Discovery of maximum length frequent itemsets

Tianming Hu[a,*], Sam Yuan Sung[b], Hui Xiong[c], Qian Fu[d]

[a]*DongGuan University of Technology, China*

[b]*South Texas College, USA*

[c]*Rutgers, the State University of New Jersey, USA*

[d]*National University of Singapore, Singapore*

Abstract

The use of frequent itemsets has been limited by the high computational cost as well as the large number of resulting itemsets. In many real-world scenarios, however, it is often sufficient to mine a small representative subset of frequent itemsets with low computational cost. To that end, in this paper, we define a new problem of finding the frequent itemsets with a maximum length and present a novel algorithm to solve this problem. Indeed, maximum length frequent itemsets can be efficiently identified in very large data sets and are useful in many application domains. Our algorithm generates the maximum length frequent itemsets by adapting a pattern fragment growth methodology based on the FP-tree structure. Also, a number of optimization techniques have been exploited to prune the search space. Finally, extensive experiments on real-world data sets validate the proposed algorithm.

Key words: Association analysis, Frequent itemsets, Maximum length frequent itemsets, FP-tree, Data mining

1 Introduction

In 1993, Agrawal et al. [2] first proposed the problem of finding frequent itemsets in their association rule mining model. Indeed, finding frequent itemsets

* Corresponding author.
Email address: `tmhu05@gmail.com` (Tianming Hu).

plays an important role in the field of data mining. Frequent itemsets are essential for many data mining problems, such as the discovery of association rules [3,12], data correlations [20,21], and sequential patterns [18,23].

The frequent itemset mining problem can be formally stated as follows: Let I be a set of distinct items. Each transaction T in database D is a subset of I. We call $X \subseteq I$ an itemset. An itemset with k items is called a k-itemset. The support of X, denoted by $supp(X)$, is the number of transactions containing X. If $supp(X)$ is no less than a user-specified minimum support ε, X is called a frequent itemset. Let FI denote the set of all frequent itemsets. X is closed if it has no proper superset with the same support. Let FCI denote the set of all frequent closed itemsets [16,25]. X is called a maximal frequent itemset if it has no proper superset that is frequent. The set of all maximal frequent itemsets is denoted by MFI [1,14,4,6,9,10]. X is a maximum length frequent itemset if X contains a maximum number of items in FI. Formally, it can be defined as follows: Let D be a transaction database over a set of distinct items I. Given a user-specified minimum support ε, an itemset X is a maximum length frequent itemset if $supp(X) \geq \varepsilon$ and for all itemset Y, if $supp(Y) \geq \varepsilon$ then $|X| \geq |Y|$, where $|Y|$ and $|X|$ denote the number of items contained in Y and X respectively.

Let LFI denote the set of all maximum length frequent itemsets. Apparently, any maximum length frequent itemset is a maximal frequent itemset. Thus we have the following relationship: LFI $\subseteq$ MFI $\subseteq$ FCI $\subseteq$ FI.

In many real world applications, the use of FI, MFI, and FCI has been limited by the high computational cost as well as the large number of resulting itemsets. Instead, it is often necessary to mine a small representative subset of frequent itemsets, such as LFI. Let us consider a case where a travel company is to propose a new tour package to some candidate places. The company conducts a survey of its customers to find their preferences among these places, i.e., which places they want to visit. Suppose that the company wants the package to satisfy the following requirements: a) the number of customers taking this tour should be no less than a certain number, for instance, 20 (quantity requirement), and b) the profit per customer is maximized (quality requirement). Here, we assume that the profit per customer is proportional to the number of places in the package. In addition, a customer is assumed to be low cost awareness, i.e., he/she will not pay for the package if the package contains the places he/she does not want to visit. This problem can be solved by finding LFI from the survey data having support no less than 20. Places in a maximum length frequent itemset form a desired package.

There exist many analogous problems. For instance, an insurance company may want to design an insurance package to attract a sufficient number of customers and maximize the number of insured subjects. A supermarket may want to design a binding sales plan to maximize the number of items purchased together by a sufficient number of customers. Similar problems also occur in web access pattern mining, where access sessions (webpages clicked) are modeled as transactions and webpages as items. Here maximum length patterns refer to those maximum number of pages a user visits in the search for the products. By studying these patterns, e.g., how many pages and which kind of the last page a user is willing to try before running out of patience and leaving the website, the company may better organize its website to facilitate the online service to its potential customers.

Another application of LFI is transaction clustering. A frequent itemset represents something common to many transactions in a cluster. Therefore, it is a natural way to use frequent itemsets for clustering. In [5,8] documents covering the same frequent itemset were put into the same cluster. Note that LFI is a special kind of frequent itemsets with maximum length. Intuitively, transactions sharing more items have a larger likelihood of belonging to the same cluster and hence it is reasonable to use LFI for transaction clustering.

Indeed, in this paper, we address the problem of mining maximum length frequent patterns (LFI). LFI can be efficiently identified even in very large databases because the number of maximum length frequent itemsets is usually very small, and may even be 1. In practice, we have observed that the FP-growth method [11] has advantages in mining long frequent itemsets. Since our goal is to find maximum length frequent itemset, those shorter ones are not of interest and need not be generated. Therefore, we developed two algorithms: LFIMiner and LFIMiner_ALL by adapting a pattern fragment growth methodology based on the FP-tree structure. For these two algorithms, LFIMiner was designed for mining only one maximum length frequent itemset and LFIMiner_ALL was designed for mining all maximum length frequent itemsets. A number of optimization techniques have also been exploited in our algorithms to prune the search space. As demonstrate by our experimental results on real-world data sets, LFIMiner and LFIMiner_ALL are computationally efficient and are scalable with respect to the number of transactions.

Overview. The remainder of the paper is organized as follows. Section 2 introduces related work on frequent itemset mining. In Section 3, we first give a brief introduction on the FP-tree structure and the FP-growth algorithm, and then describe some variants of the FPMAX algorithm [10]. The LFIMiner and LFIMiner_ALL algorithms are presented at the end of this section. The experimental results are reported in Section 4, which also includes an extensive study of various components of LFIMiner and a comparison with the variants of MAFIA [6] and FPMAX on real-world datasets. Section 5 concludes this paper with a summary and a discussion of future work.

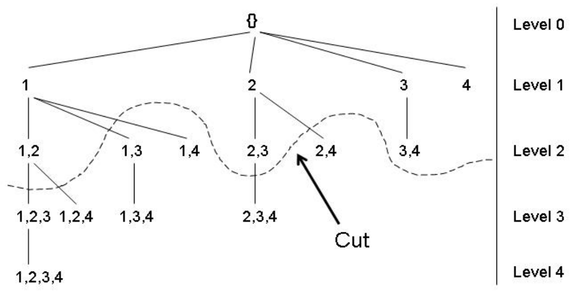


Fig. 1. Subset lattice over four items for the given order of 1, 2, 3, 4.

2 Preliminaries and related work

In this section, we first describe the conceptual framework of the item subset lattice on which many algorithms are based. Then we review some related research on frequent itemset mining.

2.1 Preliminaries

Most of the algorithms for mining frequent itemsets can be described using the item subset lattice framework [1,4,6,9]. This lattice shows how sets of items are completely enumerated in a search space. Assume there is a total lexicographic ordering $\leq_L$ of all items in the database. This ordering is used to enumerate the item subset lattice (search space). A particular ordering affects the item relationships in the lattice but not its completeness. Fig. 1 shows a sample of a complete subset lattice for four items. The lattice can be traversed in a breadth-first way, a depth-first way or some other way according to a heuristic. The problem of finding the frequent itemsets in the database can be viewed as finding a cut through this lattice so that all those tree nodes above the cut are frequent itemsets, while all those below are infrequent.

2.2 Related work

The Apriori algorithm [2,3] is a classic algorithm for finding frequent itemsets and most of algorithms are its variants. It uses frequent itemsets at level k to explore those at level $k+1$, which needs one scan of the database. Besides, it employs the heuristic that all nonempty subsets of a frequent itemset must also be frequent, which prunes unpromising candidates to narrow down the search space.

Apriori is based on the horizontal format of the database representation, in which a transaction is represented as an item list. An alternative way is to represent a database in vertical format, i.e., each item is associated with a set of transaction identifiers (TIDs) that include the item. As a representative in this group, VIPER [17] uses a vertical bit-vector with compression to store intermediate data during execution and performs counting with a vertical TID-list approach.

FP-growth [11] is a fundamentally different algorithm from the Apriori-like algorithms. The efficiency of Apriori-like algorithms suffers from the exponential enumeration of candidate itemsets and repeated database scans at each level for support check. To diminish these weaknesses, the FP-growth algorithm finds frequent itemsets without candidate set generation and records the database into a compact FP-tree structure to avoid repeated database scans. Due to the savings of storing the database in main memory, the FP-growth algorithm achieves great performance gains against Apriori-like algorithms. However, it is not scalable to very large databases, due to the requirement that the FP-trees fit in the main memory.

Representative breadth-first algorithms for mining MFI include Pincer-Search [14] and Max-Miner [4]. The former combines both the bottom-up and top-down searches. The latter uses lookahead to prune branches from the itemset lattice by quickly identifying long frequent itemsets.

The DepthProject algorithm [1] searches the itemset lattice in a depth-first manner to find MFI. To reduce search space, it also uses dynamic reordering of children nodes, superset pruning, an improved counting method and a projection mechanism. MAFIA [6] uses a vertical format to represent the database, which allows efficient support counting and is said to enhance the effect of lookahead pruning in general. Unlike DepthProject and MAFIA, GenMax [9] returns the exact MFI. It utilizes a backtracking search for efficiently enumerating all maximal patterns. Besides, it represents the database in a vertical TIDset format like Viper and uses diffset [24] propagation to perform fast support counting. FPMAX [10], as an extension of the FP-growth algorithm, utilizes a novel Maximal Frequent Itemset tree (MFI-tree) structure to keep track of all maximal frequent itemsets. Experimental results showed that FP-MAX has comparable performance with MAFIA and GenMax.

3 Mining LFI with FP-tree

In this section, we first introduce the FP-tree structure and the FP-growth algorithm, which are the bases of our algorithm. Then we describe our variant of the FPMAX algorithm. After discussing the methods to prune the search space, we present the LFIMiner algorithm, which integrates these methods to realize performance gains. The LFIMiner_ALL algorithm is presented at the end of this section.

3.1 FP-tree and the FP-growth algorithm

FP-tree is a compact data structure used by FP-growth to store the information about frequent itemsets in a database. The frequent items of each transaction are inserted into the tree in their frequency descending order. Compression is achieved by building the tree in such a way that overlapping transactions are represented by sharing common prefixes of the corresponding branches. A header table is associated with the FP-tree for facilitating tree traversal. Items are sorted in the header table in frequency descending order. Each row in the header table represents a frequent item, containing a head of node-link that links all the corresponding nodes in the tree.

Unlike Apriori-like algorithms which need several database scans, the FP-growth algorithm needs only two database scans. The first scan collects the set of frequent items. The second scan constructs the initial FP-tree, which records the information of the original database. For the example database in Fig. 2, after the first scan, the set of frequent items, $\{(b:6),(c:6),(a:4),(e:4),(d:2)\}$ (sorted in frequency descending order, minimum support is 2), is derived. In the second scan, each transaction is inserted into the tree. The scan of the first two transactions extracts their frequent items and constructs the first two branches of the tree: $\{(c:1),(e:1)\}$ and $\{(b:1),(c:1),(a:1),(e:1)\}$. For the third transaction, since its frequent item list $\{b,c,e,d\}$ shares a common prefix $\{b,c\}$ with the existing path $\{b,c,a,e\}$, the count of each node along the prefix is incremented by 1, and one new node $(e:1)$ is created and linked as a child of node $(c:2)$ and another new node $(d:1)$ is created and linked as a child of node $(e:1)$. Fig. 2 shows the initial FP-tree constructed after scanning all the transactions.

The FP-growth algorithm is based on the following principle: Let X and Y be two itemsets in database D, B be the set of transactions in D containing X. Then the support of $X \cup Y$ in D is equivalent to the support of Y in B. B is called the conditional pattern base of X. Given an item in the header table, the FP-growth algorithm constructs a new FP-tree according to its conditional pattern base, and mines this FP-tree recursively. Let us examine the mining process based on the FP-tree shown in Fig. 2. We start from the bottom of the header table. For item d, it derives a frequent itemset $(d:2)$ and two paths in the FP-tree: $\{(b:1),(c:1),(e:1)\}$ and $\{(c:1),(a:1)\}$, which constitute d's conditional pattern base. An FP-tree constructed from this conditional pattern base, called d's conditional FP-tree, has only one

TID	Itemset
100	c, e
200	a, b, c, e
300	b, c, d, e
400	b, c
500	a, b
600	b, f
700	b
800	a, c, e
900	a, c, d

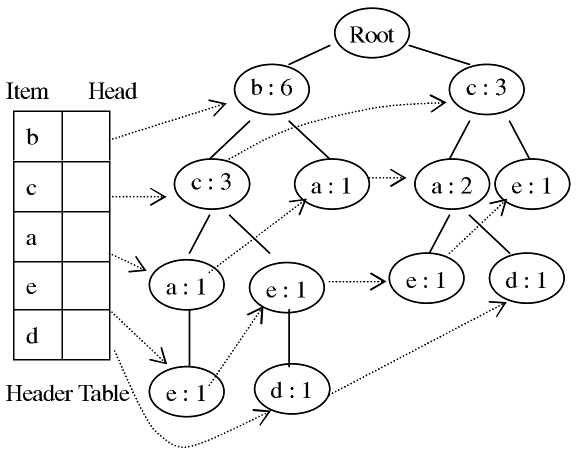


Fig. 2. FP-tree for the example database.

branch $\{(c : 2)\}$. Therefore only one frequent itemset $(cd : 2)$ is derived. The exploration for frequent itemsets associated with item d is terminated. Then one can continue to explore item e. For more information about the FP-tree and FP-growth algorithm, readers are referred to [11].

3.2 The FPMAX_LO algorithm

Based on the FP-growth algorithm, one can find all frequent itemsets. In order to solve our problem, however, some modifications are required to guarantee that the frequent itemset generated by our algorithm has the maximum length. We constructed a simple algorithm named FPMAX_LO (Longest Only) by extending either the FP-growth or the FPMAX algorithm. FPMAX_LO is shown in Fig. 3. Like FP-growth, FPMAX_LO is recursive. The initial FP-tree constructed from the two scans of the database is passed on as the parameter of the first call of the algorithm. The item list $Head$, initialized to be empty, contains the items whose conditional FP-tree will be constructed from its conditional pattern base and will then be mined recursively. Before recursive call to FPMAX_LO, we already know that the combination set of $Head$ and the items in the FP-tree is longer than the longest frequent itemset found so far (guaranteed by line (7)). Thus if there is only one single path in the FP-tree, the items in this path, together with $Head$, constitute a longer frequent itemset. If the FP-tree is not a single-path tree, then for each item in the header table, append the item to $Head$, construct the conditional pattern base of the new $Head$, and check in line (7) whether the combination set of $Head$ with all frequent items $Tail$ in the conditional pattern base is longer than the longest frequent itemset so far. If yes, we construct the conditional FP-tree based on the conditional pattern base and explore this tree recursively.

```
Input: T: an FP-tree
Global:
lfi: the longest frequent itemset found so far
Head: a list of items
Tree: the initial FP-tree
Output: The lfi that is a maximum length frequent itemset
Method: Call FPMAX_LO (Tree).
Procedure FPMAX_LO (T) {
(1)   IF T only contains a single path P
(2)   THEN update lfi with $Head \cup P$;
(3)   ELSE FOR EACH item i in header table of T DO {
(4)           Append i to Head;
(5)           Construct Head's conditional pattern base;
(6)           Tail = {frequent items in Head's conditional pattern base};
(7)           IF Length($Head \cup Tail$) > Length(lfi)
(8)           THEN {
(9)                   Construct Head's conditional FP-tree $T_{Head}$;
(10)                  FPMAX_LO ($T_{Head}$); }
(11)          Remove i from Head. }  //end of for each
}     // end of procedure
```

Fig. 3. The FPMAX_LO algorithm.

3.3 Pruning the search space

FPMAX_LO runs without any pruning. To realize better performance, we added pruning. It has three main operations: construct the conditional pattern base, find all frequent items in the conditional pattern base, and construct the conditional FP-tree. For the first operation, conditional transactions that are not long enough cannot be useful for generating a longer frequent itemset and should be removed. For the second operation, conditional transactions without enough frequent items cannot contribute to forming a longer frequent itemset and should be trimmed. For the last one, we can reorder items by descending order of frequency in each FP-tree, which often makes the FP-trees more compact and thus prunes the search space. We will elaborate on each of these pruning strategies in the following paragraphs.

3.3.1 Conditional pattern base pruning

Conditional Pattern base Pruning (CPP), as presented in Fig. 5, is applied during construction of the conditional pattern base. It prunes conditional transactions, and at the same time tries to find a frequent itemset longer than the longest frequent itemset found so far (referred to as lfi below). Straight-

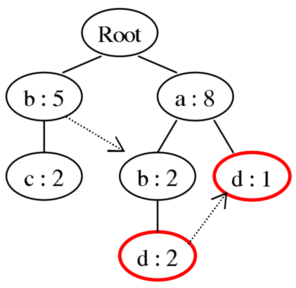


Fig. 4. An example of the conditional pattern base pruning.

```
Input: HT: a header table
Global:
lfi: the longest frequent itemset found so far
Head: a list of items
Output: CPB: the conditional pattern base of Head
Method: Call ConstructCondPatternBase (HT).
Procedure ConstructCondPatternBase (HT) {
(1)  FOR EACH conditional transaction t in HT (visiting via node links){
(2)       IF Length($Head \cup t$) > Length(lfi)
(3)       THEN IF t is frequent
(4)            THEN update lfi with $Head \cup t$;
(5)            ELSE insert t into CPB. }    // end of for each
}    // end of procedure
```

Fig. 5. Construction of conditional pattern base.

forwardly, any conditional transaction t belonging to the conditional pattern base should satisfy $Head \cup t$ is longer than lfi; otherwise it should be pruned because it cannot contribute to forming a longer itemset. This strategy has been discussed in [19]. For instance, in Fig. 4, suppose that the length of lfi is 3 and $Head$ is $\{e, d\}$. We are currently looking for a frequent itemset longer than 3. With FPMAX_LO, two conditional transactions $\{(a : 2), (b : 2)\}$ and $\{(a : 1)\}$ constitute the conditional pattern base of $Head$. However, the conditional transaction $\{(a : 1)\}$ cannot contribute to forming a frequent itemset longer than 3 because the length of $\{e, d\} \cup \{a\}$ is only 3. CPP cuts such not-long-enough conditional transactions as $\{(a : 1)\}$. If a conditional transaction t has a sufficient length and, moreover, t is frequent, a longer frequent itemset $Head \cup t$ is discovered. Let us examine Fig. 4 again and suppose that the minimum support is 2. The conditional transaction $\{(a : 2), (b : 2)\}$ is frequent, so we obtain a longer frequent itemset $\{e, d, a, b\}$ immediately. Due to its larger length than previous lfi, $\{e, d, a, b\}$ is expected to be able to prune more not-long-enough conditional transactions.

3.3.2 Frequent item pruning

Frequent Item Pruning (FIP), as described in Fig. 6, is applied during the "finding all frequent items in the conditional pattern base" phase. It is based on the observation that any conditional transaction that contains insufficient frequent items cannot contribute to generating a longer frequent itemset and should be trimmed. It imposes a stricter condition on the screening of conditional transactions than CPP. An example is given in Fig. 7(a) with three conditional transactions in the conditional pattern base. Suppose that $Head$ is $\{g, f\}$, the minimum support is 2, and the length of the longest frequent itemset found so far is 4. In the conditional pattern base, e is not a frequent item, so conditional transaction 300 could be considered as $\{(b : 1), (d : 1)\}$, which is non-contributing because $\{g, f\} \cup \{b, d\}$ is no longer than 4. Thus, transaction 300 is eliminated from the conditional pattern base, as shown in Fig. 7(b). Since some transactions have been removed from the conditional pattern base, some previously frequent items may become infrequent, and thus some other transactions may become non-contributing. We recursively call the procedure to trim more transactions. Let us continue with this example. In Fig. 7(b), b and d become infrequent this time. As above, transaction 100 and 200 are removed. Then the conditional pattern base is empty, i.e., we can stop exploring with $Head = \{g, f\}$.

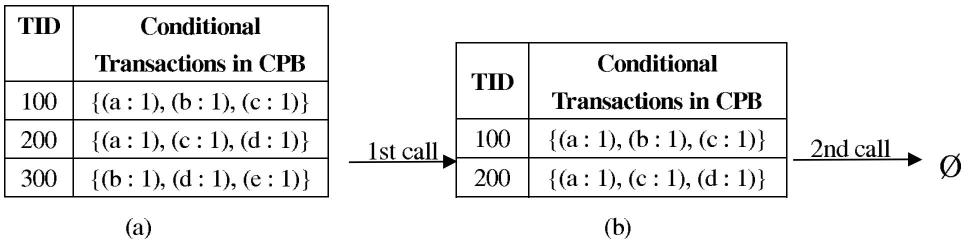
3.3.3 Dynamic reordering

As stated in [11], sorting the items in the header table by descending order of frequency will often increase the compression rate for an FP-tree compared with its corresponding database. The items in transactions will be inserted into FP-tree in their order in the header table, and a sorted header table will keep the nodes of more frequent items closer to the root, which usually enables more sharing of paths and produces higher compression. However, FP-growth and FPMAX only reorder the items in the header table of the initial FP-tree and follow this order to construct header tables of conditional FP-trees. We call this ordering method "static ordering".

In our algorithm, we apply the reordering process to the header tables of all FP-trees. We expect that the dynamic reordering process can make the FP-trees more compact. We dynamically sort items in the header table in descending order of frequency before generating each FP-tree. This "Dynamic Reordering" (DR) is also addressed in [7]. Generally, it will improve the performance in both space (less memory with smaller FP-trees) and time (fewer recursions). Let us study the following example. We refer to the example database and its corresponding initial FP-tree in Fig. 2. For item e, there are four conditional transactions, $\{(a : 1), (c : 1), (b : 1)\}$, $\{(c : 1), (b : 1)\}$, $\{(a : 1), (c : 1)\}$ and $\{(c : 1)\}$. FP-growth generates the header table as

```
Input: CPB: the conditional pattern base of Head
Global:
lfi: the longest frequent itemset found so far
Head: a list of items
Local:
Tail: the set of all frequent items in CPB
Deleted: the number of deleted transactions
Output: modified CPB and Tail
Method: Call GetFrequentItemsinCPB (CPB).
Procedure GetFrequentItemsinCPB (CPB) {
(1)  Find all frequent items (Tail) in CPB;
(2)  FOR EACH conditional transaction t in CPB {
(3)          IF Length(Head ∪ {frequent items in t}) ≤ Length(lfi)
(4)          THEN {
(5)                  Remove t from CPB;
(6)                  Deleted = Deleted + 1; } }   // end of for each
(7)  IF Deleted > 0
(8)  THEN GetFrequentItemsinCPB (CPB);
(9)  ELSE return CPB and Tail.
}    // end of procedure
```

Fig. 6. Getting frequent items in conditional pattern base.



(a)

TID	Conditional Transactions in CPB
100	{(a : 1), (b : 1), (c : 1)}
200	{(a : 1), (c : 1), (d : 1)}
300	{(b : 1), (d : 1), (e : 1)}

$\xrightarrow{\text{1st call}}$

(b)

TID	Conditional Transactions in CPB
100	{(a : 1), (b : 1), (c : 1)}
200	{(a : 1), (c : 1), (d : 1)}

$\xrightarrow{\text{2nd call}} \emptyset$

Fig. 7. An example of the frequent item pruning.

$\{b : 2, c : 4, a : 2\}$ from top to bottom following the item order in the header table of the initial FP-tree. The corresponding conditional FP-tree is shown in Fig. 8(a). In contrast, our algorithm organizes the header table as $\{c : 4, a : 2, b : 2\}$ from top to bottom after dynamic reordering and constructs the conditional FP-tree in Fig. 8(b). There are five nodes in the conditional FP-tree in Fig. 8(a), but only four nodes in Fig. 8(b), which demonstrates the contribution of dynamic reordering in this case.

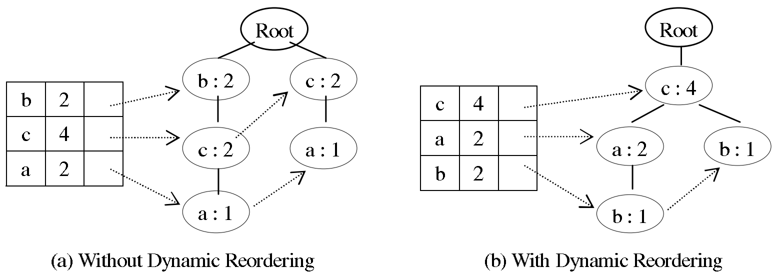


Fig. 8. An example of dynamic reordering.

```
Input: T: an FP-tree
Global:
lfi: the longest frequent itemset found so far
Head: a list of items
Tree: the initial FP-tree
Output: The lfi that is a maximum length frequent itemset
Method: Call LFIMiner (Tree).
Procedure LFIMiner (T) {
(1)  IF T only contains a single path P
(2)  THEN update lfi with Head ∪ P;
(3)  ELSE FOR EACH item i in header table of T DO {
(4)          Append i to Head;
(5)          Construct Head's conditional pattern base using CPP;
(6)          Tail = {frequent items in Head's conditional pattern base} using FIP;
(7)          IF Length(Head ∪ Tail) > Length(lfi)
(8)          THEN {
(9)                  Construct Head's conditional FP-tree T_Head using DR;
(10)                 LFIMiner (T_Head); }
(11)         Remove i from Head. }   // end of for each
}    // end of procedure
```

Fig. 9. The LFIMiner algorithm.

3.4 The LFIMiner algorithm

The LFIMiner algorithm, which incorporates CPP, FIP and DR, is shown in Fig. 9. The differences from FPMAX_LO are highlighted by underlining.

3.5 The LFIMiner_ALL algorithm

To find all the maximum length frequent itemsets in a database, we modified LFIMiner a little. Usually, the LFI set is orders of magnitude smaller than the MFI set. Thus the LFI mining process should be quick. The LFIMiner_ALL algorithm is presented in Fig. 10, with the following notations: CPB-conditional pattern base; $Head$-a list of items; $Tail$-the set of frequent items in CPB;

```
Input: T: an FP-tree
Global: Head: a list of items; Tree: the initial FP-tree
LFIList: the set of longest frequent itemsets found so far
LFILen: the length of longest frequent itemsets found so far
Output: The LFIList that is set of all the maximum length frequent itemsets
Method: Call LFIMiner_ALL (Tree).
Procedure LFIMiner_ALL (T) {
(1)  IF T only contains a single path P
(2)  THEN IF Length(Head ∪ P) > LFILen
(3)       THEN {
(4)            Empty LFIList;
(5)            Insert Head ∪ P into LFIList;
(6)            Update LFILen with Length(Head ∪ P); }
(7)       ELSE insert Head ∪ P into LFIList;
(8)  ELSE FOR EACH item i in header table of T DO {
(9)       Append i to Head;
(10)      Construct Head's conditional pattern base using CPP;
(11)      Tail = {frequent items in base} using FIP;
(12)      IF Length(Head ∪ Tail) ≥ LFILen
(13)      THEN {
(14)           Construct Head's conditional FP-tree T_Head using DR;
(15)           LFIMiner_ALL (T_Head); }
(16)      Remove i from Head. }  // end of for each
}    // end of procedure
```

Fig. 10. The LFIMiner_ALL algorithm.

```
Procedure ConstructCPB_ALL (HeaderTable HT) {
(1)  FOR EACH conditional transaction t in HT {
(2)  IF Length(Head ∪ t) ≧ LFILen
(3)  THEN IF t is frequent
(4)       THEN IF Length(Head ∪ t) > LFILen
(5)            THEN {
(6)                 Empty LFIList;
(7)                 Insert Head ∪ t into LFIList;
(8)                 Update LFILen; }
(9)            ELSE Insert Head ∪ t into LFIList;
(10)      ELSE insert t into CPB. } }
```

```
Deleted: the number of deleted transactions
Procedure GetFrequentItemsinCPB_ALL (CPB) {
(1) Tail = all frequent items in CPB;
(2) FOR EACH conditional transaction t in CPB {
(3)   IF Length(Head ∪ {frequent items in t}) < LFIlen
(4)   THEN {
(5)        Remove t from CPB;
(6)        Deleted = Deleted + 1; } }
(7) IF Deleted > 0
(8) THEN GetFrequentItemsinCPB_ALL (CPB);
(9) ELSE return CPB and Tail. }
```

Fig. 11. Changed CPP (left) and FIP (right) pruning.

$LFIList$-the set of maximum length frequent itemsets; $LFILen$-the length of longest frequent itemsets. The differences from LFIMiner are highlighted by underlining. Lines (2)-(7) insert a newly found longest frequent itemset into the LFI set. Note the difference between line (12) in Fig. 10 and line (7) in Fig. 9. Unlike LFIMiner, in LFIMiner_ALL, we cannot ignore the cases where the combination set of $Head$ with all frequent items $Tail$ in $Head$'s condi-

Table 1
Some characteristics of real-world data sets.

Dataset	File Size(KB)	#Trans	#Items	Avg Trans Len
Mushroom	558	8,124	119	23
Chess	335	3,196	76	37
Connect4	9,039	67,557	129	43
Pumsb*	11,028	49,046	7,117	50

tional pattern base has equal length with the longest frequent itemsets found so far. The CPP and FIP pruning methods were modified accordingly, which are shown in Fig. 11.

For performance comparison, we also constructed modified versions of MAFIA_LO and FPMAX_LO that find all the maximum itemsets. They are named MAFIA_LO_ALL and FPMAX_LO_ALL respectively. Their details are given in Appendix A. The major difference from the original versions is that the current candidate under examination is required to be no shorter than the longest frequent itemset found so far.

4 Experimental evaluation

In this section, we present the experimental results for our algorithms. First, we describe an in-depth study on the performance effect of each optimization component. Then we compare LFIMiner with MAFIA_LO and FPMAX_LO on some real datasets. Finally we present the results concerning the algorithms that find all the maximum length frequent itemsets.

4.1 The experimental setup

All experiments were conducted on a PC with a 2.4 GHz Intel Pentium 4 processor and 512 MB main memory, running Microsoft Windows XP Professional. All codes were compiled using Microsoft Visual C++ 6.0. MAFIA_LO and MAFIA_LO_ALL were created by modifying the original source file of MAFIA provided by its authors. FPMAX_LO and FPMAX_LO_ALL were implemented by ourselves. All timing results in the figures are averages of five runs. We tested the algorithms on the Mushroom, Chess, Connect4 and Pumsb* datasets. Mushroom is a benchmark dataset widely used in transaction clustering. As described in Section 1, LFI can be used for transaction clustering, so we chose Mushroom into the test. We also used Chess, Connect4 and Pumsb*, which contain longer patterns than Mushroom to test the efficiency of the algorithms. These datasets have been widely used in frequent itemset mining. Mushroom, Chess and Connect4 are available from the UCI Machine Learning Repository [15]. Pumsb* is census data from PUMS (Public Use Microdata Sample). In general, at the higher levels of minimum support, the longest pattern length in these datasets varies between 5-14 items, while for some lower levels of minimum support, the longest patterns have over 20 or even 30-40 items. Some characteristics of these real-world data sets are shown in Table 1.

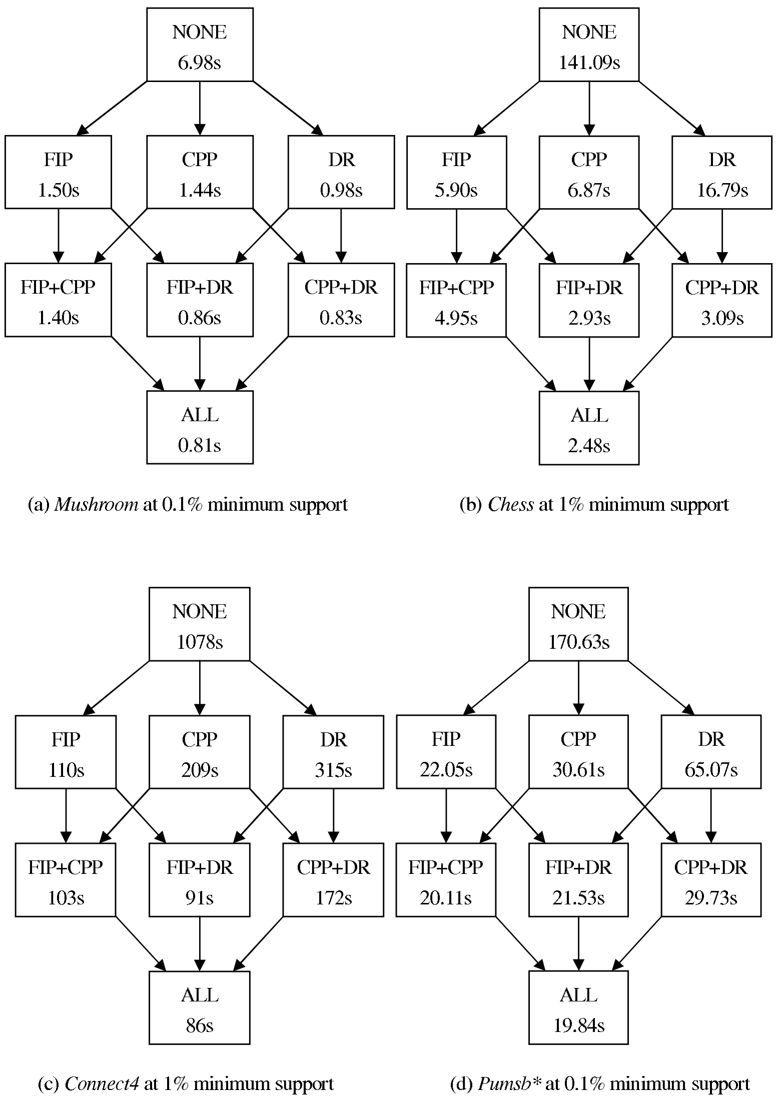
4.2 Algorithmic component analysis

First, we present a complete analysis of component effects on LFIMiner. The three main components in our algorithm are: a) CPP, b) FIP and c) DR. CPP and FIP reduce the size of the FP-tree by pruning some non-contributing conditional transactions. DR reduces the size of the FP-tree by keeping more frequent items closer to the root to enable more sharing of paths.

The results with different components combination on different datasets are presented in Fig. 12. The components of the algorithm are represented in a lattice format, in which the running time is shown. We denote FPMAX_LO by "NONE", and FPMAX_LO with each separate component by "FIP", "CPP" and "DR", respectively. "FIP+CPP" denotes the use of both FIP and CPP. Finally, LFIMiner is denoted by "ALL". The results consistently show that each component improves performance, and the best results are achieved by combining them together. FIP has the biggest impact among the three components, since it is most likely to trim a large number of candidate transactions by its recursive pruning process. In the presence of FIP, the addition of CPP does not make much a difference, either from FIP to FIP+CPP or from FIP+DR to ALL. Since FIP and CPP both trim conditional transactions, it is not surprising that their efficacy overlaps to some extent. On the other hand, DR also achieves significant savings.

4.3 A comparison with MAFIA_LO and FPMAX_LO

Bayardo described in [4] that he implemented a version of Max-Miner that finds the maximum length frequent itemsets. To our knowledge, there are no other existing algorithms to mine maximum itemsets. Many algorithms exist for mining MFI. Agarwal et al. [1] showed that DepthProject runs more than an order of magnitude faster than Max-Miner. Burdick et al. [6] showed that MAFIA outperforms DepthProject by a factor of three to five. Grahne and Zhu [10] showed that FPMAX achieves comparable performance with MAFIA and GenMax. For performance comparison, we modified the efficient MAFIA



(a) *Mushroom* at 0.1% minimum support

(b) *Chess* at 1% minimum support

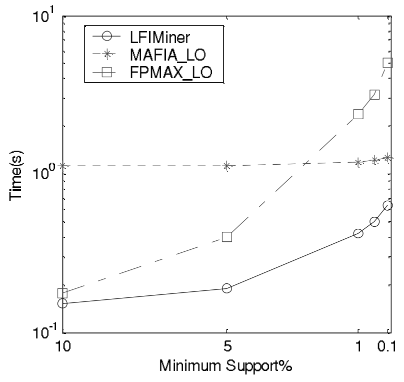
(c) *Connect4* at 1% minimum support

(d) *Pumsb** at 0.1% minimum support

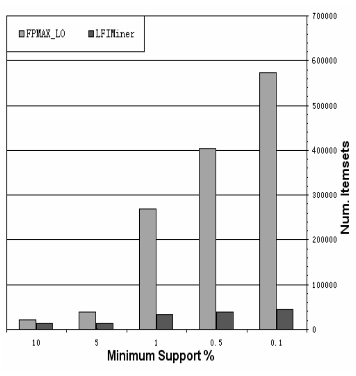
Fig. 12. Impact comparison of the three components.

and FPMAX algorithms a little to mine the maximum length frequent itemset. Why did we select these two algorithms? In fact, DepthProject, MAFIA and GenMax share a lot in common: Search the item subset lattice (or lexicographic tree) in a depth-first way, apply lookahead pruning and dynamic reordering to reduce the search space, use a compression technique for fast support counting. Thus we picked out MAFIA as the representative of the three algorithms. FPMAX is fundamentally different from the above three and resembles our algorithm, because it not only employs the same FP-tree structure but also extends the same FP-growth algorithm. Thus we also chose FPMAX as a competitor.

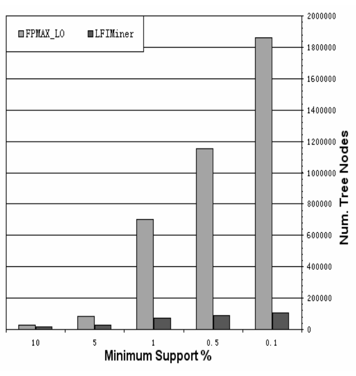
Fig. 13 illustrates the comparison results of these three algorithms for Mushroom, Chess, Connect4 and Pumsb*, respectively. The left column shows the running time of the three algorithms. The x-axis is the user-specified minimum



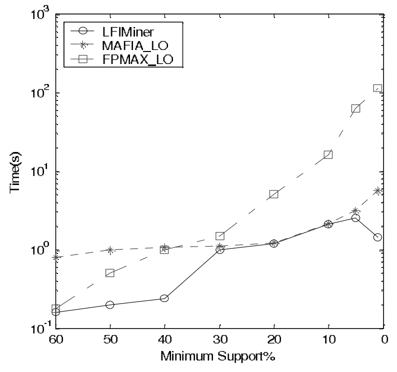
(a) Time for *Mushroom*



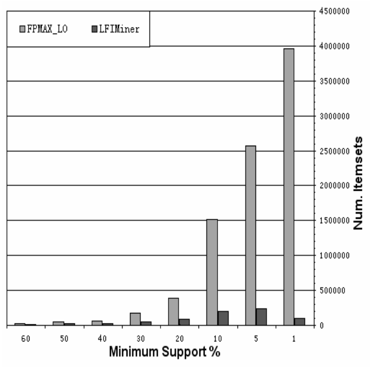
(b) Number of Itemsets



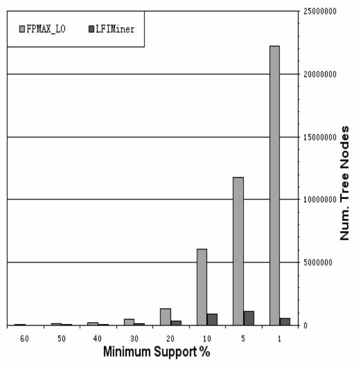
(c) Number of Tree Nodes



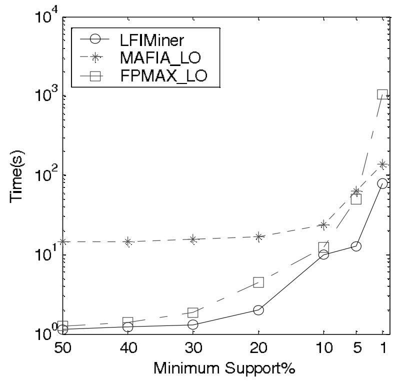
(d) Time for *Chess*



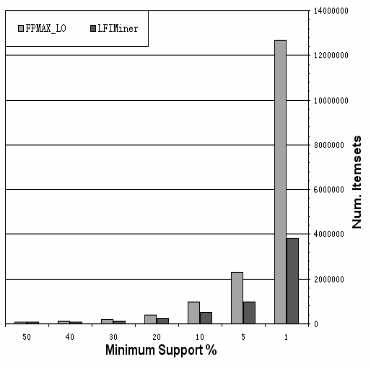
(e) Number of itemsets



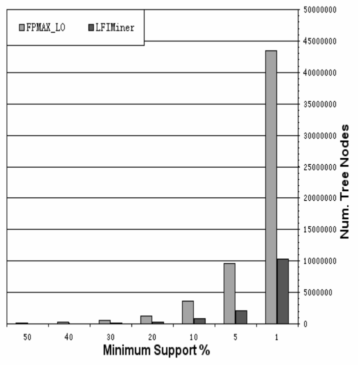
(f) Number of tree nodes



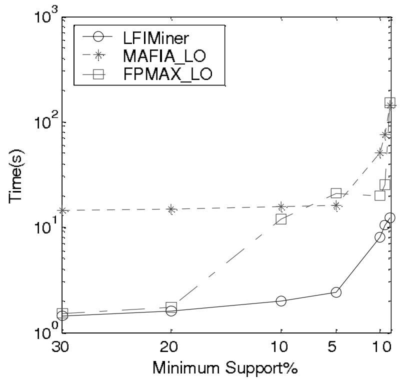
(g) Time for *Connect4*



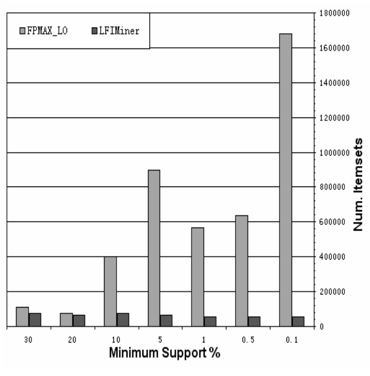
(h) Number of itemsets



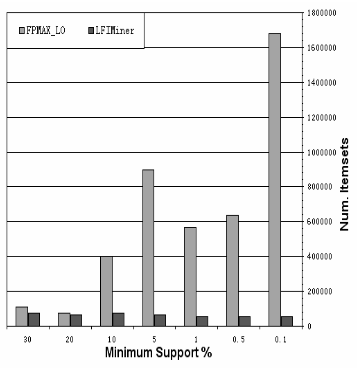
(i) Number of tree nodes



(j) Time for *Pumsb**



(k) Number of itemsets



(l) Number of tree nodes

Fig. 13. Performance comparison for mining one LFI.

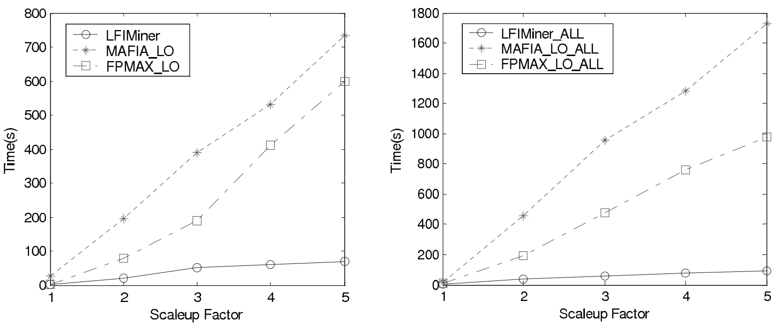


Fig. 14. Scaleup on Connect4 for mining one LFI (left) and all LFI (right).

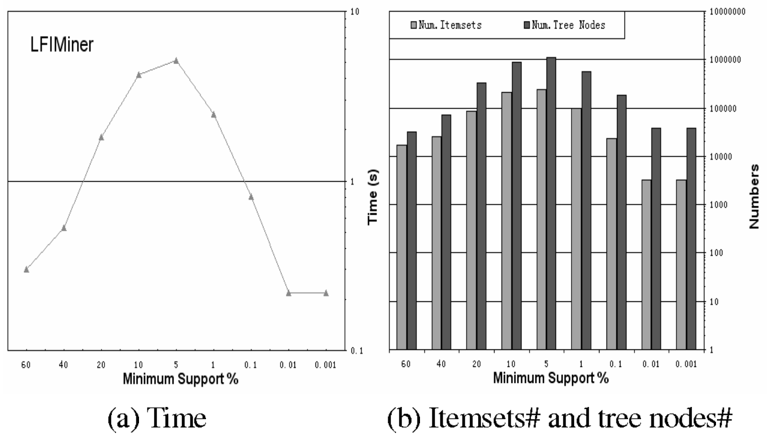


Fig. 15. LFIMiner on Chess.

support, expressed as a percentage, while the y-axis shows the running time in seconds. The middle column compares the number of itemsets processed by FPMAX_LO and LFIMiner. The x-axis is the minimum support, and the y-axis shows the number. The right column compares the number of FP-tree nodes created by FPMAX_LO and LFIMiner. The x-axis is the minimum support, and the y-axis shows the number.

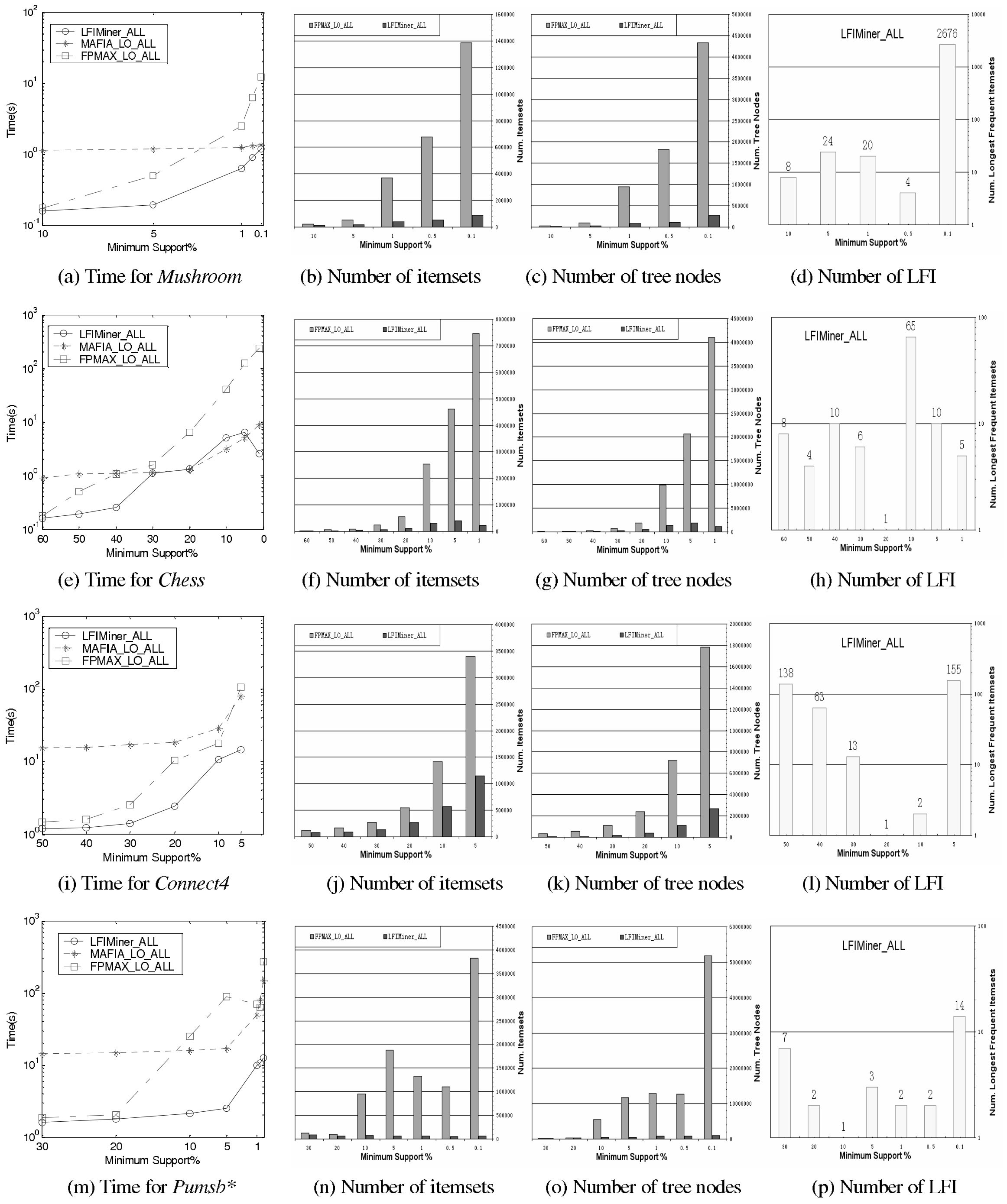
In general, LFIMiner runs consistently faster than MAFIA_LO and FPMAX_LO, especially when the database is large and the transactions in the database are large (Connect4 and Pumsb*). For high levels of support, FPMAX_LO works better than MAFIA_LO, while for low levels of support, MAFIA_LO is more efficient than FPMAX_LO. This can be explained as follows: MAFIA_LO needs a fixed time to convert the database into its vertical format, no matter what the support is. When the support is high, for FPMAX_LO, it will result in a small FP-tree, and thus mining is fast. MAFIA_LO also mines fast, but the time for database conversion cannot be overlooked. This is reflected in the figures that the time taken by MAFIA_LO for high levels of support changes slightly. A majority of the time is used for database conversion. In contrast, when the support is low, without effective pruning, FPMAX_LO spends considerable time in constructing bushy FP-trees. This largely slows down the processing. MAFIA_LO, on the other hand, is not influenced so much, due to its effective pruning and fast support counting with projected bitmap representation of the database.

Because the results on all the datasets are similar, we only explain Fig. 13 for Mushroom. LFIMiner runs faster than MAFIA_LO and FPMAX_LO. MAFIA_LO performs better than FPMAX_LO, as the support varies from 1% downwards. As shown in Fig. 13(b) and (c), when the support decreases, the number of itemsets processed by FPMAX_LO increases dramatically. This consequently leads to the great increase in the number of created tree nodes. On the other hand, for LFIMiner, the number increases slowly due to effective pruning. For example, at support 0.1%, the itemsets processed by LFIMiner are only 8% of those processed by FPMAX_LO.

To test the scalability, we ran the three programs on the Connect4 dataset, which was vertically enlarged by adding transactions into the original dataset. We created new transactions by modeling the distribution of the values of each categorical attribute in the original dataset. In contrast to the vertical scaling used in [6], which scaled the dataset by duplicating the transactions, we created similar but not duplicated transactions. This is a more realistic way of enlarging the dataset than simply duplicating the dataset.

With support fixed at 30%, the results are shown in Fig. 14. All algorithms scale almost linearly with database size. But MAFIA_LO shows a steeper increase than LFIMiner and FPMAX_LO. This is not accidental. As the number of transactions increases, we can expect more similar transactions. For LFIMiner and FPMAX_LO, adding similar transactions to the existing ones will not increase the size of the FP-tree much. For MAFIA_LO, however, it increases the cost for support counting more significantly because the bitmaps become long. In addition, because of effective pruning, LFIMiner increases much more slowly than FPMAX_LO. In conclusion, we can say that LFIMiner scales well with database size.

There is something interesting here which deserves our attention. Fig. 15(a) shows the running time of LFIMiner on Chess with different levels of support. As the support decreases, the time does not increase monotonically; instead it first increases, then decreases, and finally keeps steady. Fig. 15(b) reflects the variation of number of itemsets processed and tree nodes created by LFIMiner. We can see that it is consistent with the time variation in Fig. 15(a). Apparently, as the support decreases, the number of candidate itemsets increases, but at the same time the frequent itemset discovered grows longer, which allows more candidate itemsets to be trimmed. In the first phase, the speed of candidate itemset generation exceeds that of candidate itemset reduction, so the running time increases. In the second phase, the speed of candidate itemset reduction exceeds that of candidate itemset generation, so the running time decreases. In the final phase, the absolute support actually reduces to 1,



(a) Time for *Mushroom* (b) Number of itemsets (c) Number of tree nodes (d) Number of LFI

(e) Time for *Chess* (f) Number of itemsets (g) Number of tree nodes (h) Number of LFI

(i) Time for *Connect4* (j) Number of itemsets (k) Number of tree nodes (l) Number of LFI

(m) Time for *Pumsb** (n) Number of itemsets (o) Number of tree nodes (p) Number of LFI

Fig. 16. Performance comparison for mining all LFI.

i.e., every transaction is a frequent itemset. In this extreme case, no candidate itemset is generated and thus the running time keeps steady. Similar results were found on Mushroom, Connect4 and Pumsb* as well.

Table 2
Maximum length frequent itemsets for Mushroom at support 10%.

$S = \{$ class=edible, bruises=true, odor=none, gill-spacing=close, gill-size=broad, stalk-shape=tapering, stalk-root=bulbous, stalk-surface-above-ring=smooth, stalk-surface-below-ring=smooth, veil-type=partial, veil-color=white, ring-number=one, ring-type=pendant, spore-print-color=black, habitat=woods $\}$
$P_1 = S \cup \{$spore-print-color=black$\}, P_2 = S \cup \{$population=several$\}$, $P_3 = S \cup \{$spore-print-color=brown$\}, P_4 = S \cup \{$cap-surface=scaly $\}$, $P_5 = S \cup \{$cap-shape=flat $\}, P_6 = S \cup \{$cap-surface=fibrous $\}$, $P_7 = S \cup \{$population=solitary $\}, P_8 = S \cup \{$cap-shape=convex $\}$

4.4 Finding all maximum length frequent itemsets

Here we compare the results of LFIMiner_ALL with MAFIA_LO_ALL and FP-MAX_LO_ALL for finding all maximum length frequent itemsets in Fig. 16. The left column shows the running time, the second column shows the number of itemsets processed by FPMAX_LO_ALL and LFIMiner_ALL, the third column shows the number of FP-tree nodes created by FPMAX_LO_ALL and LFIMiner_ALL, and the right column shows the number of maximum length frequent itemsets found. Compared with Fig. 13, similar results were found as before, though the time required for mining is longer. From the right column, we can see that in general the number of maximum itemsets is under several hundred, which is orders of magnitude smaller than the number of maximal frequent itemsets. For example, at support 10%, the number of MFI is 547 for Mushroom, 2,339,525 for Chess, 130,986 for Connect4, and 16,437 for Pumsb*, while the number of LFI is 8, 65, 2 and 1 respectively. Fig. 14 demonstrates the scalability of the three algorithms. This result is similar to the results presented in previous sections.

For illustration, sample sets of LFI are shown in Tables 2 and 3 for datasets Mushroom and Connect4, respectively. For Mushroom, at support 10%, eight LFI of 16 items are mined, which share a common subset of 15 items. They all describe the connection of key mushroom attributes to edibility. For Connect4, at support 20%, there is exactly one LFI of 27 items. Being all blank "*b*", they specify a common board position in the game of connect-4.

Table 3
Maximum length frequent itemset for Connect4 at support 20%.

6	*b*	*b*	*b*	*b*	*b*	*b*	*b*
5	*b*	*b*	*b*	*b*	*b*	*b*	*b*
4	*b*	*b*	*b*	*b*	*b*	*b*	*b*
3	*b*			*b*	*b*	*b*	*b*
2					*b*		
1							
	a	b	c	d	e	f	g

5 Conclusions and future work

In this paper, we introduced the problem of finding maximum length frequent itemset and identified some real world applications for the proposed problem. We proposed an efficient algorithm, LFIMiner, which is built on top of the pattern fragment growth method and makes use of the FP-tree structure. The FP-tree structure stores compressed information about frequent itemsets in a database and the pattern growth method avoids the costly candidate set generation and test. In addition, a number of optimization techniques, such as CPP, FIP and DR, have been exploited in our algorithm for effectively pruning the search space. CPP and FIP pruning schemes help LFIMiner reduce the search space dramatically by removing non-contributing conditional itemsets and in turn narrowing the conditional FP-trees. DR reduces the size of an FP-tree by keeping more frequent items closer to the root in the FP-tree.

In comparison with MAFIA_LO and FPMAX_LO, our experimental results showed that LFIMiner is a faster method for mining maximum length frequent itemset. A detailed component analysis highlighted FIP's effectiveness in reducing the search space due to its recursive process. LFIMiner also exhibits a good scalability. Besides, a variant of this algorithm, LFIMiner_ALL, was also developed for efficiently mining all maximum length frequent itemsets.

As for future work, we plan to apply the proposed technique to a variety of large data sets where the maximum length itemsets are of interest. The benchmark datasets used in our experiments may not be representative of the particular type of data sets where users want to find the maximum length frequent itemsets. Also, other requirements may be added in the mining process for the longest patterns, such as those in correlation [21], data stream [22] and temporal pattern [13] mining. Finally, since LFI has a potential to be an interesting pattern to preserve during clustering, another direction is to exploit LFI for transaction clustering.

References

[1] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 108–118, 2000.

[2] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 207–216, 1993.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Vary Large Data Bases*, pages 487–499, 1994.

[4] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 85–93, 1998.

[5] F. Beil, M. Ester, and X. Xu. Frequent term-based text clustering. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 436–442, 2002.

[6] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transaction databases. In *Proceedings of the 17th International Conference on Data Engineering*, pages 443–452, 2001.

[7] W. Cheung and O. R. Zaiane. Incremental mining of frequent patterns without candidate generation or support constraint. In *Proceedings of the 7th International Database Engineering and Applications Symposium*, pages 111–116, 2003.

[8] B. C. M. Fung, K. Wang, and M. Ester. Large hierarchical document clustering using frequent itemsets. In *Proceedings of the 3rd SIAM International Conference on Data Mining*, 2003.

[9] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *Proceedings of the 1st IEEE International Conference on Data Mining*, pages 163–170, 2001.

[10] G. Grahne and J. Zhu. High performance mining of maximal frequent itemsets. In *Proceeding of the 6th SIAM International Workshop on High Performance Data Mining*, pages 135–143, 2003.

[11] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2000.

[12] J. Hipp, U. Guntzer, and G. Nakaeizadeh. Algorithms for association rule mining - a general survey and comparison. *SIGKDD Explorations*, 2(1):58–64, 2000.

[13] Y. Li, S. Zhu, X. S. Wang, and S. Jajodia. Looking into the seeds of time: Discovering temporal patterns in large transaction sets. *Information Sciences*, 176(8):1003–1031, 2006.

[14] D. Lin and Z. M. Kedem. Pincer-search: A new algorithm for discovering the maximum frequent itemset. In *Proceedings of the 6th International Conference on Extending Database Technology*, pages 105–119, 1998.

[15] D. J. Newman, S. Hettich, C. L. Blake, and C. J. Merz. UCI repository of machine learning databases. http://www.ics.uci.edu/∼mlearn/MLRepository.html, 1998.

[16] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *Proceedings of the 5th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[17] P. Shenoy, J. R. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 22–33, 2000.

[18] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the 5th International Conference on Extending Database Technology*, pages 3–17, 1996.

[19] M. Wojciechowski and M. Zakrzewicz. Dataset filtering techniques in constraint-based frequent pattern mining. In *Proceedings of ESF Exploratory Workshop on Pattern Detection and Discovery*, pages 77–91, 2002.

[20] H. Xiong, P.-N. Tan, and V. Kumar. Mining strong affinity association patterns in data sets with skewed support distribution. In *Proceedings of the 3rd IEEE International Conference on Data Mining*, pages 387–394, 2003.

[21] H. Xiong, P.-N. Tan, and V. Kumar. Hyperclique pattern discovery. *Data Mining and Knowledge Discovery Journal*, 13(2):219–242, 2006.

[22] J. X. Yu, Z. Chong, H. Lu, Z. Zhang, and A. Zhou. A false negative approach to mining frequent itemsets from high speed transactional data streams. *Information Sciences*, 176(14):1986–2015, 2006.

[23] M. J. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 42(1/2):31–60, 2001.

[24] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. Technical Report 01-1, Compuer Science Dept., Rensselaer Polytechnic Institute, 2001.

[25] M. J. Zaki and C.-J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *Proceedings of the 2nd SIAM International Conference on Data Mining*, pages 12–28, 2002.

```
Input: C: the current node
IsHUT: whether this is a HUT check
Global: lfi: the longest frequent itemset found so far
Local:
HUT: the set of head & tail
PEPSet: a set of items that are moved from tail to head by PEP pruning
NewNode: a node
Output: The LFI that is a maximum length frequent itemset
Method: Call MAFIA_LO (Root, True).
Procedure MAFIA_LO (C, IsHUT) {
(1)  HUT = C.head ∪ C.tail;
(2)  IF Length(HUT) ≦ Length(lfi)
(3)  THEN stop generation of children and return;
(4)  Count all children, use PEP to trim the tail, and reorder by increasing support;
(5)  PEPSet = {items moved from tail to head};
(6)  C.head = C.head ∪ PEPSet;
(7)  FOR EACH item i in C.trimmed_tail DO {
(8)          IsHUT = whether i is the first item in the tail;
(9)          NewNode = C ∪ {i};
(10)         MAFIA_LO (NewNode, IsHUT) };       // end of for each
(11) IF IsHUT and all extensions are frequent
(12) THEN stop exploring this subtree and go back up tree to when IsHUT was changed to True;
(13) IF C is a leaf and Length(C.head) > Length(lfi)
(14) THEN lfi = C.head
}    // end of procedure
```

Fig. A.1. The MAFIA_LO algorithm.

A The modified MAFIA and FPMAX algorithms

The modified MAFIA algorithms are given in Figs. A.1 and A.2 respectively for mining one/all maximum length frequent itemsets. Fig. A.3 illustrates the FPMAX_LO_ALL algorithm for mining all maximum length frequent itemsets. In the MAFIA_LO algorithm, differences from the original MAFIA algorithm are highlighted by underlining. Line (2) modifies the original HUTMFI pruning (original statement is "IF HUT is in MFI"). If a node C's HUT (head union tail) is discovered to be no longer than the longest frequent itemset found so far, we never need to explore any subset of the HUT, and thus we can prune the entire subtree rooted at node C. Lines (13) and (14) find a longer frequent itemset and perform the updating, while the original statements [6] are "IF (C is a leaf and C.head is not in MFI) THEN Add C.head to MFI" .

```
Input:
C: the current node
IsHUT: whether this is a HUT check
Global:
LFIList: the set of longest frequent itemsets found so far
LFILen: the length of longest frequent itemsets found so far
Local:
HUT: the set of head & tail
PEPSet: a set of items that are moved from tail to head by PEP pruning
NewNode: a node
Output: The LFIList that is set of all maximum length frequent itemsets
Method: Call MAFIA_LO_ALL (Root, True).
Procedure MAFIA_LO_ALL (C, IsHUT) {
(1)  HUT = C.head ∪ C.tail;
(2)  IF Length(HUT) < LFILen
(3)  THEN stop generation of children and return;
(4)  Count all children, use PEP to trim the tail, and reorder by increasing support;
(5)  PEPSet = {items moved from tail to head};
(6)  C.head = C.head ∪ PEPSet;
(7)  FOR EACH item i in C.trimmed_tail DO {
(8)        IsHUT = whether i is the first item in the tail;
(9)        NewNode = C ∪ {i};
(10)       MAFIA_LO_ALL (NewNode, IsHUT) };     // end of for each
(11) IF IsHUT and all extensions are frequent
(12) THEN stop exploring this subtree and go back up tree to when IsHUT was changed to True;
(13) IF C is a leaf and Length(C.head) ≧ LFILen
(14) THEN IF Length(C.head) > LFILen
(15)       THEN{
(16)            Empty LFIList;
(17)            Insert C.head into LFIList;
(18)            Update LFILen with Length(C.head); }
(19)       ELSE insert C.head into LFIList;}      // end of procedure
```

Fig. A.2. The MAFIA_LO_ALL algorithm.

```
**Input**: T: an FP-tree
**Global**: Head: a list of items; Tree: the initial FP-tree
LFIList: the set of longest frequent itemsets found so far
LFILen: the length of longest frequent itemsets found so far
**Output**: The LFIList that is set of all maximum length frequent itemsets
**Method**: Call FPMAX_LO_ALL (Tree).
Procedure FPMAX_LO_ALL (T) {
(1)   IF T only contains a single path P
(2)   THEN IF Length($Head \cup P$) > LFILen
(3)         THEN {
(4)               Empty LFIList;
(5)               Insert $Head \cup P$ into LFIList;
(6)               Update LFILen with Length($Head \cup P$); }
(7)         ELSE insert $Head \cup P$ into LFIList;
(8)   ELSE FOR EACH item i in header table of T DO {
(9)         Append i to Head;
(10)        Construct Head's conditional pattern base;
(11)        Tail = {frequent items in base};
(12)        IF Length($Head \cup Tail$) $\geqq$ LFILen
(13)        THEN {
(14)              Construct Head's conditional FP-tree $T_{Head}$;
(15)              FPMAX_LO_ALL ($T_{Head}$); }
(16)        Remove i from Head. }   // end of for each
}     // end of procedure
```

Fig. A.3. The FPMAX_LO_ALL algorithm.