Fast Algorithms for Frequent Itemset Mining Using FP-Trees

Gösta Grahne, *Member*, *IEEE*, and Jianfei Zhu, *Student Member*, *IEEE*

Abstract—Efficient algorithms for mining frequent itemsets are crucial for mining association rules as well as for many other data mining tasks. Methods for mining frequent itemsets 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 FP-array 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 data sets. Furthermore, we present new algorithms for mining all, maximal, and closed frequent itemsets. Our algorithms use the FP-tree data structure in combination with the FP-array technique efficiently and incorporate various optimization techniques. We also present experimental results comparing our methods with existing algorithms. The results show that our methods are the fastest for many cases. Even though the algorithms consume much memory when the data sets are sparse, they are still the fastest ones when the minimum support is low. Moreover, they are always among the fastest algorithms and consume less memory than other methods when the data sets are dense.

Index Terms—Data mining, association rules.

1 Introduction

Efficient mining of frequent itemsets (FIs) is a fundamental problem for mining association rules [5], [6], [21], [32]. It also plays an important role in other data mining tasks such as sequential patterns, episodes, multidimensional patterns, etc. [7], [22], [17]. The description of the problem is as follows: Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of *items* and D be a multiset of transactions, where each transaction τ is a set of items such that $\tau \subseteq I$. For any $X \subseteq I$, we say that a transaction τ *contains* X if $X \subseteq \tau$. The set X is called an *itemset*. The set of all $X \subseteq I$ (the powerset of I) naturally forms a lattice, called the *itemset lattice*. The *count* of an itemset X is the number of transactions in D that contain X. The *support* of an itemset X is the proportion of transactions in D that contain X. Thus, if the total number of transactions in D is n, then the support of X is the count of X divided by $n \cdot 100$ percent. An itemset X is called *frequent* if its support is greater than or equal to some given percentage s, where s is called the *minimum support*.

When a transaction database is very dense and the minimum support is very low, i.e., when the database contains a significant number of large frequent itemsets, mining *all* frequent itemsets might not be a good idea. For example, if there is a frequent itemset with size l, then all 2^l nonempty subsets of the itemset have to be generated. However, since frequent itemsets are *downward closed* in the itemset lattice, meaning that any subset of a frequent itemset is frequent, it is sufficient to discover only all the *maximal frequent itemsets* (MFIs). A frequent itemset X is called *maximal* if there does not exist frequent itemset Y such that $X \subset Y$. Mining frequent itemsets can thus be reduced to mining a "border" in the itemset lattice. All itemsets above the border are infrequent and those that are below the border are all frequent. Therefore, some existing algorithms only mine maximal frequent itemsets.

However, mining only MFIs 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. For generating association rules, we do need the support of all frequent itemsets. To solve this problem, another type of a frequent itemset, called *closed frequent itemset* (CFI), was proposed in [24]. A frequent itemset X is *closed* if none of its proper supersets have the same support. Any frequent itemset has the support of its smallest closed superset. The set of all closed frequent itemsets thus contains complete information for generating association rules. In most cases, the number of CFIs is greater than the number of MFIs, although still far less than the number of FIs.

1.1 Mining FIs

The problem of mining frequent itemsets was first introduced by Agrawal et al. [5], who proposed algorithm Apriori. Apriori is a bottom-up, breadth-first search algorithm. It uses hash-trees to store frequent itemsets and candidate frequent itemsets. Because of the downward closure property of the frequency pattern, only candidate frequent itemsets, whose subsets are all frequent, are generated in each database scan. Candidate frequent itemset generation and subset testing are all based on the hash-trees. In the algorithm, transactions are not stored in the memory and, thus, Apriori needs l database scans if the size of the largest frequent itemset is l. Many algorithms, such as [28], [29], [23], are variants of Apriori. In [23], the kDCI method applies a novel counting strategy to efficiently determine the itemset supports without necessarily performing all the l scans.

• *The authors are with the Department of Computer Science, Concordia University, 1455 De Maisonneuve Blvd. West, Montreal, Quebec, H3G 1M8, Canada. E-mail: {grahne, j_zhu}@cs.concordia.ca.*

Manuscript received 28 Apr. 2004; revised 27 Nov. 2004; accepted 11 Mar. 2005; published online 18 Aug. 2005.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0123-0404.

In [14], Han et al. introduced a novel algorithm, known as the FP-growth method, for mining frequent itemsets. The FP-growth method is a depth-first search algorithm. In the method, a data structure called the FP-tree is used for storing frequency information of the original database in a compressed form. Only two database scans are needed for the algorithm and no candidate generation is required. This makes the FP-growth method much faster than Apriori. In [27], PatriciaMine stores the FP-trees as Patricia Tries [18]. A number of optimizations are used for reducing time and space of the algorithm. In [33], Zaki also proposed a depth-first search algorithm, Eclat, in which database is "vertically" represented. Eclat 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. Later, Zaki and Gouda [35] introduced a technique, called *diffset*, for reducing the memory requirement of TID-arrays. The diffset technique only keeps track of differences in the TID's of candidate itemsets when it is generating frequent itemsets. The Eclat algorithm incorporating the diffset technique is called dEclat [35].

1.2 Mining MFIs

Maximal frequent itemsets were inherent in the *border* notion introduced by Mannila and Toivonen in [20]. Bayardo [8] introduced MaxMiner which extends Apriori to mine only "long" patterns (maximal frequent itemsets). Since MaxMiner only looks for the maximal FIs, the search space can be reduced. MaxMiner performs not only *subset* infrequency pruning, where a candidate itemset with an infrequent subset will not be considered, but also a "lookahead" to do *superset* frequency pruning. MaxMiner still needs several passes of the database to find the maximal frequent itemsets.

In [10], Burdick et al. gave an algorithm called MAFIA to mine maximal frequent itemsets. MAFIA uses a linked list to organize all frequent itemsets. Each itemset I corresponds to a bitvector; the length of the bitvector is the number of transactions in the database and a bit is set if its corresponding transaction contains I, otherwise, the bit is not set. Since all information contained in the database is compressed into the bitvectors, mining frequent itemsets and candidate frequent itemset generation can be done by bitvector *and*-operations. Pruning techniques are also used in the MAFIA algorithm.

GenMax, another depth-first algorithm, proposed by Gouda and Zaki [11], takes an approach called *progressive focusing* to do maximality testing. This technique, instead of comparing a newly found frequent itemset with all maximal frequent itemsets found so far, maintains a set of *local* maximal frequent itemsets. The newly found FI is only compared with itemsets in the small set of local maximal frequent itemsets, which reduces the number of subset tests.

In our earlier paper [12], we presented the FPmax algorithm for mining MFIs using the FP-tree structure. FPmax is also a depth-first algorithm. It takes advantage of the FP-tree structure so that only two database scans are needed. In FPmax, a tree structure similar to the FP-tree is used for maximality testing. The experimental results in [12] showed that FPmax outperforms GenMax and MAFIA for many, although not all, cases.

Another method that uses the FP-tree structure is AFOPT [19]. In the algorithm, item search order, intermediate result representation, and construction strategy, as well as tree traversal strategy, are considered dynamically; this makes the algorithm adaptive to general situations. SmartMiner [36], also a depth-first algorithm, uses a technique to quickly prune candidate frequent itemsets in the itemset lattice. The technique gathers "tail" information for a node in the lattice. The tail information is used to determine the next node to explore during the depth-first mining. Items are dynamically reordered based on the tail information. The algorithm was compared with MAFIA and GenMax on two data sets and the experiments showed that SmartMiner is about 10 times faster than MAFIA and GenMax.

1.3 Mining CFIs

In [24], Pasquier et al. introduced closed frequent itemsets. The algorithm proposed in the paper, A-close, extends Apriori to mine all CFIs. Zaki and Hsiao [34] proposed a depth-first algorithm, CHARM, for CFI mining. As in their earlier work in [11], in CHARM, each itemset corresponds to a TID-array, and the main operation of the mining is again TID-array intersections. CHARM also uses the diffset technique to reduce the memory requirement for TID-array intersections.

The algorithm AFOPT [19] described in Section 1.2 has an option for mining CFIs in a manner similar to the way AFOPT mines MFIs.

In [26], Pei et al. extended the FP-growth method to a method called CLOSET for mining CFIs. The FP-tree structure was used and some optimizations for reducing the search space were proposed. The experimental results reported in [26] showed that CLOSET is faster than CHARM and A-close. CLOSET was extended to CLOSET+ by Wang et al. in [30] to find the best strategies for mining frequent closed itemsets. CLOSET+ uses data structures and data traversal strategies that depend on the characteristics of the data set to be mined. Experimental results in [30] showed that CLOSET+outperformed all previous algorithms.

1.4 Contributions

In this work, we use the FP-tree, the data structure that was first introduced in [14]. The FP-tree has been shown to be a very efficient data structure for mining frequent patterns [14], [30], [26], [16] and its variation has been used for "iceberg" data cube computation [31].

One of the important contributions of our work is a novel technique that uses a special data structure, called an FP-array, to greatly improve the performance of the algorithms operating on FP-trees. We first demonstrate that the FP-array technique drastically speeds up the FP-growth method on sparse data sets, since it now needs to scan each FP-tree only once for each recursive call emanating from it. This technique is then applied to our previous algorithm FPmax for mining maximal frequent itemsets. We call the new method FPmax*. In FPmax*, we also introduce our technique for checking if a frequent itemset is maximal, for which a variant of the FP-tree structure, called an MFI-tree, is used. For mining closed frequent itemsets, we have designed an algorithm FPclose which uses yet another variant of the FP-tree structure, called a CFI-tree, for

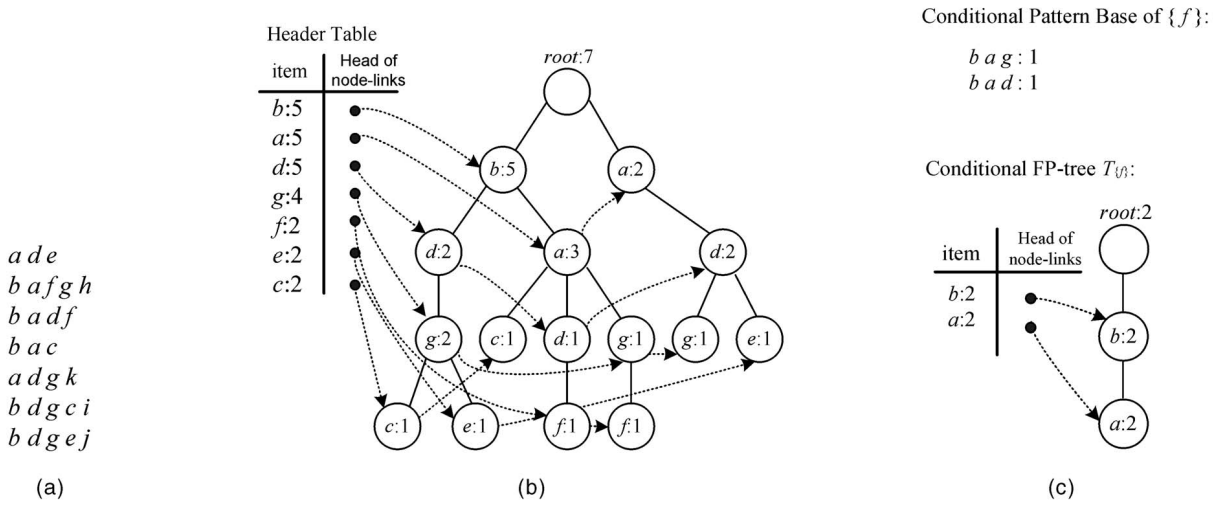

Fig. 1. An FP-tree example. (a) A database. (b) The FP-tree for the database (*minimum support = 20 percent*).

checking the closedness of frequent itemsets. The closedness checking is quite different from CLOSET+. Experimental results in this paper show that our closedness checking approach is more efficient than the approach of CLOSET+.

Both the experimental results in this paper and the independent experimental results from the first *IEEE ICDM Workshop on frequent itemset mining* (FIMI '03) [3], [32] demonstrate the fact that all of our FP-algorithms have very competitive and robust performance. As a matter of fact, in FIMI '03, our algorithms were considered to be the algorithms of choice for mining maximal and closed frequent itemsets [32].

1.5 Organization of the Paper

In Section 2, we briefly review the FP-growth method and introduce our FP-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 MFIs. Here, we also introduce our approach of maximality checking. In Section 4, we give algorithm FPclose for mining CFIs. Experimental results are presented in Section 5. Section 6 concludes and outlines directions for future research.

2 Discovering FI's

2.1 The FP-Tree and FP-Growth Method

The FP-growth method [14], [15] is a depth-first algorithm. In the method, Han et al. proposed 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.

An FP-tree T has a header table, $T.header$, 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 breadth-first algorithms such as Apriori and its variants, which may need as many database scans as the length of the longest pattern, the FP-growth method only needs two database scans when mining all frequent itemsets. The first scan is to find all frequent items. These items are inserted into the header table in decreasing order of their count. In the second scan, as each transaction is scanned, the set of frequent items in it is inserted into the FP-tree as a branch. If an itemset shares a prefix with an itemset already in the tree, this part of the branch will be shared. 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. Fig. 1a shows an example of a data set and Fig. 1b the FP-tree for that data set.

Now, 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 the empty itemset. 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 in the original database.

Given an item i in $T_X.header$, by following the linked list starting at i in $T_X.header$, all branches that contain item i are visited. The portion of these branches from i to the root forms 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 frequent items found, then revisiting the branches of T_X along the linked list of i and inserting the corresponding itemsets in $T_{X \cup \{i\}}$. Note that the order of items can be different in T_X and $T_{X \cup \{i\}}$. As an example, the conditional pattern base of $\{f\}$ and the conditional FP-tree $T_{\{f\}}$ for the

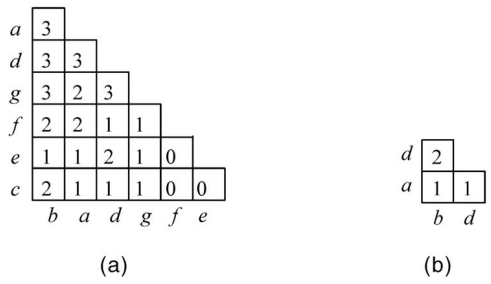

Fig. 2. Two FP-array examples. (a) $A_\emptyset$. (b) $A_{\{g\}}$.

database in Fig. 1a is shown in Fig. 1c. The above procedure is applied recursively, and it stops when the resulting new FP-tree contains only one branch. The complete set of frequent itemsets can be generated from all single-branch FP-trees.

2.2 The FP-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 percent 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 a frequent pairs array A_X while building T_X. We initialize T_X with an attribute A_X.

Definition. *Let T be a conditional FP-tree and $I=\{i_1,i_2,\ldots,i_m\}$ be the set of items in $T.header$. A frequent pairs array (FP-array) of T is a $(m-1)\times(m-1)$ matrix, where each element of the matrix corresponds to the counter of an ordered pair of items in I.*

Obviously, there is no need to set a counter for both item pairs (i_j,i_k) and (i_k,i_j). Therefore, we only store the counters for all pairs (i_k,i_j) such that $k<j$.

We use an example to explain the construction of the FP-array. In Fig. 1a, supposing that the minimum support is 20 percent, after the first scan of the original database, we sort the frequent items as b:5, a:5, d:5, g:4, f:2, e:2, c: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 FP-array $A_\emptyset$, as shown in Fig. 2a. All cells in the FP-array are initialized to 0.

According to the definition of an FP-array, in $A_\emptyset$, each cell is a counter of a pair of items. Cell $A_\emptyset[c,b]$ is the counter for itemset $\{c,b\}$, cell $A_\emptyset[c,a]$ is the counter for itemset $\{c,a\}$, and so forth. During the second scan for constructing $T_\emptyset$, for each transaction, all frequent items in the transaction are extracted. Suppose these items form itemset J. To insert J into $T_\emptyset$, the items in J are sorted according to the order in $T_\emptyset.header$. When we insert J into $T_\emptyset$, at the same time $A_\emptyset[i,j]$ is incremented by 1 if $\{i,j\}$ is contained in J. For instance, for the second transaction, $\{b,a,f,g\}$ is extracted (item h is infrequent) and sorted as b,a,g,f. This itemset is inserted into $T_\emptyset$ as usual and, at the same time, $A_\emptyset[f,b], A_\emptyset[f,a]$, $A_\emptyset[f,g], A_\emptyset[g,b]$, $A_\emptyset[g,a], A_\emptyset[a,b]$ are all incremented by 1. After the second scan, the FP-array $A_\emptyset$ contains the counts of all pairs of frequent items, as shown in Fig. 2a.

Next, the FP-growth method is recursively called to mine frequent itemsets for each item in $T_\emptyset.header$. 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 b,a,d for the conditional pattern base of g can be obtained. Sorting them according to their counts, we get b,d,a. Therefore, for each item i in $T_\emptyset$, the FP-array $A_\emptyset$ makes the first traversal of $T_\emptyset$ unnecessary and each $T_{\{i\}}$ 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 FP-array $A_{X\cup\{i\}}$ is calculated. During the construction of the new FP-tree $T_{X\cup\{i\}}$, the FP-array $A_{X\cup\{i\}}$ is filled. As an example, from the FP-tree in Fig. 1b, if the conditional FP-tree $T_{\{g\}}$ is constructed, the FP-array $A_{\{g\}}$ will be in Fig. 2b. This FP-array is constructed as follows: From the FP-array $A_\emptyset$, we know that the frequent items in the conditional pattern base of $\{g\}$ are, in descending order of their support, b,d,a. By following the linked list of g, from the first node, we get $\{b,d\}:2$, so it is inserted as $(b:2,d:2)$ into the new FP-tree $T_{\{g\}}$. At the same time, $A_{\{g\}}[b,d]$ is incremented by 1. From the second node in the linked list, $\{b,a\}:1$ is extracted and it is inserted as $(b:1,a:1)$ into $T_{\{g\}}$. At the same time, $A_{\{g\}}[b,a]$ is incremented by 1. From the third node in the linked list, $\{a,d\}:1$ is extracted and it is inserted as $(d:1,a:1)$ into $T_{\{g\}}$. At the same time, $A_{\{g\}}[d,a]$ is incremented by 1. Since there are no other nodes in the linked list, the construction of $T_{\{g\}}$ is finished and FP-array $A_{\{g\}}$ is ready to be used for construction of FP-trees at the next level of recursion. The construction of FP-arrays and FP-trees continues until the FP-growth method terminates.

Based on the foregoing discussion, we define a variant of the FP-tree structure in which, besides all attributes given in [14], an FP-tree also has an attribute, *FP-array*, which contains the corresponding FP-array.

2.3 Discussion

Let us analyze the size of an FP-array first. Suppose the number of frequent items in the first FP-tree $T_\emptyset$ is n. Then, the size of the associated FP-array is proportional to $\sum_{i=1}^{n-1} i = n(n-1)/2$, which is the same as the number of candidate large 2-itemsets in Apriori in [6]. The FP-trees constructed from the first FP-tree have fewer frequent items, so the sizes of the associated FP-arrays decrease. At any time when the space for an FP-tree is freed, so is the space for its FP-array.

There are some limitations for using the FP-array technique. One potential problem is the size of the FP-array. When the number of items in $T_\emptyset$ is small, the size of the FP-array is not very big. For example, if there are 5,000 frequent items in the original database and the size of an integer is 4 bytes, the FP-array takes only 50 megabytes or so. However, when n is large, $n(n-1)/2$ becomes an extremely large number. At this case, the FP-array technique will reduce the significance of the FP-growth method, since the method mines frequent itemsets without generating any candidate frequent itemsets. Thus, one solution is to simply give up the FP-array technique until the number of items in an FP-tree is small enough. Another possible solution is to

reduce the size of the FP-array. This can be done by generating a much smaller set of candidate large two-itemsets as in [25] and only store in memory cells of the FP-array corresponding to a two-itemset in the smaller set. However, in this paper, we suppose the main memory is big enough for all FP-arrays.

The FP-array technique works very well, especially when the data set is sparse and very large. The FP-tree for a sparse data set and the recursively constructed FP-trees will be big and bushy because there are not many shared common prefixes among the FIs in the transactions. The FP-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 FP-arrays.

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

Even for the FP-trees of sparse data sets, the first levels of recursively constructed FP-trees for the first items in a header table 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 items are unnecessary in the FP-array. As an example, in Fig. 2a, since b, a, and d are the first three items in the header table, the first two lines do not have to be calculated, thus saving counting time.

Note that the data sets (the conditional pattern bases) change during the different depths of the recursion. In order to estimate whether a data set 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 percent of the total number of nodes, we are most likely dealing with a dense data set. Otherwise, the data set is likely to be sparse.

If the data set appears to be dense, we do not calculate the FP-array for the next level of the FP-tree. Otherwise, we calculate the FP-array of each FP-tree in the next level, but the cells for the first several (we use 15 based on our experience) items in its header table are not calculated.

2.4 FPgrowth*: An Improved FP-Growth Method

Fig. 3 contains the pseudo code for our new method FPgrowth*. The procedure has an FP-tree T as parameter. T has attributes: $base$, $header$, and $FP\text{-}array$. $T.base$ contains the itemset X for which T is a conditional FP-tree, the attribute $header$ contains the header table, and $T.FP\text{-}array$ contains the FP-array A_X.

In *FPgrowth**, line 6 tests if the FP-array of the current FP-tree exists. If the FP-tree corresponds to a sparse data set, its FP-array exists, and line 7 constructs the header table of the new conditional FP-tree from the FP-array directly. One FP-tree traversal is saved for this item compared with the FP-growth method in [14]. 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 FP-array or just set $T_Y.FP\text{-}array$ as undefined.

3 FPmax*: Mining MFI's

In [12], we developed FPmax, another method that mines maximal frequent itemsets using the FP-tree structure. Since the FP-array technique speeds up the FP-growth method for sparse data sets, we can expect that it will be useful in FPmax too. This gives us an improved method, FPmax*. Compared to FPmax, in addition to the FP-array technique, the improved method FPmax* also has a more efficient maximality checking approach, as well as several other optimizations. It turns out that FPmax* outperforms FPmax for all cases we discussed in [12].

Procedure *FPgrowth**(T)

Input: A conditional FP-tree T

Output: The complete set of all FI's corresponding to T.

Method:

1. **if** T only contains a single branch B
2. **for each** subset Y of the set of items in B
3. output itemset $Y \cup T.base$ with count = smallest count of nodes in Y;
4. **else for each** i in $T.header$ **do begin**
5. output $Y = T.base \cup \{i\}$ with $i.count$;
6. **if** $T.FP\text{-}array$ is defined
7. construct a new header table for Y's FP-tree from $T.FP\text{-}array$
8. **else** construct a new header table from T;
9. construct Y's conditional FP-tree T_Y and possibly its FP-array A_Y;
10. **if** $T_Y \neq \emptyset$
11. call *FPgrowth**(T_Y);
12. **end**

Fig. 3. Algorithm FPgrowth*.

3.1 The MFI-Tree

Obviously, compared with FPgrowth*, the extra work that needs to be done by FPmax* is to check if a frequent itemset is maximal. The naive way to do this is during a postprocessing step. Instead, in FPmax, we introduced a global data structure, the *maximal frequent itemsets tree* (MFI-tree), to keep the track of MFIs. Since FPmax* is a depth-first algorithm, a newly discovered frequent itemset can only be a subset of an already discovered MFI. We therefore need to keep track of all already discovered MFIs. For this, we use the MFI-tree. A newly discovered frequent itemset is inserted into the MFI-tree, unless it is a subset of an itemset already in the tree. From experience, we learned that a further consideration for large data sets is that the MFI-tree will be quite large, and sometimes one itemset needs thousands of comparisons for maximality checking. Inspired by the way maximality checking is done in [11], in FPmax*, we still use the MFI-tree structure, but for each conditional FP-tree T_X, a small local 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 globally maximal, we only need to compare Y with the itemsets in M_X. This speeds up FPmax significantly.

Each MFI-tree is associated with a particular FP-tree. An MFI-tree resembles an FP-tree. There are two main differences between MFI-trees and FP-trees. In an FP-tree, each node in the subtree has three fields: item-name, count, and node-link. In an MFI-tree, the count is replaced by the *level* of the node. The level field is used for maximality checking in a way to be explained later. Another difference is that the header table in an FP-tree is constructed from traversing the previous FP-tree or using the associated FP-array, while the

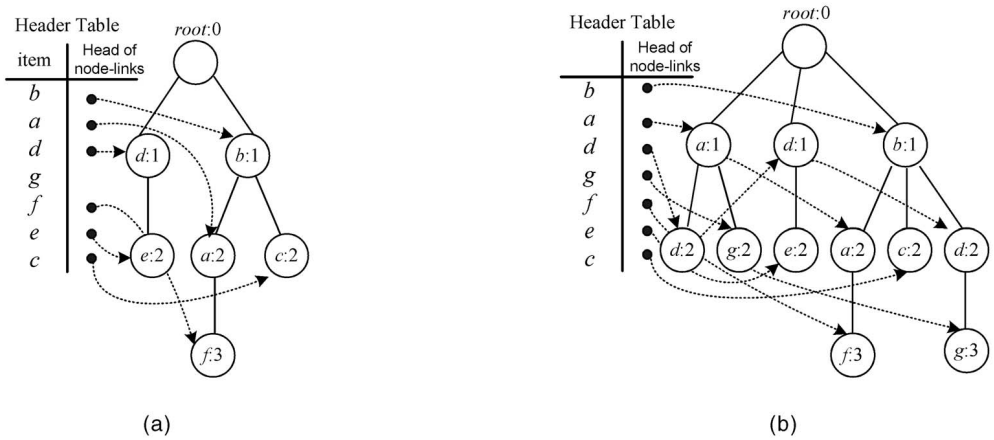

Fig. 4. Construction of maximal frequent itemset tree.

header table of an MFI-tree is constructed based on the item order in the table of the FP-tree it is associated with.

The insertion of an MFI into an MFI-tree is similar to the insertion of a frequent set into an FP-tree. Fig. 4 shows the insertions of all MFIs into an MFI-tree associated with the FP-tree in Fig. 1b. In Fig. 4, a node $x : l$ means that the node is for item x and its level is l. Fig. 4a shows the tree after inserting (b, c), (d, e), and (b, a, f). Since (b, a, f) shares prefix b with (b, c), only two new nodes for a and f are inserted. Fig. 4b is the tree after all MFIs have been inserted.

3.2 FPmax*

Fig. 5 gives algorithm FPmax*. In the figure, three attributes of T, $T.base$, $T.header$, and $T.FP\text{-}array$, are the same as the attributes we used in FPgrowth*. The first call of FPmax* will be for the FP-tree constructed from the original database, and it has an empty MFI-tree. Before a recursive call $FPmax^*(T, M)$, we already know from line 10 that the set containing $T.base$ and all items in T is not a subset of any existing MFI. During the recursion, if there is only one branch in T, the path in the branch 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-branch tree, then for each item i in $T.header$, 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 FP-array if it is defined or traverse T_X. Line 10 calls function *maximality_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 *maximality_checking* returns *false*, FPmax* will be called recursively, with (T_Y, M_Y). The implementation of function *maximality_checking* will be explained shortly.

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 branch B
2. insert B into M;
3. **else for each** i in $T.header$ **do begin**
4. set $Y = T.base \cup \{i\}$;
5. **if** $T.FP\text{-}array$ is defined
6. let $tail$ be the set of frequent items for i in $T.FP\text{-}array$
7. **else**
8. let $tail$ be the set of frequent items in i's conditional pattern base;
9. sort $tail$ in decreasing order of the items' counts;
10. **if not** *maximality_checking*$(Y \cup tail, M)$
11. construct Y's conditional FP-tree T_Y and possibly its FP-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;
15. **end**

Fig. 5. Algorithm FPmax*.

Note that, before and after calling *maximality_checking*, if $Y \cup tail$ is not a 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 branch, we can conclude that $Y \cup tail$ is frequent. Since $Y \cup tail$ was not a subset of any previously discovered MFI, it is maximal and will be inserted into M_Y.

The function *maximality_checking* works as follows: Suppose $tail = i_1 i_2, \ldots i_k$, in decreasing order of frequency according to $M.header$. By following the linked list of i_k, 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 toward the front of $tail$. The function maximality_checking returns *true* if $tail$ is a subset of an existing MFI, otherwise, *false* is returned.

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 preceding 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, B

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

In details, at line 12, when an MFI-tree M_{Y_j} for $Y_j = i_1 i_2 \dots i_j$ is created for the next call of FPmax*, we know that conditional FP-trees and conditional MFI-trees for $Y_{j-1} = i_1 i_2 \dots i_{j-1}$, $Y_{j-2} = i_1 i_2 \dots i_{j-2}$, $\dots$, $Y_1 = i_1$, and $Y_0 = \emptyset$ are all in memory. To make M_{Y_j} store all already found MFIs that contain Y_j, M_{Y_j} is initialized by extracting MFIs from $M_{Y_{j-1}}$. The initialization can be done by following the linked list for i_j from the header table of $M_{Y_{j-1}}$ and extracting the maximal frequent itemsets containing i_j. Each such found itemset I is sorted according to the order of items in $M_{Y_j}.header$ (the same item order as in $T_{Y_j}.header$) and then inserted into M_{Y_j}. On line 2, we have found a new MFI B in T_{Y_j}, so B is inserted into M_{Y_j}. Since $Y_j \cup B$ also contains $Y_{j-1}, \dots, Y_1$, Y_0 and the trees $M_{Y_{j-1}}, \dots, M_{Y_1}, M_{Y_0}$ are all in memory, to make these MFI-trees consistently store all already discovered MFIs that contain their corresponding itemset, for each $k = 0, 1, \dots, j$, the MFI $B \cup (Y_j - Y_k)$ is inserted into the corresponding MFI-tree M_{Y_k}. At the end of the execution of FPmax*, the MFI-tree M_{Y_0} (i.e., $M_\emptyset$) contains all MFIs mined from the original database. Since FPmax* is a depth-first algorithm, it is straightforward to show that the maximality checking is correct. Based on the correctness of the FPmax method, we can conclude that FPmax* returns all and only the maximal frequent itemsets in a given data set.

In FPmax*, we also used an optimization for reducing the number of recursive calls. Suppose that, at some level of the recursion, the item order in $T.header$ is $i_1, i_2, \dots, i_k$. Then, starting from i_k, 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_m where $m \leq k$, its maximal frequent itemset contains items $i_1, i_2, \dots, i_{m-1}$, i.e., all the items that have not yet called FPmax* recursively, these recursive calls can be omitted. This is because any frequent itemset found by such a recursive call must be a subset of $\{i_1, i_2, \dots, i_{m-1}\}$, thus, it could not be maximal.

3.3 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 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 its descendant FP-trees. From the experience of [14], we know the recursively constructed FP-trees are relatively small. If we store all the MFIs in memory, we can expect that the total size of those FP-trees is not greater than the final size of the MFI-tree for $\emptyset$. For the same reason, we can expect that the MFI-trees constructed from their parents are also small. During the mining, the MFI-tree for $\emptyset$ grows gradually, other small descendant MFI-trees will be constructed for recursive calls and then discarded after the call. Thus, we can conclude that the total memory requirement for running FPmax* on a data set is proportional to the sum of the size of the FP-tree and the MFI-tree for $\emptyset$.

The foregoing discussion is relevant for the case when we store all MFIs in memory throughout the mining process. Actually, storing all MFIs in memory is not necessary. Suppose the current FP-tree is T_X, the items in $T_X.header$ are $j_1, j_2, \dots, j_m$ and the current MFI-tree is M_X. During the current call of $FPmax^*(T_X, M_X)$, for each item j_k, by a recursive call of $FPmax^*(T_{X\cup\{j_k\}}, M_{X\cup\{j_k\}})$, many MFIs could be mined. For each MFI $X \cup \{j_k\} \cup Z$, the branch $\{j_k\} \cup Z$ will be inserted into M_X. However, recall that since none of the candidate MFIs $X \cup Z'$ generated from the rest of the current call contain j_k, it is sufficient to compare Z' with Z, instead of comparing Z' with $\{j_k\} \cup Z$. Therefore, in our implementation, when an MFI is inserted, we used an alternative approach where only the Z-part of the MFI is inserted into the MFI-trees. This leads to a reduction in the sizes of the MFI-trees. As an example, Fig. 6 shows the size-reduced MFI-tree for $\emptyset$ associated with the data set in Fig. 1. Shown in Fig. 4b is the complete MFI-tree for $\emptyset$ which has 12 nodes, while the size-reduced MFI-tree has only six nodes.

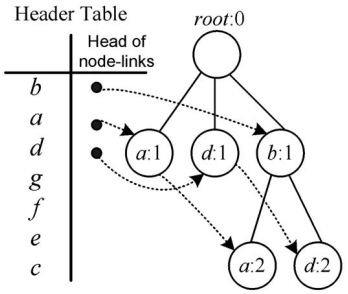

Fig. 6. Size-reduced maximal frequent itemset tree.

Our initial experimental results showed that the memory requirement of the size-reduced approach is drastically lowered compared to storing complete maximal frequent itemsets. Consequently, FPmax* in the experiments of Section 5 is implemented with the size-reduced approach.

4 FPclose: Mining CFI's

Recall that an itemset X is closed if none of the proper supersets of X have the same support. 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 closed frequent itemsets tree (CFI-tree), which is another variation on the FP-tree.

4.1 The CFI-Tree and Algorithm FPclose

As in algorithm FPmax*, a newly discovered frequent itemset can be a subset only of a previously discovered CFI. Like an MFI-tree, a CFI-tree depends on an FP-tree T_X and is denoted as C_X. The itemset X is represented as an attribute of T, $T.base$. The CFI-tree C_X always stores all already found CFIs containing itemset X and their counts. A newly found frequent itemset Y that contains X only needs to be compared with the CFIs in C_X. If there is no proper superset of Y in C_X with the same count as Y, the set Y is closed.

In a CFI-tree, each node in the subtree has four fields: item-name, count, node-link, and level. Here, level is still used for subset testing, as in MFI-trees. The count field is needed because when comparing 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 the same as the order of items in header table of its corresponding FP-tree.

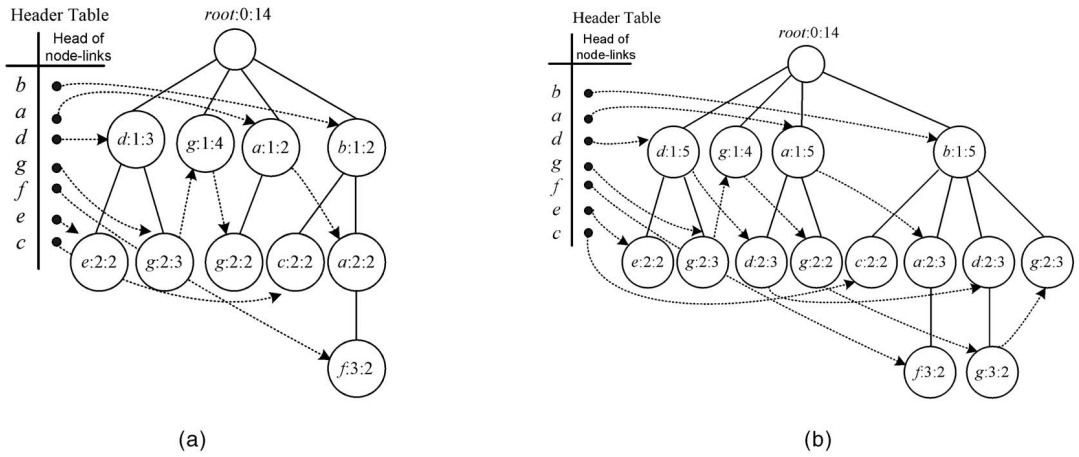

Fig. 7. Construction of closed frequent itemset tree.

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, but replaced by the maximal count up-to-date. Fig. 7 shows some snapshots of the construction of a CFI-tree with respect to the FP-tree in Fig. 1b. The item order in the two trees are the same because they are both for base $\emptyset$. Note that insertions of CFIs into the top level CFI-tree will occur only after recursive calls have been made. In the following example, the insertions would be performed during various stages of the execution, not in bulk as the example might suggest. In Fig. 7, a node $x : l : c$ means that the node is for item x, that its level is l and that its count is c. In Fig. 7a, after inserting the first six CFIs into the CFI-tree, we insert (d, g) with count 3. Since (d, g) shares the prefix d with (d, e), only node g is appended and, at the same time, the count for node d is changed from 2 to 3. The tree in Fig. 7b contains all CFIs for the data set in Fig. 1a.

Fig. 8 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 1) $T.base \subset X$ and 2) $T.base$ and X have the same count (this corresponds to optimization 4 in [30]). If there is only one single branch in T, the nodes and their counts in this single branch can be easily used to list the $T.base$-local closed frequent itemsets. These itemsets will be compared with the CFIs in C. If an itemset is closed, it is inserted into C. If the FP-tree T is not a single-branch tree, we execute line 6. Lines 9 to 12 use the FP-array if it is defined, otherwise, T is traversed. Lines 4 and 8 call function $closed\_checking(Y, C)$ to check whether a frequent itemset Y is closed. Lines 14 and 15 construct Y's conditional FP-tree T_Y and CFI-tree C_Y. 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 memory.

To list all candidate closed frequent itemsets from an FP-tree with only a single branch, suppose the branch with counts is $(i_1 : c_1, i_2 : c_2, \ldots, i_p : c_p)$, where $i_j : c_j$ means item i_j has the count c_j. Starting from i_1, comparing the counts of every two adjacent items $i_j : c_j$ and $i_{j+1} : c_{j+1}$, in the branch, if $c_j \neq c_{j+1}$, we list $i_1, i_2, \ldots, i_j$ as a candidate closed frequent itemset with count c_j.

CFI-trees are initialized similarly to MFI-trees, described in Section 3.2. The implementation of function *closed_checking* is almost the same as the implementation of function *maximality_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, we first 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. Here, the level of a node can also be used for saving comparison time, as in Section 3.2. 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. At the end of the execution of FPclose, the CFI-tree $C_\emptyset$ contains all CFIs mined

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 branch B
2. generate all CFI's from B;
3. **for each** CFI X generated
4. **if not** *closed_checking*(X, C)
5. insert X into C;
6. **else for each** i in $T.header$ **do begin**
7. set $Y = T.base \cup \{i\}$;
8. **if not** *closed_checking*(Y, C)
9. **if** $T.FP\text{-}array$ is defined
10. let $tail$ be the set of frequent items for i in $T.FP\text{-}array$
11. **else**
12. let $tail$ be the set of 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 possibly its FP-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;
18. **end**

Fig. 8. Algorithm FPclose.

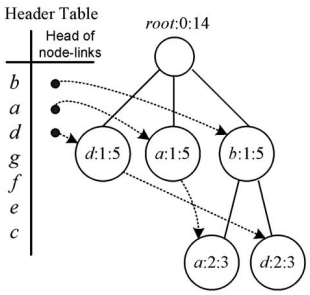

Fig. 9. Size-reduced closed frequent itemset tree.

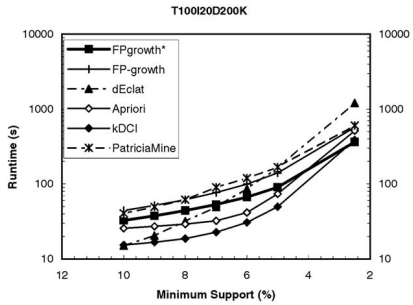

Fig. 10. Runtime of mining all FIs on *T100I20D200K*.

from the original database. The proof of correctness of FPclose is straightforward.

When implementing FPclose, we also used an optimization which was not used in CLOSET+. Suppose an FP-tree for itemset Y has multiple branches, the count of Y is c and the order of items in its header table is $i_1, i_2, \ldots, i_k$. Starting from i_k, for each item in the header table, we may need to do the work from line 7 to line 17. If for any item, say i_m, where $m \leq k$, $Y \cup \{i_1, i_2, \ldots, i_m\}$ is a closed itemset with count c, FPclose can terminate the current call. This is because *closed_checking* would always return *true* for the remaining items.

By a similar analysis as in Secion 3.3, we can estimate the total memory requirement for running FPclose on a data set. If the tree that contains all CFIs needs to be stored in memory, the algorithm needs space approximately equal to the sum of the size of the first FP-tree and its CFI-tree. In addition, as we did in Section 3.3 for mining MFIs, for each CFI, by inserting only part of the CFI into CFI-trees, less memory is used and the implementation is faster than the one that stores all complete CFIs. Fig. 9 shows the size-reduced CFI-tree for $\emptyset$ corresponding to the data set in Fig. 1. In this CFI-tree, only 6 nodes are inserted, instead of 15 nodes in the complete CFI-tree in Fig. 7b.

5 Experimental Evaluation and Performance Study

In 2003, the first *IEEE ICDM Workshop on Frequent Itemset Mining* (FIMI '03) [3] was organized. The goal of the workshop was to find the most efficient algorithms for mining frequent itemsets. The independent experiments conducted by the organizers compared the performance of three categories of algorithms for mining frequent itemsets. From the experimental results [3], [32], we can see that the algorithms described in this paper outperformed all the other algorithms submitted to the workshop for many cases. FPmax* and FPclose were recommended by the organizers as the best algorithms for mining maximal frequent itemsets and closed frequent itemsets, respectively.

In this paper, we report three sets of experiments in which the runtime, memory consumption, and scalability of FPgrowth*, FPmax*, and FPclose were compared with many well-known algorithms. These algorithms included SmartMiner, CLOSET+, an optimized version of Apriori, MAFIA, FP-growth, and GenMax.

Due to the lack of space, we only show part of our experimental results here. In the data sets, *T100I20D200K* is a synthetic and sparse data set. It was generated from the application of [1]. It has 200,000 transactions and 1,000 items. The average transaction length was 100 and the average pattern length was 20. Another two data sets, *connect* and *accidents*, are real data sets taken from [3]. The data set *connect* is compiled from game state information. It has 129 items and its average transaction length is 43. Data set *accidents* contains (anonymous) traffic accident data. It has 468 items and its average transaction length is 33.8. These two real data sets are both quite dense, so a large number of frequent itemsets will be mined even for very high values of minimum support.

When measuring running time and memory consumption, in each data set, we kept the number of transactions constant and varied the minimum support. We do not report experiments on the synthetic data sets where we varied the number of items or the avarage length of transactions. We found that when the number of items was changed, the number of *frequent* items was still determined by the minimum support. Similarly, we observed that the minimum support determined the average transaction length (once infrequent items were removed). Thus, the level of minimum support was found to be the principal influence on running time and memory consumption.

For scalability testing, we used the synthetic data sets and varied the number of transactions between 2×10^5 and 10^6.

The experiments were performed on a DELL Inspiron 8,600 laptop with a 1.6 GHz Pentium M and 1GB of RAM. The operating system was RedHat version 9, using Linux 2.4.20 and a gcc 3.2.2 C++ compiler. Both time and memory consumption of each algorithm running on each data set were recorded. Runtime was recorded by the "time" command and memory consumption was recorded by "memusage."

5.1 FI Mining

In the first set of experiments, we studied the performance of FPgrowth* by comparing it with the original FP-growth method [14], [15], kDCI [23], dEclat [35], Apriori [5], [6], and PatriciaMine [27]. To see the performance of the FP-array technique, we implemented the original FP-growth method on the basis of the paper [14]. The Apriori algorithm was implemented by Borgelt in [9] for FIMI '03. The source codes of the other algorithms were provided by their authors.

5.1.1 The Runtime

Fig. 10 shows the time of all algorithms running on *T100I20D200K*. In the figure, FPgrowth* is slower than kDCI, Apriori, and dEclat for high minimum support. For low minimum support, FPgrowth* becomes the fastest. The algorithm which was the fastest, dEclat, now becomes the slowest. The FP-array technique also shows its great

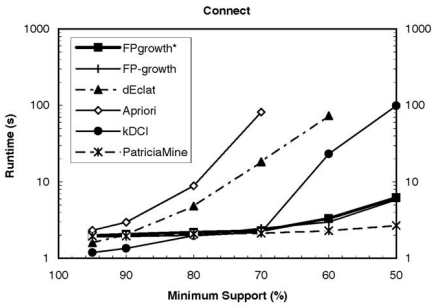

Fig. 11. Runtime of mining all FIs on *connect*.

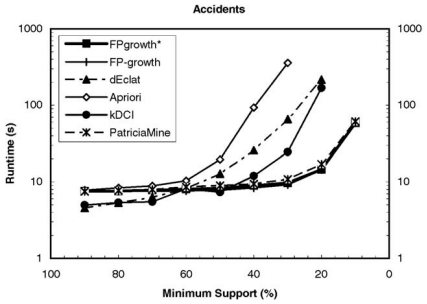

Fig. 12. Runtime of mining all FIs on *accidents*.

improvement on the FP-growth method. FPgrowth* is always faster than the FP-growth method and it is almost two times faster than the FP-growth method for low minimum support. When the minimum support is low, it means that the FP-tree is bushy and wide and the FP-array technique saves much time for traversing the FP-trees.

The results shown in Fig. 10 can be explained as follows: With sparse data sets, such as the synthetic ones, FPgrowth* constructs bushy and wide FP-trees. However, when the minimum support is high, not many frequent itemsets will be mined from the FP-trees. On the contrary, when the minimum support is low, the data set could have many frequent itemsets. Then, the time used for constructing FP-trees pays off. For Apriori, kDCI, and dEclat, when the minimum support is high, there are fewer frequent itemsets and candidate frequent itemsets to be produced. Thus, they only need to build small data structures for storing frequent itemsets and candidate frequent itemsets, which does not take much time. But, for low minimum support, considerable time is spent on storing and pruning candidate frequent itemsets. These operations take more time than mining frequent itemsets from the FP-trees. This is why in the experiments in Fig. 10, FPgrowth* is relatively slow for high minimum support and relatively fast for low minimum support when compared with the other methods.

Fig. 11 and Fig. 12 show the performance of all algorithms on two dense real data sets, *connect* and *accidents*. In Fig. 12, FPgrowth* is the fastest algorithm for low values of minimum support, and the curves for FP-growth and PatriciaMine almost totally overlap the curve for FPgrowth*.

In the two figures, one observation is that FPgrowth* does not outperform FP-growth method for dense data sets. This is consistent with our discussion in Section 2. We mentioned that when the FP-trees constructed from the data sets are compact, the time spent on constructing FP-arrays will be more than the time for traversing FP-trees, therefore, the FP-array technique is not applied on dense data sets.

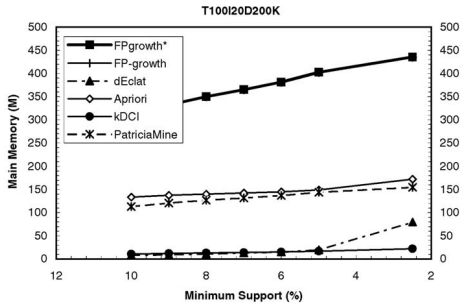

Fig. 13. Memory consumption of mining all FIs on *T100I20D200K*.

Another observation is that PatriciaMine has performance similar to the FPgrowth* method. This is because both algorithms are based on a prefix tree data structure. In Fig. 11, when the minimum support is low, FPgrowth* is even slower than PatriciaMine. This means the Patricia Trie data structure used in PatriciaMine saves not only the space but also the time for traversing FP-trees.

These three figures show that no algorithm is always the fastest. As discussed before, the sparsity/density of the data set has a significant well understood influence on the performance of the algorithms. The reasons for the difference in performance between Figs. 11 and 12 are less well understood. We believe that the data distributions in the data sets have a big influence on the performances of the algorithms. Unfortunately, the exact influence of the data distribution for each algorithm is still unknown. We also found that it is time-consuming to determine the data distribution before mining the frequent itemsets. Thus, it would be wise to choose a stable algorithm such as FPgrowth* to mine frequent itemsets. Note that FPgrowth* also is fairly stable with regard to the minimum support.

5.1.2 Memory Consumption

Fig. 13 shows the peak memory consumption of the algorithms on the synthetic data set. The FPgrowth* and the FP-growth method consume almost the same memory, their curves overlap again. In the figure, kDCI uses the lowest amount of memory when the minimum support is high. The algorithm dEclat also consumes far less memory than the other algorithms except kDCI.

We can see that FPgrowth* and the FP-growth method unfortunately use the maximum amount of memory. Their memory consumption is almost four times greater than the data set size. Since the FPgrowth* and FP-growth methods consume almost the same amount of memory, it means that the memory spent on the FP-array technique is negligible. The memory is mainly used by FP-trees constructed in the FP-growth method. The question of why the FP-growth method consumes so much memory when running on a synthetic data set can be answered as follows: In both figures, the minimum support is fairly low, so there are many frequent single items in the data sets. Therefore, wide and bushy trees have to be constructed for mining all frequent itemsets. Since the number of frequent single items stays almost the same when the minimum support changes, the sizes of the FP-trees remain almost the same, as we can see from the figure.

Comparing Fig. 10 with Fig. 13, we also can see that FPgrowth* and the FP-growth method still have good speed even when they have to construct big FP-trees.

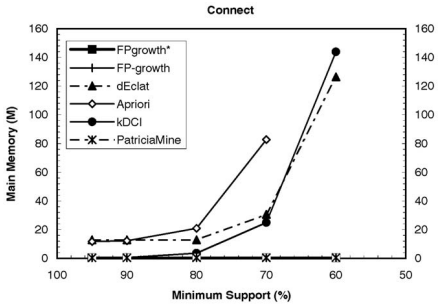

Fig. 14. Memory consumption of mining all FIs on *connect*.

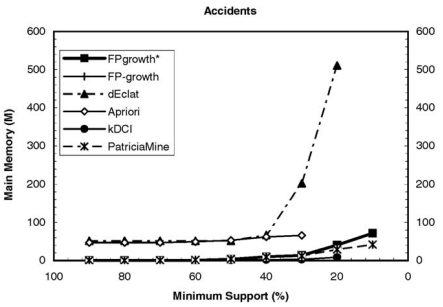

Fig. 15. Memory consumption of mining all FIs on *accidents*.

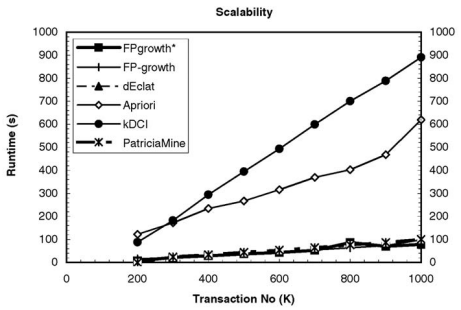

Fig. 16. Scalability of runtime of mining all FIs.

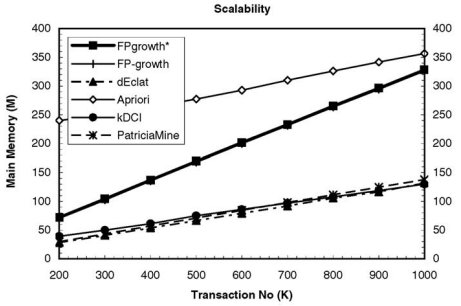

Fig. 17. Scalability of memory consumption of mining all FIs.

We also can see from the figures that PatriciaMine consumes less memory than FPgrowth*. This is because we implemented FPgrowth* by a standard trie. On the other hand, in PatriciaMine, the FP-tree structure was implemented as a Patricia trie, which will save some memory. In Figs. 14 and 15, the FP-tree structure shows great compactness. Now, the memory consumption of FPgrowth* is almost the same as that of the PatriciaMine and the FP-growth method. In the figures, we almost cannot see the curves for the PatriciaMine and the FP-growth method because they overlap the curves for FPgrowth*. Although not clearly visible in the figures, the experimental data show that, for high minimum support, the memory consumed by FPgrowth* is even smaller than the memory consumption of PatriciaMine, which means the Patricia trie needs more memory than the standard trie when the FP-trees are very compact.

From the figures, we also notice that when minimum support is low, the memory used by algorithms such as Apriori and kDCI increases rapidly, while the memory used by FPgrowth* and PatriciaMine does not change much. This is because algorithms such as Apriori and kDCI have to store a large number of frequent itemsets and candidate frequent itemsets. The number of itemsets that needs to be stored increases exponentially when the minimum support becomes lower. For FPgrowth*, FP-growth, and PatriciaMine, if the number of frequent single items does not change much when minimum support becomes lower, the sizes of FP-trees do not change much, either.

5.1.3 Scalability

Though a synthetic data set is not the most favorable for FPgrowth*, its parameters are easily adjustable. We therefore tested the scalability of all algorithms by running them on data sets generated from *T20I10*. The number of transactions in the data sets for Fig. 16 and Fig. 17 ranges from 200K to 1 million. In all data sets, the number of items is 1,000, the average transaction length is 20, the number of patterns used as generation seeds is 5,000, and the average pattern length is 10.

All algorithms ran on all data sets for minimum support 0.1 percent. Both runtime and memory consumption were recorded. Fig. 16 shows the speed scalability of all algorithms. In the figure, the curves for PatriciaMine, FPgrowth*, and FPgrowth overlap each other, which means that the three algorithms have almost the same scalability, which is also the best scalability. Runtime increases about four times when the size of data set increases five times, while runtime of algorithms such as kDCI increases about 10 times when the number of transactions increases from 200K to 1 million. Fig. 17 gives the memory scalability of all algorithms. In the figure, the curve for the FP-growth method overlaps the curve for FPgrowth* and the curve for dEclat overlaps the curve for PatriciaMine. The figure shows that memory consumption of FPgrowth* and FP-growth increases linearly when size of data sets changes. It also shows that the growth rates of FPgrowth* and FP-growth are not as good as other algorithms.

From the figures of runtime, memory consumption, and scalability of all algorithms, we can draw the conclusion that FP-growth* is one of the best algorithms for mining all frequent itemsets especially when the data sets are dense. For very sparse data sets, since FPgrowth* is an extension of the FP-growth method, its memory consumption is unfortunately very high. However, when the minimum support is low, FP-growth* is still one of the fastest algorithms, thanks to the FP-array technique. We also note as FIMI '03 [32] did that PatriciaMine is a good algorithm for mining all frequent itemsets.

5.2 MFI Mining

In our paper [12], we analyzed and tested 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 FP-array technique and the new *maximality_checking* function that we are using in FPmax*,

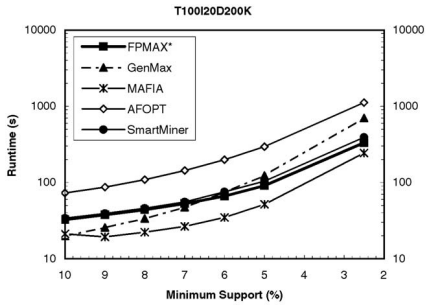

Fig. 18. Runtime of mining MFIs on *T100I20D200K*.

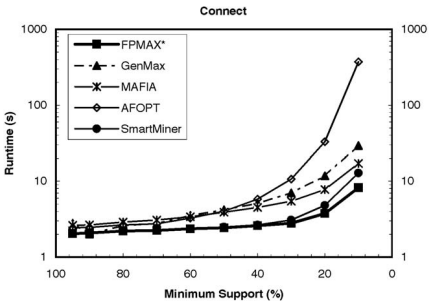

Fig. 19. Runtime of mining MFIs on *Connect*.

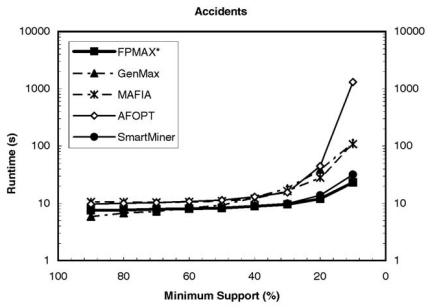

Fig. 20. Runtime of mining MFIs on *Accidents*.

in the second set of experiments, we compared FPmax* with MAFIA [10], AFOPT [19], GenMax [11], and SmartMiner [36]. FPmax* was implemented by storing reduced MFIs in MFI-trees, as explained in 3.3, to save memory and CPU time. Since the authors of SmartMiner implemented their algorithm in Java, we implemented SmartMiner ourselves in C++. The source codes of other algorithms were provided by their authors.

5.2.1 The Runtime

Fig. 18 gives the result for running the five algorithms on the data set *T100I20D200K*. In the figure, MAFIA is the fastest algorithm. SmartMiner is as fast as FPmax* for high minimum support, but slower than FPmax* for low minimum support. In *T100I20D200K*, since the average transaction length and average pattern length are fairly large, FPmax* has to construct bushy FP-trees from the data set, which can be seen from Fig. 21. The time for constructing and traversing the FP-trees dominates the whole mining time, especially for high minimum support. On the contrary, if MAFIA does not have much workload for high minimum support, few maximal frequent itemsets and candidate maximal frequent itemsets will be generated. However, as shown in Fig. 18, when the minimum support is low, FPmax* outperforms all other algorithms except MAFIA because now the construction of the large FP-tree

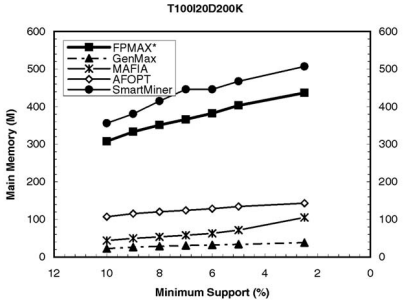

Fig. 21. Memory consumption of mining MFIs on *T100I20D200K*.

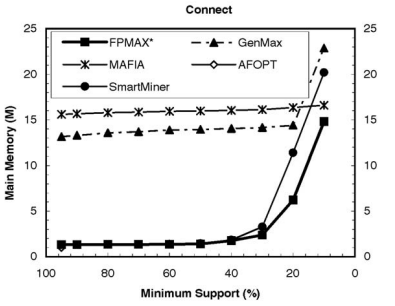

Fig. 22. Memory consumption of mining MFIs on *Connect*.

offers a big gain as there will be a large number of maximal frequent itemsets.

Fig. 19 and Fig. 20 show the experimental results of running the five algorithms on real data sets. In the figures, FPmax* shows the best performance on both data sets, for both high and low minimum support, benefiting from the great compactness of the FP-tree structure on dense data sets. SmartMiner has performance similar to FPmax* when minimum support is high, and is slower than FPmax* when minimum support is low.

All experiments on both synthetic and real data sets show that our *maximality_checking* function is indeed very effective.

5.2.2 Memory Consumption

Similar to the memory consumption for mining all frequent itemsets on synthetic data sets, FPmax* in Fig. 21 still uses much memory for mining maximal frequent itemsets. However, by comparing Fig. 21 with Fig. 13, we also can see that the amounts of memory consumed by FPmax* and FPgrowth* are very close. This means the MFI-trees constructed from the sparse data sets are fairly small, and the memory is still mainly used for constructing FP-trees. This is consistent with the explanation of why FPmax* is slower than MAFIA in Fig. 18.

Figs. 22 and 23 show the memory consumption of all algorithms running on data sets *Connect* and *Accidents*. Because of the compactness of FP-tree data structure for dense data sets, FPmax* consumes far less memory than for sparse data sets. In the two figures, the curves for SmartMiner overlap the curves for FPmax* for high minimum support. However, when the minimum support is low, SmartMiner consumes much more memory than FPmax*. The curve of AFOPT in Fig. 22 totally overlaps the curve for FPmax* and, in Fig. 23, FPMAX* and AFOPT consume similar amounts of memory. MAFIA uses a large amount of memory in Fig. 22 and it consumes the least memory in

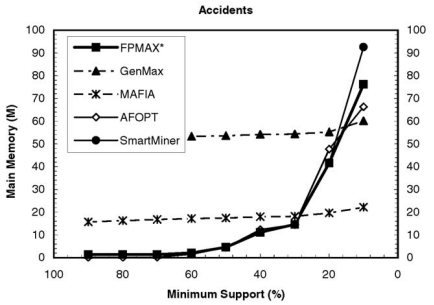

Fig. 23. Memory consumption of mining MFIs on *Accidents*.

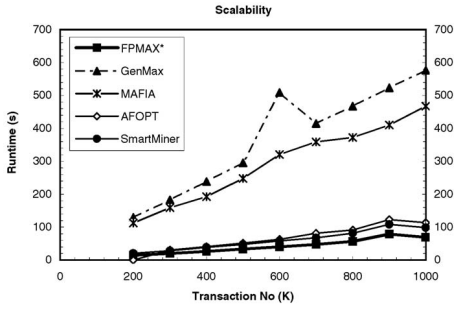

Fig. 24. Scalability of runtime of mining MFIs.

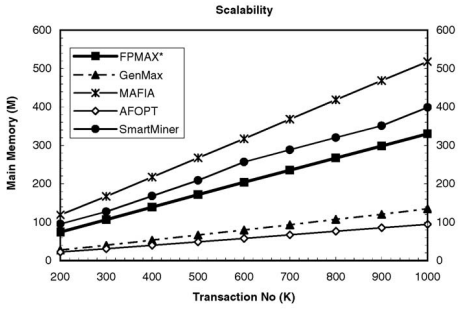

Fig. 25. Scalability of memory consumption of mining MFIs.

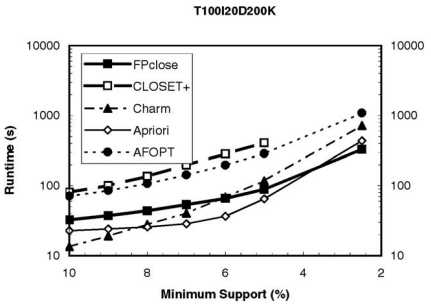

Fig. 26. Runtime of mining CFIs on *T100I20D200K*.

Fig. 23 when the minimum support is very low. For most cases, GenMax consumes more memory than FPmax*.

One observation from the two figures is that the memory consumption of FPmax* and GenMax increases exponentially when the minimum support becomes very low. This can be observed, for example, in Fig. 23. The increase happens because the algorithms have to store a large number of maximal frequent itemsets in memory, and the data structures such as MFI-trees become very large. We can also see that when the memory needed by the algorithms increases rapidly, their runtime also increases very fast. The pair of Fig. 19 and Fig. 22. is a typical example.

5.2.3 Scalability

Fig. 24 shows the speed scalability of all algorithms on synthetic data sets. The same data sets were used in Section 5.1 for testing the scalability of all algorithms for mining all frequent itemsets. Fig. 24 shows that FPmax* is also a scalable algorithm. Runtime increases almost five times when the data size increases five times. The figures also demonstrate that other algorithms have good scalability. No algorithms have exponential runtime increase when the data set size increases.

Fig. 25 shows that FPmax* possesses good scalability of memory consumption as well. Memory consumption grows from 76 megabytes to 332 megabytes when data size grows from 16 megabytes to 80 megabytes. All algorithms have similar scalability on synthetic data sets.

In conclusion, the experimental results of runtime and memory consumption of all algorithms show that our MFI-tree data structure and maximality-checking technique work extremely well. Though FPmax* is not the fastest algorithm for some cases and it sometimes consumes much more memory than other algorithms, overall, it still has very competitive performance when compared with four other excellent algorithms. This result is consistent with the results of FIMI '03 which showed FPmax* as one of the best algorithms for mining maximal frequent itemsets.

5.3 CFI Mining

In the third set of experiments, we compared the performances of FPclose, Charm [34], AFOPT [19], Apriori [5], [6], and CLOSET+ [30]. FPclose is implemented by storing reduced CFIs in CFI-trees, as explained in the end of Section 4. The executable code of CLOSET+ was downloaded from [4]. The algorithm Apriori for mining closed frequent itemsets was implemented by Borgelt in [9] for FIMI '03. In Borgelt's implementation, CFIs are computed by generating all FIs first, then removing nonclosed ones. The source codes of Charm and AFOPT were provided by their authors.

In order to show a clear distinction between FPclose and CLOSET+, in this set of experiments, we report the results of running the algorithms on five data sets. In the two previous sets of experiments, the two additional data sets did not expose any further trends, and the results were therefore not included there. Besides the runtime and memory consumption of all algorithms running on data sets *T100I20D200K*, *Connect*, and *Accidents*, we now also compare the performance of the algorithms running on two additional real data sets, *Kosarak* and *Pumsb**. Both data sets were taken from [3]. *Kosarak* contains (anonymous) click-stream data of a Hungarian online news portal. It has 41,270 items and its average transaction length is 8.1. Data set *pumsb** is produced from census data of Public Use Microdata Sample (PUMS). There are 2,088 items in the data set and the average transaction length of the data set is 50.5.

5.3.1 The Runtime

Fig. 26 shows the runtime of all algorithms on the synthetic data set. FPclose is slower than Apriori and CHARM only for high minimum support. When the minimum support is low, FPclose becomes the fastest algorithm. CLOSET+ is always the slowest one. FPclose is slower than Apriori and CHARM for high minimum support because FPclose spends much time constructing a bushy and wide FP-tree while the FP-tree yields only small number of closed frequent itemsets. On the contrary, Apriori and CHARM

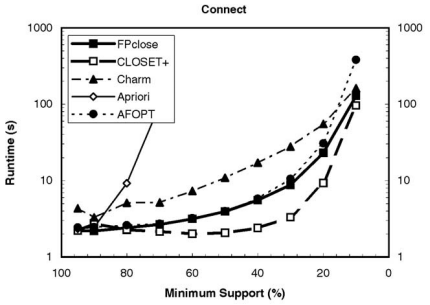

Fig. 27. Runtime of mining CFIs on *Connect*.

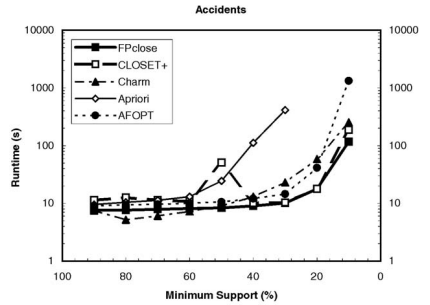

Fig. 28. Runtime of mining CFIs on *Accidents*.

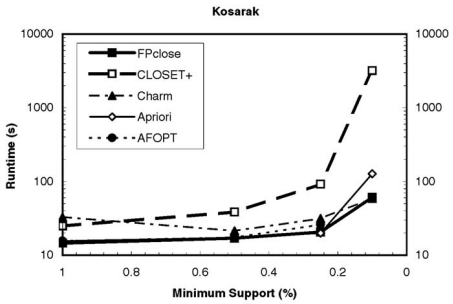

Fig. 29. Runtime of mining CFIs on *Kosarak*.

only need small sized data structures to store and generate candidate frequent itemsets and closed frequent itemsets. When the minimum support is low, FPclose is fast because its work on the first stage pays off, as many closed frequent itemsets will be generated. In this case, Apriori and other algorithms have to do a considerable amount of work to deal with a large number of candidate frequent itemsets.

When running the algorithms on real data sets, FPclose and CLOSET+ both show great speed. In Fig. 27, CLOSET+ is the fastest algorithm, which means that the strategies introduced in [30] work especially well for data set *connect*. FPclose is faster than CLOSET+ when running on all other data sets, as indicated in Figs. 28, 29, and 30. In Fig. 29, CLOSET+ even demonstrates the overall worst performance.

One of the reasons why FPclose is usually faster than CLOSET+ is that CLOSET+ uses a global tree to store already found closed frequent itemsets. When there is a large amount of frequent itemsets, each candidate closed frequent itemset has to be compared with many existing itemsets. In FPclose, on the contrary, multiple smaller CFI-trees are constructed. Consequently, a candidate closed frequent itemset only needs to be compared with small set of itemsets, saving a lot of time.

In Figs. 27, 28, 29, and 30, Apriori, which now has to construct big hash-trees for storing frequent itemsets and

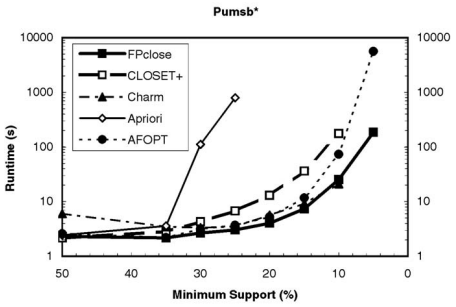

Fig. 30. Runtime of mining CFIs on *Pumsb**.

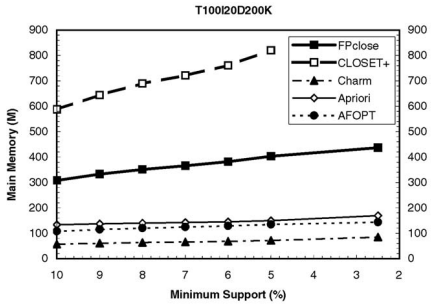

Fig. 31. Memory consumption of mining CFIs on *T100I20D200K*.

candidate frequent itemsets, always has bad performance. AFOPT and CHARM have similar performance, but they are both slower than FPclose.

5.3.2 Memory Consumption

Fig. 31 shows the peak memory consumption of the algorithms when running them on the synthetic data set. In Section 5.1 and Section 5.2, the memory consumption of FPgrowth* and FPmax* for synthetic data was high. Here, FPclose consumes much memory as well. By comparing Fig. 31 with Fig. 13, we can see that the amounts of memory consumed by FPclose and FPgrowth* are very close. This means the CFI-trees constructed from the sparse data sets are fairly small, and the memory is still mainly used for constructing FP-trees.

CLOSET+ uses the maximum amount of memory in Fig. 31. This is not surprising. Besides the memory consumed for the bushy and wide FP-trees for the synthetic data sets, it has to store a big tree for closedness testing as well. At the same time in FPclose, only reduced closed frequent itemsets were stored in small CFI-trees, as explained in Section 3.3.

The FP-tree and CFI-tree structure once again show great compactness on dense data sets. In Figs. 32, 33, 34, and 35, FPclose uses the least amount of memory. The only exception is in Fig. 33 where FPclose uses more memory than CHARM does. This happens when the minimum support is very low, although, FPclose still uses less memory than CLOSET+. In Fig. 32, for the minimum support 10 percent, CLOSET+ consumes eight times more memory than FPclose. The memory consumption of CLOSET+ is only less than that of Apriori.

5.3.3 Scalability

Figs. 36 and 37 show the scalability of all algorithms when running them on synthetic data sets. These data sets are the same that were used in Section 5.1 and Section 5.2.

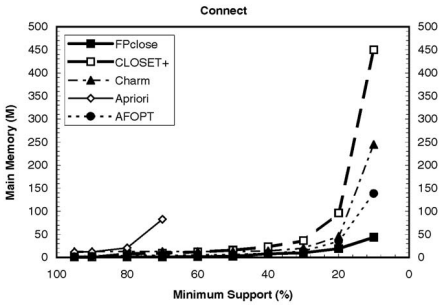

Fig. 32. Memory consumption of mining CFIs on *Connect*.

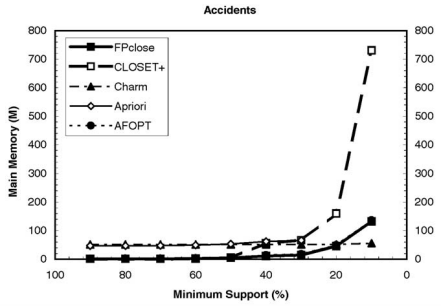

Fig. 33. Memory consumption of mining CFIs on *Accidents*.

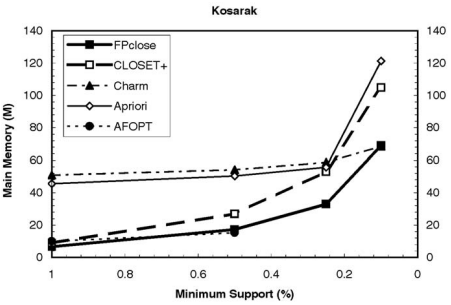

Fig. 34. Memory consumption of mining CFIs on *Kosarak*.

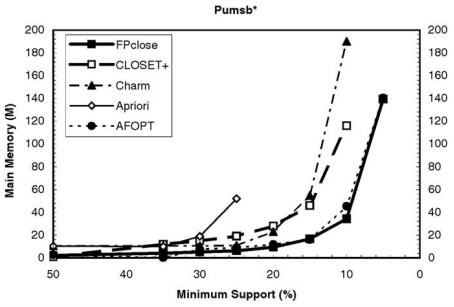

Fig. 35. Memory consumption of mining CFIs on *Pumsb**.

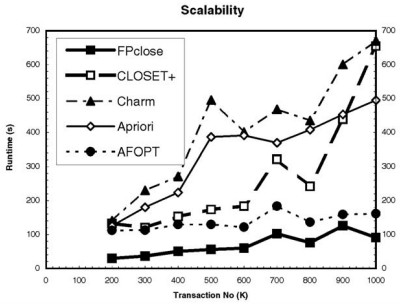

Fig. 36. Scalability of runtime of mining CFIs.

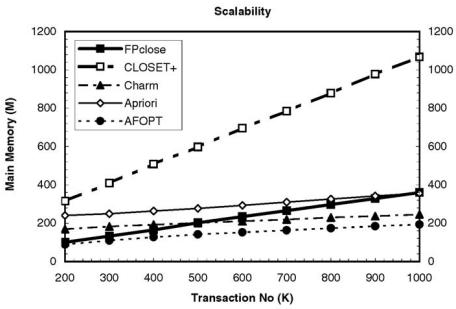

Fig. 37. Scalability of memory consumption of mining CFIs.

All the algorithms are scalable algorithms for runtime, according to Fig. 36. The difference is their growth rates. In the figure, we can see that FPclose has the smallest rate.

In Fig. 37, CLOSET+ shows the worst scalability with respect to main memory consumption, while FPclose has a reasonable scalability compared with the other algorithms. Main memory size increases four times when the size of synthetic data sets increases five times. AFOPT shows the best memory scalability.

In summary, though FPclose consumes a large amount of memory for very sparse data sets, it is still the fastest algorithm when the minimum support is low. For dense data sets, FPclose has a performance usually better than CLOSET+ for speed, and its memory consumption is far lower than the memory consumption of CLOSET+. Our CFI-tree data structure and the closedness testing technique are very efficient. The results of FIMI '03 that did not include CLOSET+ showed that FPclose is "an overall best algorithm" [32]. Our results in this paper also allow us to conclude that overall FPclose is one of the best algorithms for mining closed frequent itemsets, even when compared to CLOSET+.

6 Conclusions

We have introduced a novel FP-array technique that allows using FP-trees more efficiently when mining frequent itemsets. Our technique greatly reduces the time spent on traversing FP-trees, and works especially well for sparse data sets.

By incorporating the FP-array technique into the FP-growth method, the FPgrowth* algorithm for mining all frequent itemsets was introduced. Then we presented new algorithms for mining maximal and closed frequent itemsets. For mining maximal frequent itemsets, we extended our earlier algorithm FPmax to FPmax*. FPmax* not only uses the FP-array technique, but also an effective maximality checking approach. For the maximality testing, a variation on the FP-tree, called an MFI-tree was introduced for keeping track of all MFIs. In FPmax*, a newly found FI is always compared with a small set of MFIs that are stored in a local MFI-tree, thus making maximality-checking very efficient. For mining closed frequent itemsets we gave the FPclose algorithm. In this algorithm, a CFI-tree, another variation of the FP-tree, is used for testing the closedness of frequent itemsets. For all of our algorithms we have introduced several optimizations for further reducing their running time and memory consumption.

Both our experimental results and the results of the independent experiments conducted by the organizers of FIMI '03 show that FPgrowth*, FPmax*, and FPclose are among the best algorithms for mining frequent itemsets.

The algorithms are the fastest algorithms for many cases. For sparse data sets, the algorithms need more memory than other algorithms because the FP-tree structure needs a large amount of memory in these cases. However, the algorithms need less memory for dense data sets 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, the problem that FPgrowth*, FPmax* and FPclose consume lots of memory when the data sets are very sparse still needs to be solved. Consuming too much memory reduces the scalability of the algorithms. We notice from the experimental result in Section 5.1 that using a Patricia Trie to implement the FP-tree data structure could be a good solution for the problem. Furthermore, when the FP-tree is too large to fit in memory, the current solutions need a very large number of disk I/Os for reading and writing FP-trees onto secondary memory or generating many intermediate databases [15], which makes mining frequent itemsets too time-consuming. We are currently investigating techniques to reduce the number of disk I/Os and make frequent itemset mining scale to extremely large databases. Preliminary results are reported in [13].

REFERENCES

[1] http://www.almaden. ibm.com/cs/quest/syndata.html, 2003.
[2] http://www.almaden.ibm.com/cs/people/bayardo/resources.html, 2003.
[3] http://fimi.cs.helsinki.fi, 2003.
[4] http://www-sal.cs.uiuc.edu/~hanj/pubs/software.htm, 2004.
[5] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases," *Proc. ACM-SIGMOD Int'l Conf. Management of Data,* pp. 207-216, May 1993.
[6] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules," *Proc. Int'l Conf. Very Large Data Bases,* pp. 487-499, Sept. 1994.
[7] R. Agrawal and R. Srikant, "Mining Sequential Patterns," *Proc. Int'l Conf. Data Eng.,* pp. 3-14, Mar. 1995.
[8] R.J. Bayardo, "Efficiently Mining Long Patterns from Databases," *Proc. ACM-SIGMOD Int'l Conf. Management of Data,* pp. 85-93, 1998.
[9] C. Borgelt, "Efficient Implementations of Apriori and Eclat," *Proc. IEEE ICDM Workshop Frequent Itemset Mining Implementations,* CEUR Workshop Proc., vol. 80, Nov. 2003.
[10] D. Burdick, M. Calimlim, and J. Gehrke, "MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases," *Proc. Int'l Conf. Data Eng.,* pp. 443-452, Apr. 2001.
[11] K. Gouda and M.J. Zaki, "Efficiently Mining Maximal Frequent Itemsets," *Proc. IEEE Int'l Conf. Data Mining,* pp. 163-170, 2001.
[12] G. Grahne and J. Zhu, "High Performance Mining of Maximal Frequent Itemsets," *Proc. SIAM Workshop High Performance Data Mining: Pervasive and Data Stream Mining,* May 2003.
[13] G. Grahne and J. Zhu, "Mining Frequent Itemsets from Secondary Memory," *Proc. IEEE Int'l Conf. Data Mining,* pp. 91-98, Nov. 2004.
[14] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation," *Proc. ACM-SIGMOD Int'l Conf. Management of Data,* pp. 1-12, May 2000.
[15] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach," *Data Mining and Knowledge Discovery,* vol. 8, no. 1, pp. 53-87, 2004.
[16] J. Han, J. Wang, Y. Lu, and P. Tzvetkov, "Mining Top-K Frequent Closed Patterns without Minimum Support," *Proc. Int'l Conf. Data Mining,* pp. 211-218, Dec. 2002.
[17] M. Kamber, J. Han, and J. Chiang, "Metarule-Guided Mining of Multi-Dimensional Association Rules Using Data Cubes," *Proc. ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining,* pp. 207-210, Aug. 1997.
[18] D. Knuth, *Sorting and Searching.* Reading, Mass.: Addison Wesley, 1973.
[19] G. Liu, H. Lu, J.X. Yu, W. Wei, and X. Xiao, "AFOPT: An Efficient Implementation of Pattern Growth Approach," *Proc. IEEE ICDM Workshop Frequent Itemset Mining Implementations,* CEUR Workshop Proc., vol. 80, Nov. 2003.
[20] H. Mannila and H. Toivonen, "Levelwise Search and Borders of Theories in Knowledge Discovery," *Data Mining and Knowledge Discovery,* vol. 1, no. 3, pp. 241-258, 1997.
[21] H. Mannila, H. Toivonen, and I. Verkamo, "Efficient Algorithms for Discovering Association Rules," *Proc. ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining,* pp. 181-192, July 1994.
[22] H. Mannila, H. Toivonen, and I. Verkamo, "Discovery of Frequent Episodes in Event Sequences," *Data Mining and Knowledge Discovery,* vol. 1, no. 3, pp. 259-289, 1997.
[23] S. Orlando, C. Lucchese, P. Palmerini, R. Perego, and F. Silvestri, "kDCI: A Multi-Strategy Algorithm for Mining Frequent Sets," *Proc. IEEE ICDM Workshop Frequent Itemset Mining Implementations,* CEUR Workshop Proc., vol. 80, Nov. 2003.
[24] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering Frequent Closed Itemsets for Association Rules," *Proc. Int'l Conf. Database Theory,* pp. 398-416, Jan. 1999.
[25] J. Park, M. Chen, and P. Yu, "Using a Hash-Based Method with Transaction Trimming for Mining Association Rules," *IEEE Trans. Knowledge and Data Eng.* vol. 9, no. 5, pp. 813-825, 1997.
[26] J. Pei, J. Han, and R. Mao, "CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets," *Proc. ACM SIGMOD Workshop Research Issues in Data Mining and Knowledge Discovery,* pp. 21-30, May 2000.
[27] A. Pietracaprina and D. Zandolin, "Mining Frequent Itemsets Using Patricia Tries," *Proc. IEEE ICDM Workshop Frequent Itemset Mining Implementations,* CEUR Workshop Proc., vol. 80, Nov. 2003.
[28] A. Savasere, E. Omiecinski, and S. Navathe, "An Efficient Algorithm for Mining Association Rules in Large Databases," *Proc. Int'l Conf. Very Large Data Bases,* pp. 432-443, Sept. 1995.
[29] H. Toivonen, "Sampling Large Databases for Association Rules," *Proc. Int'l Conf. Very Large Data Bases,* pp. 134-145, Sept. 1996.
[30] J. Wang, J. Han, and J. Pei, "CLOSET+: Searching for the Best Strategies for Mining Frequent Closed Itemsets," *Proc. Int'l Conf. Knowledge Discovery and Data Mining,* pp. 236-245, Aug. 2003.
[31] D. Xin, J. Han, X. Li, and B.W. Wah, "Star-Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration," *Proc. Int'l Conf. Very Large Data Bases,* pp. 476-487, Sept. 2003.
[32] *Proc. IEEE ICDM Workshop Frequent Itemset Mining Implementations,* B. Goethals and M.J. Zaki, eds., CEUR Workshop Proc., vol. 80, Nov. 2003, http://CEUR-WS.org/Vol-90.
[33] M.J. Zaki, "Scalable Algorithms for Association Mining," *IEEE Trans. Knowledge and Data Mining,* vol. 12, no. 3, pp. 372-390, 2000.
[34] M.J. Zaki and C. Hsiao, "CHARM: An Efficient Algorithm for Closed Itemset Mining," *Proc. SIAM Int'l Conf. Data Mining,* pp. 457-473, Apr. 2002.
[35] M.J. Zaki and K. Gouda, "Fast Vertical Mining Using Diffsets," *Proc. ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining,* pp. 326-335, Aug. 2003.
[36] Q. Zou, W.W. Chu, and B. Lu, "SmartMiner: A Depth First Algorithm Guided by Tail Information for Mining Maximal Frequent Itemsets," *Proc. IEEE Int'l Conf. Data Mining,* Dec. 2002.

Gösta Grahne recieved the PhD degree from the University of Helsinki and was a postdoctoral fellow at the University of Toronto. He is currently an associate professor in the Department of Computer Science, Concordia University. His research interests include database theory, data mining, database engineering, semistructured databases, and emerging database environments. He is the author of more than 50 papers in various areas of database technology. He is a member of the IEEE and ACM.

Jianfei Zhu received the MEng degree in computer science from Zhejiang University, China, in 1995 and the PhD degree in computer science from Concordia University, Canada, in 2004. He is currently a software engineer at Google Inc. He was a senior lecturer at Zhejiang University, China, and has taught many computer science courses at Zhejiang University and Concordia University. His research interests include data mining, data warehousing, online analytical processing, and information retrieval. He is a member of the ACM and a student member of the IEEE.