote by T_X. We can view the FP-tree constructed from the initial database as $T_\emptyset$, the conditional FP-tree for $\emptyset$. Note that for any itemset Y that is frequent in the conditional pattern base of X, the set $X \cup Y$ is a frequent itemset for the original database.

Given an item i in the header table of an FP-tree T_X, by following the linked list starting at i in the header table of T_X, all branches that contain item i are visited. These branches form the conditional pattern base of $X \cup \{i\}$, so the traversal obtains all frequent items in this conditional pattern base. The FP-growth method then constructs the conditional FP-tree $T_{X \cup \{i\}}$, by first initializing its header table based on the found frequent items, and then visiting the branches of T_X along the linked list of i one more time and inserting the corresponding itemsets in $T_{X \cup \{i\}}$. Note that the order of items can be different in T_X and $T_{X \cup \{i\}}$. The above procedure is applied recursively, and it stops when the resulting new FP-tree contains only one single path. The complete set of frequent itemsets is generated from all single-path FP-trees.

2.2. An array technique

The main work done in the FP-growth method is traversing FP-trees and constructing new conditional FP-trees after the first FP-tree is constructed from the original database. From numerous experiments we found out that about 80% of the CPU time was used for traversing FP-trees. Thus, the question is, can we reduce the traversal time so that the method can be sped up?

The answer is yes, by using a simple additional data structure. Recall that for each item i in the header of a conditional FP-tree T_X, two traversals of T_X are needed for constructing the new conditional FP-tree $T_{X \cup \{i\}}$. The first traversal finds all frequent items in the conditional pattern base of $X \cup \{i\}$, and initializes the FP-tree $T_{X \cup \{i\}}$ by constructing its header table. The second traversal constructs the new tree $T_{X \cup \{i\}}$. We can omit the first scan of T_X by constructing an array A_X while building T_X. The following example will explain the idea. In Figure 1 (a), supposing that the minimum support is 20%, after the first scan of the original database, we sort the frequent items as e:8, c:8, a:8, g:5, b:2, f:2, d:2. This order is also the order of items in the header table of $T_\emptyset$. During the second scan of the database we will construct $T_\emptyset$, and an array $A_\emptyset$. This array will store the counts of all 2-itemsets. All cells in the array are initialized as 0.

c	6					
a	6	8				
g	4	5	5			
b	2	2	2	0		
f	2	2	2	0	2	
d	1	2	2	1	0	0
	e	c	a	g	b	f

(a) $A_\emptyset$

c	5	
e	4	4
	a	c

(b) $A_{\{g\}}$

Figure 2. Two array examples

In $A_\emptyset$, each cell is a counter of a 2-itemset, cell $A_\emptyset[d,e]$ is the counter for itemset $\{d,e\}$, cell $A_\emptyset[d,c]$ is the counter for itemset $\{d,c\}$, and so forth. During the second scan for constructing $T_\emptyset$, for each transaction, first all frequent items in the transaction are extracted. Suppose these items form itemset I. To insert I into $T_\emptyset$, the items in I are sorted according to the order in header table of $T_\emptyset$. When we insert I into $T_\emptyset$, at the same time $A_\emptyset[i,j]$ is incremented by 1 if $\{i,j\}$ is contained in I. For example, for the first transaction, $\{a,b,c,e,f\}$ is extracted (item o is infrequent) and sorted as e,c,a,b,f. This itemset is inserted into $T_\emptyset$ as usual, and at the same time, $A_\emptyset[f,e]$, $A_\emptyset[f,c]$, $A_\emptyset[f,a]$, $A_\emptyset[f,b]$, $A_\emptyset[b,a]$, $A_\emptyset[b,c]$, $A_\emptyset[b,e]$, $A_\emptyset[a,e]$, $A_\emptyset[a,c]$, $A_\emptyset[c,e]$ are all incremented by 1. After the second scan, array $A_\emptyset$ keeps the counts of all pairs of frequent items, as shown in table (a) of Figure 2.

Next, the FP-growth method is recursively called to mine frequent itemsets for each item in header table of $T_\emptyset$. However, now for each item i, instead of traversing $T_\emptyset$ along the linked list starting at i to get all frequent items in i's conditional pattern base, $A_\emptyset$ gives all frequent items for i. For example, by checking the third line in the table for $A_\emptyset$, frequent items e,c,a for the conditional pattern base of g can be obtained. Sorting them according to their counts, we get a,c,e. Therefore, for each item i in $T_\emptyset$ the array $A_\emptyset$ makes the first traversal of $T_\emptyset$ unnecessary, and $T_{\{i\}}$ can be initialized directly from $A_\emptyset$.

For the same reason, from a conditional FP-tree T_X, when we construct a new conditional FP-tree for $X \cup \{i\}$, for an item i, a new array $A_{X \cup \{i\}}$ is calculated. During the construction of the new FP-tree $T_{X \cup \{i\}}$, the array $A_{X \cup \{i\}}$ is filled. For instance, in Figure 1, the cells of array $A_{\{g\}}$ is shown in table (b) of Figure 2. This array is constructed as follows. From the array $A_\emptyset$, we know that the frequent items in the conditional pattern base of $\{g\}$ are, in order, a,c,e. By following the linked list of g, from the first node we get $\{e,c,a\}:4$, so it is inserted as $(a:4, c:4, e:4)$ into the new FP-tree $T_{\{g\}}$. At the same time, $A_{\{g\}}[e,c]$, $A_{\{g\}}[e,a]$ and $A_{\{g\}}[c,a]$ are incremented by 4. From the second node in the linked list, $\{c,a\}:1$ is extracted, and it is inserted as $(a:1, c:1)$ into $T_{\{g\}}$. At the same time, $A_{\{g\}}[c,a]$ is incremented by 1. Since there are no other nodes in the linked list, the construction of $T_{\{g\}}$ is finished, and array $A_{\{g\}}$ is ready to be used for construction of FP-trees in next level of recursion. The construction of arrays and FP-trees continues until the FP-growth method terminates.

Based on above discussion, we define a variation of the FP-tree structure in which besides all attributes given in [6], an FP-tree also has an attribute, *array*, which contains the corresponding array.

Now let us analyze the size of an array. Suppose the number of frequent items in the first FP-tree is n. Then the size of the associated array is $\sum_{i=1}^{n-1} i = n(n-1)/2$. We can expect that FP-trees constructed from the first FP-tree have fewer frequent items, so the sizes of the associated arrays decrease. At any time, since an array is an attribute of an FP-tree, when the space for the FP-tree is freed, the space for the array is also freed.

2.3. Discussion

The array technique works very well especially when the dataset is sparse. The FP-tree for a sparse dataset and the recursively constructed FP-trees will be big and bushy, due to the fact that they do not have many shared common prefixes. The arrays save traversal time for all items and the next level FP-trees can be initialized directly. In this case, the time saved by omitting the first traversals is far greater than the time needed for accumulating counts in the associated array.

However, when a dataset is dense, the FP-trees are more compact. For each item in a compact FP-tree, the traversal is fairly rapid, while accumulating counts in the associated array may take more time. In this case, accumulating counts may not be a good idea.

Even for the FP-trees of sparse datasets, the first levels of recursively constructed FP-trees are always conditional FP-trees for *the most common prefixes*. We can therefore expect the traversal times for the first items in a header table to be fairly short, so the cells for these first items are unnecessary in the array. As an example, in Figure 2 table (a), since e, c, and a are the first 3 items in the header table, the first two lines do not have to be calculated, thus saving counting time.

Note that the datasets (the conditional pattern bases) change during the different depths of the recursion. In order to estimate whether a dataset is sparse or dense, during the construction of each FP-tree we count the number of nodes in each level of the tree. Based on experiments, we found that if the upper quarter of the tree contains less than 15% of the total number of nodes, we are most likely dealing with a dense dataset. Otherwise the dataset is likely to be sparse.

If the dataset appears to be dense, we do not calculate the array for the next level of the FP-tree. Otherwise, we calculate array for each FP-tree in the next level, but the cells for the first several (say 5) items in its header table are not set.

2.4. FPgrowth* : an improved FP-growth method

Figure 3 contains the pseudocode for our new method FPgrowth*. The procedure has an FP-tree T as parameter. The tree has attributes: *base*, *header* and *array*. $T.base$ contains the itemset X, for which T is a conditional FP-tree, the attribute *header* contains the head table, and $T.array$ contains the array A_X.

```
Procedure FPgrowth*($T$)
Input:   A conditional FP-tree $T$
Output:  The complete set of FI's
         corresponding to $T$.
Method:
1. if $T$ only contains a single path $P$
2. then for each subpath $Y$ of $P$
3.   output pattern $Y \cup T.base$ with
     count = smallest count of nodes
     in $Y$
4. else for each $i$ in $T.header$
5.   output $Y = T.base \cup \{i\}$ with $i.count$
6.   if $T.array$ is not NULL
7.     construct a new header table
       for $Y$'s FP-tree from $T.array$
8.   else construct a new header table
          from $T$;
9.     construct $Y$'s conditional
        FP-tree $T_Y$ and its array $A_Y$;
10.  if $T_Y \neq \emptyset$
11.    call FPgrowth*($T_Y$);
```

Figure 3. Algorithm FPgrowth*

In *FPgrowth**, line 6 tests if the array of the current FP-tree is NULL. If the FP-tree corresponds to a sparse dataset, its array is not NULL, and line 7 will be used to construct the header table of the new conditional FP-tree from the array directly. One FP-tree traversal is saved for this item compared with the FP-growth method in [6]. In line 9, during the construction, we also count the nodes in the different levels of the tree, in order to estimate whether we shall really calculate the array, or just set $T_Y.array = NULL$.

From our experimental results we found that an FP-tree could have millions of nodes, thus, allocating and deallocating those nodes takes plenty of time. In our implementation, we used our own main memory management for allocating and deallocating nodes. Since all memory for nodes in an FP-tree is deallocated after the current recursion ends, a chunk of memory is allocated for each FP-tree when we create the tree. The chunk size is changeable. After generating all frequent itemsets from the FP-tree, the chunk is discarded. Thus we successfully avoid freeing nodes in the FP-tree one by one, which is more time-consuming.

3. FPmax*: Mining MFI's

In [5] we developed FPmax, a variation of the FP-growth method, for mining maximal frequent itemsets. Since the array technique speeds up the FP-growth method for sparse datasets, we can expect that it will be useful in FPmax too. This gives us an improved method, FPmax*. Compared to FPmax, the improved method FPmax* also has a more efficient subset test, as well as some other optimizations. It turns out that FPmax* outperforms GenMax[4] and MAFIA [3] for all cases we discussed in [5].

3.1. The MFI-Tree

Since FPmax is a depth-first algorithm, a frequent itemset can be a subset only of an already discovered MFI. In FPmax we introduced a global data structure, the *Maximal Frequent Itemset tree* (MFI-tree), to keep the track of MFI's. A newly discovered frequent itemset is inserted into the MFI-tree, unless it is a subset of an itemset already in the tree. However, for large datasets, the MFI-tree will be quite large, and sometimes one itemset needs thousands of comparisons for subset testing. Inspired by the way subset checking is done in [4], in FPmax*, we still use the MFI-tree structure, but for each conditional FP-tree T_X, a small MFI-tree M_X is created. The tree M_X will contain all maximal itemsets in the conditional pattern base of X. To see if a local MFI Y generated from a conditional FP-tree T_X is maximal, we only need to compare Y with the itemsets in M_X. This achieves a significant speedup of FPmax.

Each MFI-tree is associated with a particular FP-tree. Children of the root of the MFI-tree are item prefix subtrees. In an MFI-tree, each node in the subtree has three fields: item-name, level and node-link. The level-field will be useful for subset testing. All nodes with same item-name are linked together, as in an FP-tree. The MFI-tree also has a header table. However, unlike the header table in an FP-tree, which is constructed from traversing the previous FP-tree or using the associated array, the header table of an

MFI-tree is constructed based on the item order in the table of the FP-tree it is associated with. Each entry in the header table consists of two fields, item-name and head of a linked list. The head points to the first node with the same item-name in the MFI-tree.

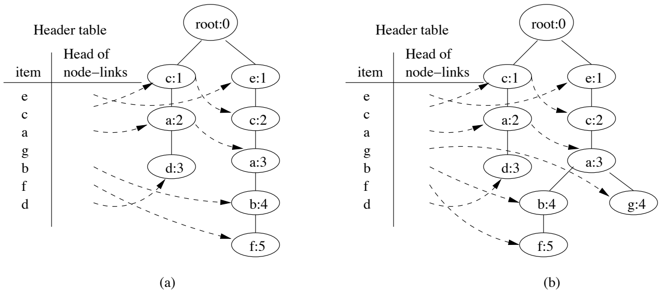


Figure 4. Construction of MFI-Tree

The insertion of an MFI into an MFI-tree is similar to the insertion of a frequent set into an FP-tree. Figure 4 shows the insertions of three MFI's into an MFI-tree associated with the FP-tree in Figure 1 (b). In Figure 4, a node $x : \ell$ means that the node is for item x and its level is ℓ. Figure 4 (a) shows the tree after (c, a, d) and (e, c, a, b, f) have been inserted. In Figure 4 (b), since new MFI (e, c, a, b, g) shares prefix (e, c, a) with (e, c, a, b, f), only one new node for g is inserted.

3.2. FPmax*

Figure 5 gives algorithm FPmax*. The first call will be for the FP-tree constructed from the original database, and it will have an empty MFI-tree. Before a recursive call *FPmax*(T,M)*, we already know from line 10 that the set containing $T.base$ and the items in the current FP-tree is not a subset of any existing MFI. During the recursion, if there is only one single path in T, this single path together with $T.base$ is an MFI of the database. In line 2, the MFI is inserted into M. If the FP-tree is not a single-path tree, then for each item i in the header table, we start preparing for the recursive call *FPmax**(T_Y, M_Y), for $Y = T.base \cup \{i\}$. The items in the header table of T are processed in increasing order of frequency, so that maximal frequent itemsets will be found before any of their frequent subsets. Lines 5 to 8 use the array technique, and line 10 calls function *subset_checking* to check if Y together with all frequent items in Y's conditional pattern base is a subset of any existing MFI in M (thus we do superset pruning here). If *subset_checking* return false, *FPmax** will be called recursively, with (T_Y, M_Y). The implementation of function *subset_checking* will be explained shortly.

Note that before and after calling *subset_checking*, if $Y \cup Tail$ is not subset of any MFI, we still do not know whether $Y \cup Tail$ is frequent. If, by constructing Y's conditional FP-tree T_Y, we find out that T_Y only has a single path, we can conclude that $Y \cup Tail$ is frequent. Since $Y \cup Tail$ was not a subset of any previously discovered MFI, it is a new MFI and will be inserted into M_Y.

```
Procedure FPmax*($T, M$)
Input:   $T$, an FP-tree
         $M$, the MFI-tree for $T.base$
Output:  Updated $M$
Method:
1. if $T$ only contains a single path $P$
2.  insert $P$ into $M$
3. else for each $i$ in $T.header$
4.  set $Y = T.base \cup \{i\}$;
5.  if $T.array$ is not NULL
6.    Tail=$\{$frequent items for $i$ in
        $T.array\}$
7.  else
8.    Tail=$\{$frequent items in $i$'s
        conditional pattern base$\}$
9.  sort Tail in decreasing order of
       the items' counts
10. if not subset_checking($Y \cup Tail, M$)
11.    construct $Y$'s conditional
         FP-tree $T_Y$ and its array $A_Y$;
12.    initialize $Y$'s conditional
         MFI-tree $M_Y$;
13.    call FPmax*($T_Y, M_Y$);
14.    merge $M_Y$ with $M$
```

Figure 5. Algorithm FPmax*

3.3. Implementation of subset testing

The function *subset_checking* works as follows. Suppose *Tail* = $i_1 i_2, \ldots i_k$, in decreasing order of frequency according to the header table of M. By following the linked list of i, for each node n in the list, we test if *Tail* is a subset of the ancestors of n. Here, the level of n can be used for saving comparison time. First we test if the level of n is smaller than k. If it is, the comparison stops because there are not enough ancestors of n for matching the rest of *Tail*. This pruning technique is also applied as we move up the branch and towards the front of *Tail*.

Unlike an FP-tree, which is not changed during the execution of the algorithm, an MFI-tree is dynamic. At line 12, for each Y, a new MFI-tree M_Y is initialized from the predecessor MFI-tree M. Then after the recursive call, M is updated on line 14 to contain all newly found frequent itemsets. In the actual implementation, we however found that it was more efficient to update all MFI-trees along the recursive path, instead of merging only at the current level. In other words, we omitted line 14, and instead on line 2, P

is inserted into the current M, and also into all predecessor MFI-trees that the implementation of the recursion needs to keep in main memory in any case.

Since *FPmax** is a depth-first algorithm, it is straightforward to show that the above subset checking is correct. Based on the correctness of the FP-growth method, we can conclude that *FPmax** returns all and only the maximal frequent itemsets in a given dataset.

3.4. Optimizations

In the method FPmax*, one more optimization is used. Suppose, that at some level of the recursion, the header table of the current FP-tree is $i_1, i_2, \ldots, i_m$. Then starting from i_m, for each item in the header table, we may need to do the work from line 4 to line 14. If for any item, say i_k, where $k \leq m$, its maximal frequent itemset contains items $i_1, i_2, \ldots, i_{k-1}$, i.e., all the items that have not yet called *FPmax** recursively, these recursive calls can be omitted. This is because for those items, their tails must be subsets of $\{i_1, i_2, \ldots, i_{k-1}\}$, so $subset\_checking(Y \cup Tail)$ would always return true.

FPmax* also uses the memory management described in Section 2.4, for allocating and deallocating space for FP-trees and MFI-trees.

3.5. Discussion

One may wonder if the space required for all the MFI-trees of a recursive branch is too large. Actually, before the first call of *FPmax**, the first FP-tree has to fit in main memory. This is also required by the FP-growth method. The corresponding MFI-tree is initialized as empty. During recursive calls of *FPmax**, new conditional FP-trees are constructed from the first FP-tree or from an ancestors FP-tree. From the experience of [6], we know the recursively constructed FP-trees are relatively small. We can expect that the total size of these FP-trees is not greater than the final size of the MFI-tree for $\emptyset$. Similarly, the MFI-trees constructed from ancestors are also small. All MFI-trees grow gradually. Thus we can conclude that the total main memory requirement for running *FPmax** on a dataset is proportional to the sum of the size of the FP-tree and the MFI-tree for $\emptyset$.

4. FPclose: Mining CFI's

For mining frequent closed itemsets, FPclose works similarly to FPmax*. They both mine frequent patterns from FP-trees. Whereas FPmax* needs to check that a newly found frequent itemset is maximal, FPclose needs to verify that the new frequent itemset is closed. For this we use a CFI-tree, which is another variation of an FP-tree.

One of the first attempts to use FP-trees in CFI mining was the algorithm CLOSET+ [9]. This algorithm uses one global prefix-tree for keeping track of all closed itemsets. As we pointed out before, one global tree will be quite big, and thus slows down searches. In FPclose we will therefore use multiple, conditional CFI-trees for checking closedness of itemsets. We can thus expect that FPclose outperforms CLOSET+.

4.1. The CFI-tree and algorithm FPclose

Similar to an MFI-tree, a CFI-tree is related to an FP-tree and an itemset X, and we will denote the CFI-tree as C_X. The CFI-tree C_X always stores all already found CFI's containing itemset X, and their counts. A newly found frequent itemset Y that contains X only needs to be compared with the CFI's in C_X. If in C_X, there is no superset of Y with same count as Y, Y is closed.

In a CFI-tree, each node in the subtree has four fields: item-name, count, node-link and level. Here, the count field is needed because when comparing a Y with a set Z in the tree, we are trying to verify that it is not the case that $Y \subset Z$, and Y and Z have the same count. The order of the items in a CFI-tree's header table is same as the order of items in header table of its corresponding FP-tree.

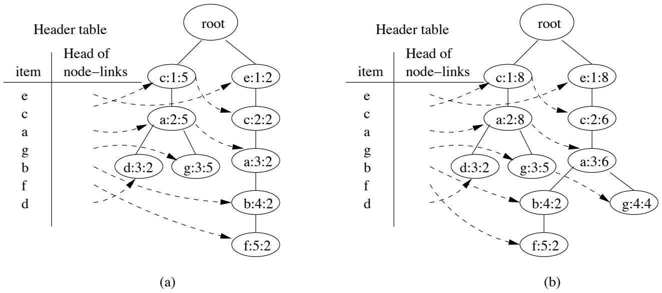


Figure 6. Construction of CFI-Tree

The insertion of a CFI into a CFI-tree is similar to the insertion of a transaction into an FP-tree, except now the count of a node is not incremented, it is always replaced by the maximal count up-to-date. Figure 6 shows some snapshots of the construction of a CFI-tree with respect to the FP-tree in Figure 1 (b). The item order in two trees are same because they are both for base $\emptyset$. Note that insertions of CFI's into the top level CFI-tree will occur only after recursive calls have been made. In the following example, the insertions would in actuality be performed during various stages of the execution, not in bulk as the example might suggest. In Figure 6, a node $x : \ell : c$ means that the node is for item x, its level is ℓ and its count is c. In Figure 6 (a), after inserting (c, a, d) and (e, c, a, b, f) with count 2, then we insert (c, a, g) with count 5. Since (c, a, g) shares the prefix (c, a) with (c, a, d), only node g is appended, and at the same time, the counts for nodes c and a are both changed to be 5. In part (b) of Figure 6, the CFI's $(e, c, a, g) : 4$,

$(c, a) : 8$, $(c, a, e) : 6$ and $(e) : 8$ are inserted. At this stage the tree contains all CFI's for the dataset in Figure 1 (a).

```
Procedure FPclose($T, C$)
Input:   $T$, an FP-tree
         $C$, the CFI-tree for $T.base$
Output:  Updated $C$
Method:
1. if $T$ only contains a single path $P$
2.   generate all CFI's from $P$
3.   for each CFI $X$ generated
4.     if not closed_checking($X, C$)
5.       insert $X$ into $C$
6. else for each $i$ in $T.header$
7.    set $Y = T.base \cup \{i\}$;
8.    if not closed_checking($Y, C$)
9.      if $T.array$ is not NULL
10.       Tail = {frequent items for
           $i$ in $T.array$}
11.     else
12.       Tail={frequent items in $i$'s
           conditional pattern base}
13.     sort Tail in decreasing order
         of items' counts
14.     construct the FP-tree $T_Y$ and
         its array $A_Y$;
15.     initialize $Y$'s conditional
         CFI-tree $C_Y$;
16.     call FPclose($T_Y, C_Y$);
17.     merge $C_Y$ with $C$
```

Figure 7. Algorithm *FPclose*

Figure 7 gives algorithm *FPclose*. Before calling *FPclose* with some (T, C), we already know from line 8 that there is no existing CFI X such that $T.base \subset X$, and $T.base$ and X have the same count. If there is only one single path in T, the nodes and their counts in this single path can be easily used to list the $T.base$-local closed frequent itemsets. These itemsets will be compared with the CFI's in C. If an itemset is closed, it is inserted into C. If the FP-tree T is not a single-path tree, we execute line 6. Lines 9 to 12 use the array technique. Lines 4 and 8 call function $closed\_checking(Y, C)$ to check if a frequent itemset Y is closed. If it is, the function returns true, otherwise, false is returned. Lines 14 and 15 construct Y's conditional FP-tree and CFI-tree. Then FPclose is called recursively for T_Y and C_Y.

Note that line 17 is not implemented as such. As in algorithm FPmax*, we found it more efficient to do the insertion of lines 3–5 into all CFI-trees currently in main memory.

CFI-trees are initialized similarly to MFI-trees, described in Section 3.3. The implementation of function *closed_checking* is almost the same as the implementation of function *subset_checking*, except now we also consider the count of an itemset. Given an itemset $Y = \{i_1, i_2, \ldots, i_k\}$ with count c, suppose the order of the items in header table of the current CFI-tree is $i_1, i_2, \ldots, i_k$. Following the linked list of i_k, for each node in the list, first we check if its count is equal to or greater than c. If it is, we then test if Y is a subset of the ancestors of that node. The function *closed_checking* returns true only when there is no existing CFI Z in the CFI-tree such that Z is a superset of Y and the count of Y is equal to or greater than the count of Z.

Memory management allocating and deallocating space for FP-trees and CFI-trees is similar to the memory management of *FPgrowth** and *FPmax**.

By a similar reasoning as in Section 3.5, we conclude that the total main memory requirement for running *FPclose* on a dataset is approximately sum of the size of the first FP-tree and its CFI-tree.

5. Experimental Evaluation

We now present a performance comparison of our FP-algorithms with algorithms dEclat, GenMax, CHARM and MAFIA. Algorithm dEclat is a depth-first search algorithm proposed by Zaki and Gouda in [10]. dEclat uses a linked list to organize frequent patterns, however, each itemset now corresponds to an array of transaction IDs (the "TID-array"). Each element in the array corresponds to a transaction that contains the itemset. Frequent itemset mining and candidate frequent itemset generation are done by TID-array intersections. A technique called *diffset*, is used for reducing the memory requirement of TID-arrays. The diffset technique only keeps track of differences in the TID's of a candidate itemsets when it is generating frequent itemsets. GenMax, also proposed by Gouda and Zaki [4], takes an approach called *progressive focusing* to do maximality testing. CHARM is proposed by Zaki and Hsiao [11] for CFI mining. In all three algorithms, the main operation is the intersection of TID-arrays. Each of them has been shown as one of the best algorithms for mining FI's, MFI's or CFI's. MAFIA is introduced in [3] by Burdick *et al.* for mining maximal frequent itemsets. It also has options for mining FI's and CFI's. We give the results of three different sets of experiments, one set for FI's, one for MFI's and one for CFI's.

The source codes for dEclat, CHARM, GenMax and MAFIA were provided by their authors. We ran all algorithms on many synthetic and real datasets. Due to the lack of space, only the results for two synthetic datasets and two real datasets are shown here. These datasets should be representative, as recent research papers [2, 3, 4, 11, 10, 8, 9], use these or similar datasets.

The two synthetic datasets, *T40I10D100K* and *T100I20D100K*, were generated from the application on the website of IBM [1]. They both use 100,000 transactions and 1000 items. The two real datasets, *pumsb** and *connect-4*, were also downloaded from the IBM website [2]. Dataset *connect-4* is compiled from game state information. Dataset *pumsb** is produced from census data of Public Use Microdata Sample (PUMS). These two real datasets are both quite dense, so a large number of frequent itemsets can be mined even for very high values of minimum support.

All experiments were performed on a 1Ghz Pentium III with 512 MB of memory running RedHat Linux 7.3. All times in the figures refer to CPU time.

5.1. FI Mining

In [6], the original FPgrowth method has been shown to be an efficient and scalable algorithm for mining frequent itemsets. FPgrowth is about an order of magnitude faster than the Apriori. Subsequently, it was shown in [10], that the algorithm dEclat outperforms FPgrowth on most datasets. Thus, in the first set of experiments, FP-growth* is compared with the original FP-growth method and with dEclat. The original FP-growth method is implemented on the basis of the paper [6]. In this set of experiments we also included with MAFIA [3], which has an option for mining all FI's. The results of the first set of experiments are shown in Figure 8.

Figure 8 (a) shows the CPU time of the four algorithms running on dataset *T40I10D100K*. We see that FPgrowth* is the best algorithm for this dataset. It outperforms dEclat and MAFIA at least by a factor of two. Main memory is used up by dEclat when the minimum support goes down to 0.25%, while FPgrowth* can still run for even smaller levels of minimum support. MAFIA is the slowest algorithm for this dataset and its CPU time increases rapidly.

Due to the use of the array technique, and the fact that *T40I10D100K* is a sparse dataset, FPgrowth* turns out to be faster than FPgrowth. However, when the minimum support is very low, we can expect the FP-tree to achieve a good compactification, starting at the initial recursion level. Thus the array technique does not offer a big gain. Consequently, as verified in Figure 8 (a), for very low levels minimum support, FPgrowth* and FPgrowth have almost the same running time.

Figure 8 (b) shows the CPU time for running the four algorithms on dataset *T100I20D100K*. The result is similar to the result in Figure 8 (a). FPgrowth* is again the best. Since the dataset *T100I20D100K* is sparser than *T40I10D100K*, the speedup from FPgrowth to FPgrowth* is increased.

From Figure 8 (c) and (d), we can see that the FP-methods are faster than dEclat by an order of magnitude in both experiments. Since *pumsb** and *connect-4* are both very dense datasets, FPgrowth* and FPgrowth have almost same running time, as the array technique does not achieve a significant speedup for dense datasets.

In Figure 8 (c), the CPU time increases drastically when the minimum support goes down below 25%. However, this is not a problem for FPgrowth and FPgrowth*, which still are able to produce results. The main reason for the nevertheless steeply increased CPU time is that a long time has to be spent listing frequent itemsets. Recall, that if there is a frequent "long" itemset of size ℓ, then we have to generate 2^ℓ frequent sets from it.

We also ran the four algorithms on many other datasets, and we found that FPgrowth* was always the fastest.

To see why FPgrowth* is the fastest, let us consider the main operations in the algorithms. As discussed before, FPgrowth* spends most of its time on constructing and traversing FP-trees. The main operation in dEclat is to generate new candidate FI's by TID-array intersections. In MAFIA, generating new candidate FI's by bitvector *and*-operations is the main work. Since FPgrowth* uses the compact FP-tree, further boosted by the array technique, the time it spends constructing and traversing the trees, is less than the time needed for TID-array intersections and bitvector *and*-operations. Moreover, the main memory space needed for storing FP-trees is far less than that for storing diffsets or bitvectors. Thus FPgrowth* runs faster than the other two algorithms, and it scales to very low levels of minimum support.

Figure 11 (a) shows the main memory consumption of three algorithms by running them on dataset *connect-4*. We can see that FP-growth* always use the least main memory. And even for very low minimum support, it still uses a small amount of main memory.

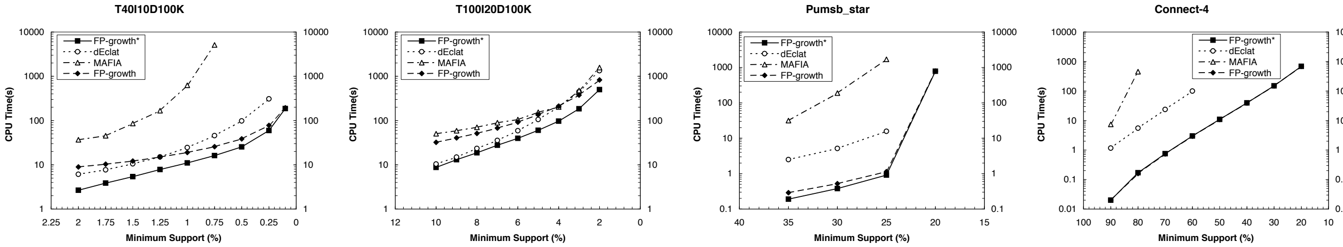
5.2. MFI Mining

In our paper [5], we analyzed and verified the performance of algorithm FPmax. We learned that FPmax outperformed GenMax and MAFIA in some, but not all cases. To see the impact of the new array technique and the new *subset_checking* function that we are using in FPmax*, in the second set of experiments, we compared FPmax* with FPmax, GenMax, and MAFIA.

Figure 9 (a) gives the result for running these algorithms on the sparse dataset *T40I10D100K*. We can see that FPmax is slower than GenMax for all levels of minimum support, while FPmax* outperforms GenMax by a factor of at least two. Figure 9 (b) shows the results for the very sparse dataset *T100I20D100K*, FPmax is the slowest algorithm, while FPmax* is the fastest algorithm. Figure 9 (c) shows that FPmax* is the fastest algorithm for the dense dataset *pumsb**, even though FPmax is the slowest algorithm on this dataset for very low levels of minimum support. In

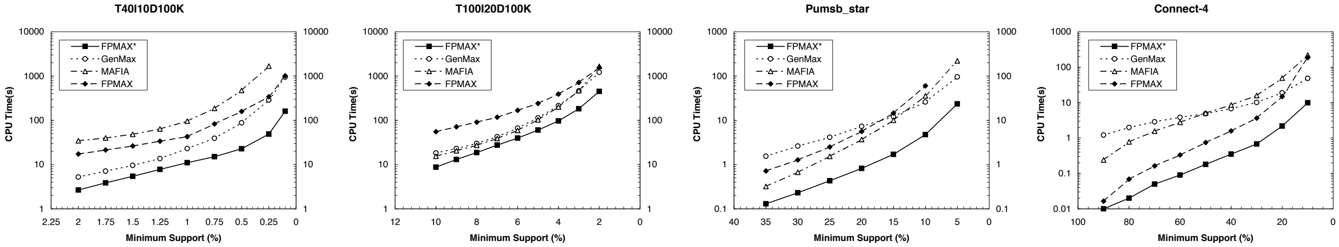
[1] http://www.almaden.ibm.com/cs/quest/syndata.html

[2] http://www.almaden.ibm.com/cs/people/bayardo/resources.html



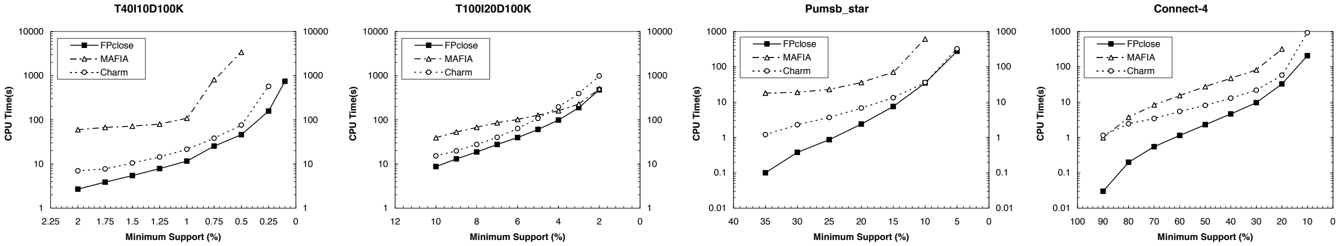
(a) (b) (c) (d)

Figure 8. Mining FI's



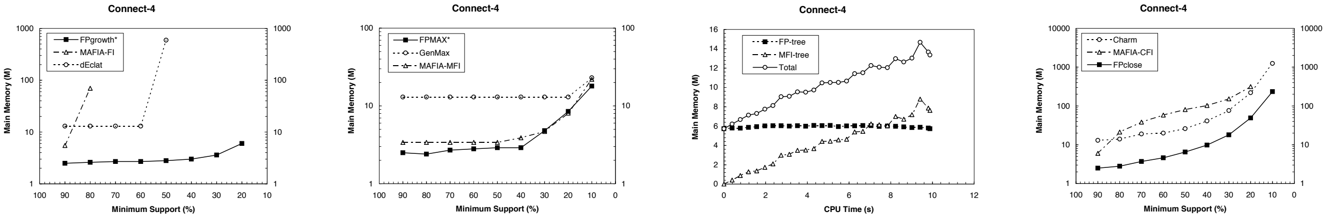
(a) (b) (c) (d)

Figure 9. Mining MFI's



(a) (b) (c) (d)

Figure 10. Mining CFI's



(a) (b) (c) (d)

Figure 11. Main Memory used by the algorithms

Figure 9 (d), FPmax outperforms GenMax and MAFIA for high levels of minimum support, but it is slow for very low levels. FPmax*, on the other hand is about one to two orders of magnitude faster than GenMax and MAFIA for all levels of minimum support.

All experiments in this second set show that the array technique and the new *subset_checking* function are indeed very effective. Figure 11 (b) shows the main memory used by three algorithms when running them on dataset *connect-4*. From the figure, we can see that FPmax* uses less main memory than the other algorithms. Figure 11 (c) shows the main memory used by FP-trees, MFI-trees and the whole algorithm when running FPmax* on dataset *connect-4*. The minimum support was set as 10%. In the figure, the last point of the line for FP-tree is for the main memory of the first FP-tree ($T_\emptyset$), since at this point the space for all conditional FP-trees has been freed. The last point of the line for MFI-tree is for the main memory of the MFI-tree that contains whole set of MFI's, i.e., $M_\emptyset$. The figure confirms our analysis of main memory used by FPmax* in Section 3.5.

We also run these four algorithms on many other datasets, and we found that FPmax* always was the fastest algorithm.

5.3. CFI Mining

In the third set of experiments, the performances of FPclose, CHARM and MAFIA, with the option of mining closed frequent itemset, were compared.

Figure 10 shows the results of running FPclose, CHARM and MAFIA on datasets *T40I10D100K*, *T100I20D100K*, *pumsb** and *connect-4*. FPclose shows good performance on all datasets, due to the fact that it uses the compact FP-tree and the array technique. However, for very low levels of minimum support FPclose has performance similar to CHARM and MAFIA. By analyzing the three algorithms, we found that FPclose generates more non-closed frequent itemsets than the other algorithms. For each of the generated frequent itemsets, the function *closed_checking* must be called. Although the *closed_checking* function is very efficient, the increased number of calls to it means higher total running time. For high levels of minimum support, the time saved by using the compact FP-tree and the array technique compensates for the time FPclose spends on *closed_checking*. In all cases, FPclose uses less main memory for mining CFI's than CHARM and MAFIA. Figure 11 (d) shows the memory used by three algorithms by running them on dataset *connnect-4*. We can see that for very low levels of minimum support, CHARM and MAFIA were aborted because they ran out of memory, while FPclose was still able to run and produce output.

6. Conclusions

We have introduced a novel array-based technique that allows using FP-trees more efficiently when mining frequent itemsets. Our technique greatly reduces the time spent traversing FP-trees, and works especially well for sparse datasets. Furthermore, we presented new algorithms for mining maximal and closed frequent itemsets.

The FPgrowth* algorithm, which extends original FP-growth method, also uses the novel array technique to mine all frequent itemsets.

For mining maximal frequent itemsets, we extended our earlier algorithm FPmax to FPmax*. FPmax* not only uses the array technique, but also a new subset-testing algorithm. For the subset testing, a variation of the FP-tree, an MFI-tree, is used for storing all already discovered MFI's. In FP-max*, a newly found FI is always compared with a small set of MFI's that are kept in an MFI-tree, thus making subset-testing much more efficient.

For mining closed frequent itemsets we give the FPclose algorithm. In the algorithm, a CFI-tree —another variation of a FP-tree— is used for testing the closedness of frequent itemsets.

For all of our algorithms we have presented several optimizations that further reduce their running time.

Our experimental results showed that FPgrowth* and FPmax* always outperforms existing algorithms. FPclose also demonstrates extremely good performance. All of the algorithms need less main memory because of the compact FP-trees, MFI-trees, and CFI-trees.

Though the experimental results given in this paper show the success of our algorithms, in the future we will test them on more applications to further study their performance. We are also planning to explore ways to improve the FPclose algorithm by reducing the number of closedness-tests needed.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of VLDB'94*, pages 487–499, 1994.

[2] R. J. Bayardo, Jr. Efficiently mining long patterns from databases. In *Proceedings of ACM SIGMOD'98*, pages 85–93, 1998.

[3] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of ICDE'01*, pages 443–452, Apr. 2001.

[4] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *Proceedings of ICDM'01*, San Jose, CA, Nov. 2001.

[5] G. Grahne and J. Zhu. High performance mining of maximal frequent itemsets. In *SIAM'03 Workshop on High Performance Data Mining: Pervasive and Data Stream Mining*, May 2003.

[6] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of ACM SIGMOD'00*, pages 1–12, May 2000.

[7] J. Han, J. Wang, Y. Lu, and P. Tzvetkov. Mining top-k frequent closed patterns without minimum support. In *Proceedings of ICDM'02*, pages 211–218, Dec. 2002.

[8] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD'00 Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[9] J. Wang, J. Han, and J. Pei. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *Proceedings of ACM SIGKDD'03*, Washington, DC, 2003.

[10] M. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proceedings of ACM SIGKDD'03*, Washington, DC, Aug. 2003.

[11] M. Zaki and C. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *Proceedings of SIAM'02*, Arlington, Apr. 2002.