Proc. the 2nd IEEE ICDM'04 Workshop on Frequent Itemset Mining Implementations (FIMI'04), 2004.

(Awarded by FIMI'04 BEST IMPLEMENTATION AWARD, Nov. 2004)

LCM ver. 2: Efficient Mining Algorithms for Frequent/Closed/Maximal Itemsets

Takeaki Uno[1], Masashi Kiyomi[1], Hiroki Arimura[2]
[1] National Institute of Informatics 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430, Japan
e-mail: uno@nii.jp, masashi@grad.nii.ac.jp
[2] Information Science and Technology, Hokkaido University
Kita 14-jo Nishi 9-chome, 060-0814 Sapporo, JAPAN, e-mail: arim@ist.hokudai.ac.jp

Abstract: For a transaction database, a frequent itemset is an itemset included in at least a specified number of transactions. A frequent itemset P is maximal if P is included in no other frequent itemset, and closed if P is included in no other itemset included in the exactly same transactions as P. The problems of finding these frequent itemsets are fundamental in data mining, and from the applications, fast implementations for solving the problems are needed. In this paper, we propose efficient algorithms LCM (Linear time Closed itemset Miner), LCMfreq and LCMmax for these problems. We show the efficiency of our algorithms by computational experiments compared with existing algorithms.

1 Introduction

Frequent item set mining is one of the fundamental problems in data mining and has many applications such as association rule mining, inductive databases, and query expansion. From these applications, fast implementations of frequent itemset mining problems are needed. In this paper, we propose the second versions of LCM, LCMfreq and LCMmax, for enumerating closed, all and maximal frequent itemsets. LCM is an abbreviation of *Linear time Closed item set Miner*.

In FIMI03[7], we proposed the first version of LCM, which is for enumerating frequent closed itemsets. LCM uses *prefix preserving closure extension* (ppc extension in short), which is an extension from a closed itemset to another closed itemset. The extension induces a search tree on the set of frequent closed itemsets, thereby we can completely enumerate closed itemsets without duplications. Generating a ppc extension needs no previously obtained closed itemset. Hence, the memory use of LCM does not depend on the number of frequent closed itemsets, even if there are many frequent closed itemsets.

The time complexity of LCM is theoretically bounded by a linear function in the number of frequent closed itemsets, while the existing algorithms are not. We further developed algorithms for the frequency counting, *occurrence deliver* and *hybrid of diffsets*. They reduce the practical computation time efficiently. Moreover, the framework of LCM is simple. Generating ppc extensions needs no sophisticated data structure such as binary trees. LCM is implemented with only arrays. Therefore, LCM is fast, and outperforms than other algorithms for some sparse datasets.

However, LCM does not have any routine for reducing the database, while many existing algorithms have. Thus, the performance of LCM is not good for dense datasets with large minimum supports, which involve many unnecessary items and transactions. At FIMI03, we also proposed modifications of LCM, LCMfreq and LCMmax, for enumerating all frequent itemsets and maximal frequent itemsets. Although they are fast for some instances, if LCM is not fast for an instance, they are also not fast for the instance. Existing maximal frequent itemset mining algorithms have efficient pruning methods to reduce the number of iterations, while LCMmax does not have. It is also a reason of the slowness of LCMmax.

This paper proposes the second version of LCM algorithms. We added database reduction to LCM, so that problems of dense datasets can be solved in short time. The second version of LCMmax includes a pruning method, thus the computation time is reduced when the number of maximal frequent itemsets is small. We further developed new algorithms for checking the maximality of a frequent itemset and

for taking the closure of an itemset. We compare the performance of LCM algorithms and other algorithms submitted to FIMI03 by computational experiments. In many instances, LCM algorithms perform above other algorithms.

The organization of the paper is as follows. Section 2 introduces preliminaries. The main algorithms and practical techniques of LCM algorithms are described in Section 3. Section 4 shows the results of computational experiments, and Section 5 concludes the paper.

2 Preliminaries

Let $\mathcal{I} = \{1, ..., n\}$ be the set of *items*. A *transaction database* on $\mathcal{I}$ is a set $\mathcal{T} = \{t_1, \ldots, t_m\}$ such that each t_i is included in $\mathcal{I}$. Each t_i is called a *transaction*. We denote by $||\mathcal{T}||$ the sum of sizes of all transactions in $\mathcal{T}$, that is, the size of database $\mathcal{T}$. A set $P \subseteq \mathcal{I}$ is called an *itemset*.

For itemset P, a transaction including P is called an *occurrence* of P. The *denotation* of P, denoted by $\mathcal{T}(P)$ is the set of the occurrences of P. $|\mathcal{T}(P)|$ is called the *frequency* of P, and denoted by $frq(P)$. For given constant θ, called a *minimum support*, itemset P is *frequent* if $frq(P) \geq \theta$. If a frequent itemset P is included in no other frequent itemset, P is called *maximal*. For any itemsets P and Q, $\mathcal{T}(P \cup Q) = \mathcal{T}(P) \cap \mathcal{T}(Q)$ holds, and if $P \subseteq Q$ then $\mathcal{T}(Q) \subseteq \mathcal{T}(P)$. An itemset P is called *closed* if no other itemset Q satisfies $\mathcal{T}(P) = \mathcal{T}(Q), P \subseteq Q$.

Given set $\mathcal{S} \subseteq \mathcal{T}$ of transactions, let $\mathcal{I}(\mathcal{S})$ be the set of items common to all transactions in $\mathcal{S}$, i.e., $\mathcal{I}(\mathcal{S}) = \bigcap_{T \in \mathcal{S}} T$. Then, we define the *closure* of itemset P in $\mathcal{T}$, denoted by $clo(P)$, by $\mathcal{I}(\mathcal{T}(P))(= \bigcap_{t \in \mathcal{T}(P)} t)$. For every pair of itemsets P and Q, the following properties hold[13, 14].

(1) If $P \subseteq Q$, then $clo(P) \subseteq clo(Q)$.
(2) If $\mathcal{T}(P) = \mathcal{T}(Q)$, then $clo(P) = clo(Q)$.
(3) $clo(clo(P)) = clo(P)$.
(4) $clo(P)$ is the unique smallest closed itemset including P.
(5) A itemset P is a closed itemset if and only if $clo(P) = P$.

For itemset P and item $i \in P$, let $P(i) = P \cap \{1, \ldots, i\}$ be the subset of P consisting only of elements no greater than i, called the *i-prefix* of P. An itemset Q is a *closure extension* of an itemset P if $Q = clo(P \cup \{i\})$ holds for some $i \notin P$. If Q is a closure extension of P, then $Q \supset P$, and $frq(Q) < frq(P)$. We call the item with the maximum index in P the *tail* of P, and denote by $tail(P)$.

3 Algorithms for Efficient Enumeration

In this section, we explain the techniques used in the second versions of LCM algorithms. We explain them one-by-one with comparing to the techniques used by the other algorithms, in the following subsections. The new techniques used in the second version are:

3.2. new database reduction (reduce the frequency counting cost)
3.6. database reduction for fast checking closedness
3.8. database reduction for fast checking maximality
3.7. new pruning algorithm for backtracking-based maximal frequent itemset mining.

The techniques also used in the first versions are:

3.4. occurrence deliver (compute frequency in linear time)
3.5. ppc extension (generates closed itemsets with neither memory nor duplication)
3.3. hypercube decomposition (fast enumeration by grouping frequent itemsets by equivalence class).

The techniques used in the existing algorithms and the first version have citations to the previous papers.

3.1 Enumerating Frequent Itemsets

Any itemset included in a frequent itemset is itself frequent. Thereby, the property "frequent" is monotone. From this, we can construct any frequent itemset from the empty set by adding items one-by-one without passing through any infrequent itemset. Roughly speaking, the existing algorithms are classified into two groups, and algorithms in both groups use this property.

The first group is so called *apriori* or *level-by-level* algorithms [1, 2]. Let $\mathcal{D}_k$ be the set of frequent itemsets of size k. Apriori algorithms start with $\mathcal{D}_0$, that is $\{\emptyset\}$, and compute $\mathcal{D}_k$ from $\mathcal{D}_{k-1}$ in the increasing order of k from $k = 1$. Any itemset in $\mathcal{D}_k$ is obtained from an itemset of $\mathcal{D}_{k-1}$ by adding an item. Apriori algorithms add every item to each itemset of $\mathcal{D}_{k-1}$, and choose frequent itemsets among them. If $\mathcal{D}_k = \emptyset$ holds for some k, then $\mathcal{D}_{k'} = \emptyset$ holds for any $k' > k$. Thus, apriori algorithms stop at such k. This is the scheme of apriori algorithms.

The other group is so called *backtracking* algorithms[3, 18, 19]. Backtracking algorithm is based on recursive calls. An iteration of a backtracking algorithm inputs a frequent itemset P, and generates itemsets by adding every items to P. Then, for each itemset being frequent among them, the iteration generates recursive calls with respect to it. To avoid duplications, an iteration of backtracking algorithms adds items with indices larger than the tail of P. We describe the framework of backtracking algorithms as follows.

ALGORITHM BackTracking (P:current solution)
1. **Output** P
2. **For each** $e \in \mathcal{I}, e > tail(P)$ **do**
3. **If** $P \cup \{e\}$ is frequent **then**
call BackTracking ($P \cup \{e\}$)

An execution of backtracking algorithms gives a tree structure such that the vertices of the tree are iterations, and edges connect two iterations if one of the iteration calls the other. If an iteration I recursively calls another iteration I', then we say that I is the parent of I', and I' is a child of I. For an iteration, the itemset received from the parent is called the *current solution*.

Apriori algorithms use much memory for storing $\mathcal{D}_k$ in memory, while backtracking algorithms use less memory since they keep only the current solution. Backtracking algorithms need no computation for maintaining previously obtained itemsets, so the computation time of backtracking algorithms is generally short. However, apriori algorithms have advantages for the frequency counting.

LCM algorithms are based on backtracking algorithms, and use an efficient techniques for the frequency counting, which are occurrence deliver and anytime database reduction described below. Hence, LCM algorithms compute the frequency efficiently without keeping previously obtained itemsets in memory.

3.2 Maintaining Databases

In the existing studies, *database reduction* is said to be important to reduce the computation time. It is to reduce the input database as the following rules:

1. remove each item included in less than θ transactions
2. remove each item included in all transactions
3. merge the identical transactions into one.

Database reduction performs well when the minimum support is large, and many existing algorithms use it. LCM algorithms also use database reduction.

In the existing studies, the input databases are often stored and maintained by using FP-tree (frequent pattern tree), which is a version of prefix tree (trie) [9]. By using FP-tree, we can search specified transactions from the datasets efficiently. FP-tree compresses the common prefix, so we can decrease the memory use. In addition, FP-tree can detect the identical transactions, thus we can merge them into one. This merge accelerates the frequency counting. From these reasons, FP-trees are used in many algorithms and implementations.

Although FP-tree has many good advantages, we do not use it in the implementation of LCM, but use simple arrays. The main reason is that LCM does not have to search transactions in the database. The main operation of LCM is tracing the transactions in the the denotation of the current solution. Thus, we do not need to use sophisticated data structures for searching.

The other reason is the computation time for the initialization. If we use a standard binary tree for implementing FP-tree, the initialization of the input database takes $O(||\mathcal{T}|| + |\mathcal{T}| \log |\mathcal{T}|)$ time. for constructing FP-tree in memory. Compared to this, LCM detects the identical transactions and stores the database in memory within linear time of the database size. This is because that LCM uses radix sort for this task, which sorts the transactions in a lexicographic order in linear time. In general, the datasets of data mining problems have many transactions, and each transaction has few items. Thus, $||\mathcal{T}||$ is usually smaller than $|\mathcal{T}| \log |\mathcal{T}|$, and LCM has an advantage. The constant factors of the computation time of binary tree operations are relatively larger than that of array operations. LCM also has an advantage at this point. Again, we recall that LCM never search the transactions, so each operation required by LCM can be done in constant time.

FP-tree has an advantage in reducing the memory use. This memory reduction can also reduce the computation time of the frequency counting. To check the efficiency of the reduction, we checked the reduction ratio by FP-tree for some datasets examined in FIMI03. The result is shown in Table 1. Each cell shows the ratio of the number of items needed to be stored by arrays and FP-tree. Usually, the input database is reduced in each iteration, hence we sum up the numbers over all iterations to compute the ratio. In the results of our experiments,

the ratio was not greater 3 in many instances. If $|\mathcal{I}|$ is small, $|\mathcal{T}|$ is large, and the dataset has a randomness, such as accidents, the ratio was up to 6. Generally, a binary tree uses memory three times as much as an array. Thus, the performance of FP-tree seems to be not quite good rather than an array in both memory use and computation time, for many datasets.

3.3 Hypercube Decomposition

LCM finds a number of frequent itemsets at once for reducing the computation time[18]. Since the itemsets obtained at once compose a hypercube in the itemset lattice, we call the technique *hypercube decomposition*. For a frequent itemset P, let $H(P)$ be the set of items e satisfying $e > tail(P)$ and $\mathcal{T}(P) = \mathcal{T}(P \cup \{e\})$. Then, for any $Q \subseteq H(P)$, $\mathcal{T}(P \cup Q) = \mathcal{T}(P)$ holds, and $P \cup Q$ is frequent. LCMfreq uses this property. For two itemsets P and $P \cup Q$, we say that P' is between P and $P \cup Q$ if $P \subseteq P' \subseteq P \cup Q$. In the iteration with respect to P, we output all P' between P and $P \cup H(P)$. This saves about $2^{|H(P)|}$ times of the frequency counting.

To avoid duplications, we do not generate recursive calls with respect to items included in $H(P)$. Instead of generating these recursive calls, we output frequent itemsets including items of $H(P)$ in recursive calls with respect to items not included in $H(P)$. When the algorithm generates a recursive call with respect to $e \notin H(P)$, we pass $H(P)$ to it. In the recursive call, we output all itemsets between $P \cup \{e\}$ and $P \cup \{e\} \cup H(P) \cup H(P \cup \{e\})$. Since any itemset Q satisfies $\mathcal{T}(P \cup Q \cup H(P)) = \mathcal{T}(P \cup Q)$, the itemsets output in the recursive calls are frequent. We describe hypercube decomposition as follows.

ALGORITHM HypercubeDecomposition
(P:current solution, S:itemset)
$S' := S \cup H(P)$
Output all itemsets including P
and included in $P \cup S'$
For each item $e \in \mathcal{I} \setminus (P \cup S')$, $e > tail(P)$ **do**
If $P \cup \{e\}$ is frequent **then**
call HypercubeDecomposition $(P \cup \{e\}, S')$
End for

3.4 Frequency Counting

Generally, the most heavy part of the frequent itemset mining is the frequency counting, which is to count the number of transactions including a newly generated itemset. To reduce the computation time, existing algorithms uses *down project*. For an itemset P, down project computes its denotation $\mathcal{T}(P)$ by using two subsets P_1 and P_2 of P. If $P = P_1 \cup P_2$, then $\mathcal{T}(P) = \mathcal{T}(P_1) \cap \mathcal{T}(P_2)$. Under the condition that the items of P_1 and P_2 are sorted by their indices, the intersection can be computed in $O(|\mathcal{T}(P_1)| + |\mathcal{T}(P_2)|)$ time. Down project uses this property, and computes the denotations quickly. Moreover, if $|\mathcal{T}(P_1)| < \theta$ or $|\mathcal{T}(P_2)| < \theta$ holds, we can see that P never be frequent. It also helps to reduce the computation time.

Apriori-type algorithms accelerates the frequency counting by finding a good pair P_1 and P_2 of subsets of P, such that $|\mathcal{T}(P_1)| + |\mathcal{T}(P_2)|$ is small, or either P_1 or P_2 is infrequent. Backtracking algorithm adds an item e to the current solution P in each iteration, and compute its denotation. By using $\mathcal{T}(\{e\})$, the computation time for the frequency counting is reduced to $O(\sum_{e>tail(P)}(|\mathcal{T}(P)| + |\mathcal{T}(\{e\})|))$.

The bitmap method[5] is a technique for speeding up the computation of taking the intersection in down project. It uses a bitmap image (the characteristic vector) of the denotations. To take the intersection, we have to take $O(|\mathcal{T}|)$ time with bitmap. However, a 32bit CPU can take the intersection of 32bits at once, thus roughly speaking the computation time is reduced to 1/32. This method has a disadvantage for sparse datasets, and is not orthogonal to anytime database reduction described in the below. From the results of the experiments in FIMI 03, bitmap method seems to be not good for sparse large datasets.

LCM algorithms use another method for the frequency counting, called *occurrence deliver*[18, 19]. Occurrence deliver computes the denotations of $P \cup \{e\}$ for $e = tail(P) + 1, ..., |\mathcal{I}|$ at once by tracing transactions in $\mathcal{T}(P)$. It use a bucket for each e to be added, and set them to empty set at the beginning. Then, for each transaction $t \in \mathcal{T}(P)$, occurrence deliver inserts t to the bucket of e for each $e \in t, e > tail(P)$. After these insertions, the bucket of e is equal to $\mathcal{T}(P \cup \{e\})$. For each transaction t, occurrence deliver takes $O(|t \cap \{tail(P)+1, ..., |\mathcal{I}|\}|)$ time. Thus, the computation time is $O(\sum_{T \in \mathcal{T}(P)} |T \cap \{tail(P)+1, ..., |\mathcal{I}|\}|) = O(|\mathcal{T}(P)| + \sum_{e>tail(P)} |\mathcal{T}(P \cup \{e\})|)$. This time complexity is smaller than down project. We describe the pseudo code of occurrence deliver in the following.

ALGORITHM OccurrenceDeliver
(T:database, P:itemset)
1. Set $Bucket[e] := \emptyset$ for each item $e > tail(P)$

dataset and minimum support	chess 40%	accidents 30%	BMS-WebView2 0.05%	T40I10D100K 0.1%
reduction factor by FP-tree	2.27	6.01	1.9	1.57
reduction factor by Hypercube decomposition	6.25	1	1.21	1
reduction factor by apriori (best)	1.11	1.34	1.35	2.85

Table 1: Efficiency test of FP-tree, hypercube decomposition, and apriori: the reduction factor of FP-tree is (sum of # of elements in reduced database by LCM) / (sum of # of elements in reduced database by FP-tree), over all iterations, the reduction ratio of hypercube decomposition is the average number of output frequent itemsets in an iteration, and the reduction ratio of apriori is (sum of $\sum_{e>tail(P)} |\mathcal{T}(P \cup \{e\})|$) / (sum of $\sum_{e>F(P)} |\mathcal{T}(P \cup \{e\})|$), over all iterations.

2. **For each** transaction $t \in \mathcal{T}(P)$ **do**
3. **For each** item $e \in t$, $e > tail(P)$ **do**
4. Insert t to $Bucket[e]$
5. **End for**
6. **End for**
7. **Output** $Bucket[e]$ for all $e > tail(P)$

Let $F(P)$ be the set of items e such that $e > tail(P)$ and $P \cup \{e\}$ is frequent. Apriori algorithms have possibility to find out in short time that $P \cup \{e\}$ is infrequent, thus, in the best case, their computation time can be reduced to $O(\sum_{e \in F(P)} |\mathcal{T}(P \cup \{e\})|)$. If $\sum_{e>tail(P), e \notin F(P)} |\mathcal{T}(P \cup \{e\})|$ is large, occurrence deliver will be slow.

To decrease $\sum_{e>tail(P), e \notin F(P)} |\mathcal{T}(P \cup \{e\})|$, LCM algorithms sort indices of items e in the increasing order of $|\mathcal{T}(\{e\})|$. As we can see in Table 1, this sort reduces $\sum_{e>tail(P), e \notin F(P)} |\mathcal{T}(P \cup \{e\})|$ to 1/4 of $\sum_{e>tail(P)} |\mathcal{T}(P \cup \{e\})|$ in many cases. Since apriori algorithms take much time to maintain previously obtained itemsets, the possibility of speeding up by apriori algorithms is not so large.

LCM algorithms further speeds up the frequency counting by iteratively reducing the database. Suppose that an iteration I of a backtracking algorithm receives a frequent itemset P from its parent. Then, in any descendant iteration of I, no item of indices smaller than $tail(P)$ is added. Hence, any such item can be removed from the database while the execution of the descendant iterations. Similarly, the transactions not including P never include the current solution of any descendant iteration, thus such transactions can be removed while the execution of the descendant iterations. Indeed, infrequent items can be removed, and the identical transactions can be merged.

According to this, LCM algorithms recursively reduce the database while the execution of recursive calls. Before the recursive call, LCM algorithms generate a reduced database according to the above discussion, and pass it to the recursive call. We call this technique *anytime database reduction*.

Anytime database reduction reduces the computation time of the iterations located at the lower levels of the recursion tree. In the recursion tree, many iterations are on the lower levels and few iterations are on the upper levels. Thus, anytime database reduction is expected to be efficient. In our experiments, anytime database reduction works quite well. The following table shows the efficiency of anytime database reduction. We sum up over all iterations the sizes of the database received from the parent, in both cases with anytime database reduction and without anytime database reduction. Each cell shows the sum. The reduction ratio is large especially if the dataset is dense and the minimum support is large.

3.5 Prefix Preserving Closure Extension

Many existing algorithms for mining closed itemsets are based on frequent itemset mining. That is, the algorithms enumerate frequent itemsets, and output those being closed. This approach is efficient when the number of frequent itemsets and the number of frequent closed itemsets differ not so much. However, if the difference between them is large, the algorithms generate many non-closed frequent itemsets,

dataset and minimum support	connect 50%	pumsb 60%	BMS-WebView2 0.1%	T40I10D100K 0.03%
Database reduction	188319235	2125460007	2280260	1704927639
Anytime database reduction	538931	7777187	521576	77371534
Reduction factor	349.4	273.2	4.3	22.0

Table 2: Accumulated number of transactions in database in all iterations

thus they will be not efficient. Many pruning methods have been developed for speeding up, however they are not complete. Thus, the computation time is not bounded by a linear function in the number of frequent closed itemsets. There is a possibility of over linear increase of computation time in the number of output.

LCM uses *prefix preserving closure extension* (ppc-extension in short) for generating closed itemsets[18, 19]. For a closed itemset P, we define the closure tail $clo\_tail(P)$ by the item i of the minimum index satisfying $clo(P(i)) = P$. $clo\_tail(P)$ is always included in P. We say that P' is a ppc extension of P if $P' = clo(P \cup \{e\})$ and $P'(e-1) = P(e-1)$ hold for an item $e > clo\_tail(P)$. Let P_0 be the itemset satisfying $\mathcal{T}(P') = \mathcal{T}$. Any closed itemset $P' \neq P_0$ is a ppc extension of another closed itemset P, and such P is unique for P'. Moreover, the frequency of P is strictly larger than P', hence ppc extension induces a rooted tree on frequent closed itemsets. LCM starts from P_0, and finds all frequent closed itemsets in a depth first manner by recursively generating ppc extensions. The proof of ppc extension algorithms are described in [18, 19].

By ppc extension, the time complexity is bounded by a linear function in the number of frequent closed itemsets. Hence, the computation time of LCM never be super linear in the number of frequent closed itemsets.

3.6 Closure Operation

To enumerate closed itemsets, we have to check whether the current solution P is a closed itemset or not. In the existing studies, there are two methods for this task. The first method is to store in memory previously obtained itemsets which are currently maximal among itemsets having the identical denotation. In this method, we find frequent itemsets one-by-one, and store them in memory with removing itemsets included in another itemset having the identical denotation. After finding all frequent itemsets, only closed itemsets remain in memory. We call this *storage method*. The second method is to generate the closure of P. By adding to P all items e such that $frq(P) = frq(P \cup \{e\})$, we can construct the closure of P. We call the second *closure operation*.

LCM uses closure operations for generating ppc extensions. Similar to the frequency counting, we use database reduction for closure operation. Suppose that the current solution is P, the reduced database is composed of transactions $S_1, ..., S_h$, and each S_l is obtained from transactions $T_1^l, ..., T_k^l$ of the original database. For each S_l, we define the *interior intersection* $In(S_l)$ by $\bigcap_{T \in \{T_1^l, ..., T_k^l\}} T$. Here the closure of P is equal to $\bigcap_{S \in \{S_1, ..., S_h\}} In(S)$. Thus, by using interior intersections, we can efficiently construct the closure of P.

When we merge transactions to reduce the database, interior intersections can be updated efficiently, by taking the intersection of their interior intersections. In the same way as the frequency counting, we can remove infrequent items from the interior intersections for more reduction. The computation time for the closure operation in LCM depends on the size of database, but not on the number of previously obtained itemsets. Thus, storage method has advantages if the number of frequent closed itemsets is small. However, for the instances with a lot of frequent closed itemsets, which take long time to be solved, LCM has an advantage.

3.7 Enumerating Maximal Frequent Itemsets

Many existing algorithms for maximal frequent itemset enumeration are based on the enumeration of frequent itemsets. In breadth-first manner or depth-first manner, they enumerate frequent itemsets and output maximal itemsets among them. To reduce the computation time, the algorithms prune the unnecessary itemsets and recursive calls.

Similar to these algorithms, LCMmax enumerates closed itemsets by backtracking, and outputs maximal itemsets among them. It uses a pruning to cut off unnecessary branches of the recursion. The pruning is based on a re-ordering of the indices of items, in each iteration. We explain the re-ordering in the

following.

Let us consider a backtracking algorithm for enumerating frequent itemsets. Let P be the current solution of an iteration of the algorithm. Suppose that P' is a maximal frequent itemset including P. LCMmax puts new indices to items with indices larger than $tail(P)$ so that any item in P' has an index larger than any item not in P'. Note that this re-ordering of indices has no effect to the correctness of the algorithm.

Let $e > tail(P)$ be an item in P', and consider the recursive call with respect to $P \cup \{e\}$. Any frequent itemset $\hat{P}$ found in the recursive call is included in P', since every item having an index larger than e is included in P', and the recursive call adds to P items only of indices larger than e. From this, we can see that by the re-ordering of indices, recursive calls with respect to items in $P' \cap H$ generates no maximal frequent itemset other than P'.

According to this, an iteration of LCMmax chooses an item $e^* \in H$, and generates a recursive call with respect to $P \cup \{e^*\}$ to obtain a maximal frequent itemset P'. Then, re-orders the indices of items other than e^* as the above, and generates recursive calls with respect to each $e > tail(P)$ not included in $P' \cup \{e^*\}$. In this way, we save the computation time for finding P', and by finding a large itemset, increase the efficiency of this approach. In the following, we describe LCMmax.

ALGORITHM LCMmax (P:itemset, H:items to be added)
1. $H' :=$ the set of items e in H s.t. $P \cup \{e\}$ is frequent
2. **If** $H' = \emptyset$ **then**
3. **If** $P \cup \{e\}$ is infrequent for any e **then** **output** P ; **return**
4. **End if**
5. **End if**
6. Choose an item $e^* \in H'$; $H' := H' \setminus \{e^*\}$
7. LCMmax ($P \cup \{e\}$, H')
8. $P' :=$ frequent itemset of the maximum size found in the recursive call in 7
9. **For each** item $e \in H \setminus P'$ **do**
10. $H' := H' \setminus \{e\}$
11. LCMmax ($P \cup \{e\}$, H')
12. **End for**

3.8 Checking Maximality

When LCMmax finds a frequent itemset P, it checks the current solution is maximal or not. We call this operation *maximality check*. Maximality check is a heavy task, thus many existing algorithms avoid it. They store in memory maximal itemsets among previously obtained frequent itemsets, and update them when they find a new itemset. When the algorithms terminate and obtain all frequent itemsets, only maximal frequent itemsets remain in memory. We call this *storage method*. If the number of maximal frequent itemsets is small, storage method is efficient. However, if the number is large, storage method needs much memory. When a frequent itemset is newly found, storage method checks whether the itemset is included in some itemsets in the memory or not. If the number of frequent itemsets is large, the operation takes long time.

To avoid the disadvantage of storage method, LCMmax operates maximality check. LCMmax checks the maximality by finding an item e such that $P \cup \{e\}$ is frequent. If and only if such e exists, P is not maximal. To operate this efficiently, we reduce the database. Let us consider an iteration of LCMmax with respect to a frequent itemset P. LCM algorithms reduce the database by anytime database reduction for the frequency counting. Suppose that the reduced database is composed of transactions $S_1, ..., S_h$, and each S_l is obtained by merging transactions $T_1^l, ..., T_k^l$ of the original database. Let H be the set of items to be added in the iteration. Suppose that we remove all items e from H such that $P \cup \{e\}$ is infrequent. Then, for any l, $T_1^l \cap H = T_2^l \cap H = , ..., = T_k^l \cap H$ holds. For an item e and a transaction S_l, we define the weight $w(e, S_l)$ by the number of transactions in $T_1^l, ..., T_k^l$ including e. Here the frequency of $P \cup \{e\}$ is $\sum_{S \in \{S_1, ..., S_h\}} w(e, S)$. Thus, by using the weights, we can efficiently check the maximality, in linear time of the size of the reduced database.

When we merge transactions to reduce the database, the weights can be updated easily. For each item e, we take the sum of $w(e, S)$ over all transactions S to be merged. In the same way as frequency counting, we can remove infrequent items from the database for maximality checking, for more reduction.

The computation time for maximality check in LCMmax depends on the size of database, but not on the number of previously obtained itemsets. Thus, storage method has advantages if the number of maximal frequent itemsets is small, but for the instances with a lot of maximal frequent itemsets, which take long time to be solved, LCMmax has an advantage.

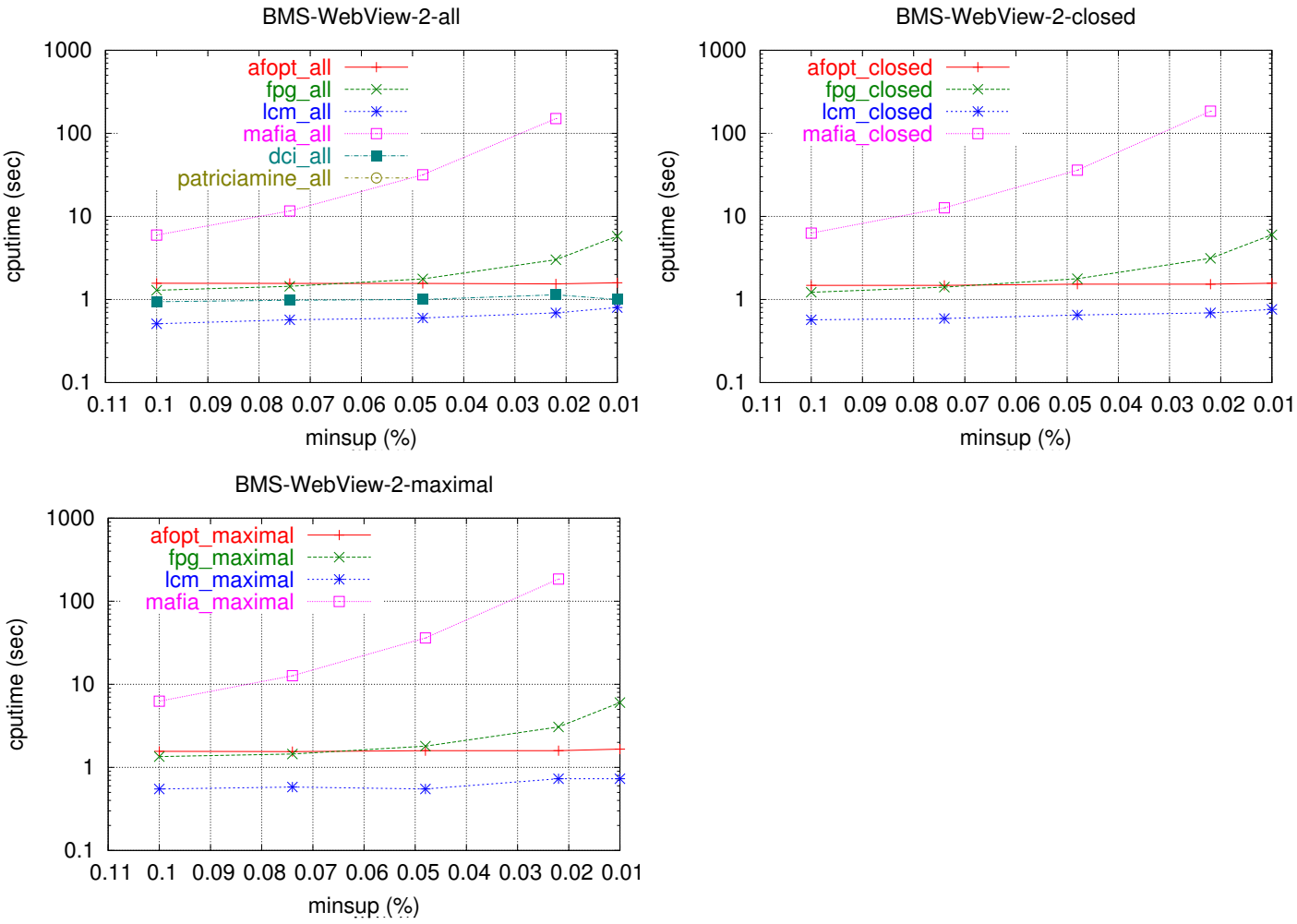

Figure 1: Results 1

4 Computational Experiments

In this section, we show the results of our computational experiments. We implemented our three algorithms LCM, LCMfreq, and LCMmax. They are coded by ANSI C, and complied by gcc. The experiments were executed on a notebook PC, with AMD athron XP 1600+ of 224MB memory. The performance of LCM algorithms are compared with the algorithms which marked good score on FIMI 03: fpgrowth[8], afopt[11], MAFIA[5, 6], kDCI[12], and PATRICIAMINE[16]. We note that kDCI and PATRICIAMINE are only for all frequent itemset mining. To reduce the time for experiments, we stop the execution when an algorithm takes more than 10 minute. The following figures show the results. We do not plot if the computation time is over 10 minutes, or abnormal terminations. The results are displayed in Figure 1 and 2. In each graph, the horizontal axis is the size of minimum supports, and the virtical axis is the CPU time written in a log scale.

From the performances of implementations, the instances were classified into three groups, in which the results are similar. Due to the space limitation, we show one instance as a representative for each group.

The first group is composed of BMS-WebView1, BMS-WebView2, BMS-POS, T10I4D100K, kosarak, and retail. These datasets have many items and transactions but are sparse. We call these datasets *sparse datasets*. We chosen BMS-WebView2 as the representative.

The second group is composed of datasets taken

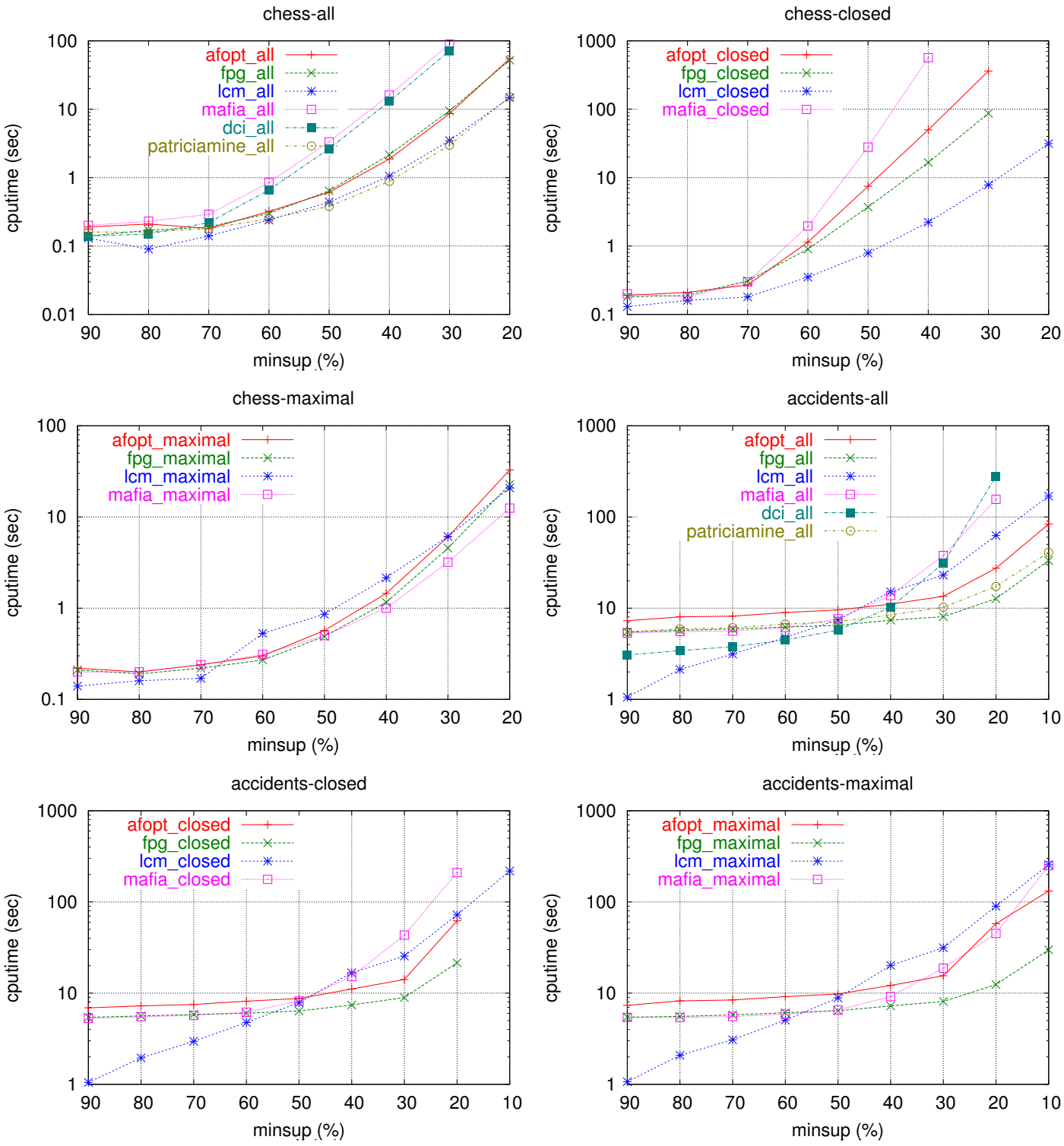

Figure 2: Results 2

from UCI-Machine Learning Repository[1], connect, chess, mushrooms, pumsb, and pumsb-star. These datasets have many transactions but few items. We call these datasets *middle density datasets*. As a representative, we show the result of chess.

The third group is accidents. It is different from any other dataset. It has huge number of transactions, but few items. Transactions includes many items, so the dataset is very dense. We call this dataset *very dense dataset*.

In almost instances and minimum supports, LCM algorithms perform well. When the minimum support is large, LCM algorithms are the fastest for all instances, because of the fast initialization. For all instances with any minimum support, LCM outperforms other closed itemset mining algorithms. This shows the efficiency of ppc extension.

For sparse datasets, LCM algorithms are the fastest, for any minimum support. The efficiency of FP-tree is not large, and occurrence deliver works efficiently. The performances of afopt and fp-growth are quite similar for these problems. They are the second bests, and 2 to 10 times slower than LCM algorithms. For enumerating frequent closed itemsets, they take much time when the number of closed itemsets is large. Although PATRICIAMINE is fast as much as fp-groth and afopt, it abnormally terminated for some instances. kDCI is slow when the number of frequent itemsets is large. MAFIA was the slowest for these instances, for any minimum support.

For middle density datasets, LCM is the fastest for all instances on closed itemset mining. On all and maximal frequent itemset mining, LCMfreq and LCMmax are the fastest for large minimum supports, for any dataset. For small minimum supports, for half instances LCMfreq and LCMmax are the fastest. For the other instances, the results are case by case: each algorithm won in some cases.

For accidents, LCM algorithms are the fastest when the minimum support is large. For small supports, LCM(closed) is the fastest, however LCMfreq and LCMmax are slower than fp-growth For this dataset, the efficiency of FP-tree is large, and the compression ratio is up to 6. Bitmap is also efficient from the density. Hence, the computation time for the frequency counting is short in the execution of existing implementations. However, by ppc extension, LCM has an advantage for closed itemset mining. hence LCM(closed) is the fastest.

[1] `http://www.ics.uci.edu/ mlearn/MLRepository.html`

5 Conclusion

In this paper, we proposed a fast implementation of LCM for enumerating frequent closed itemsets, which is based on prefix preserving closure extension. We further gave implementations LCMfreq and LCMmax for enumerating all frequent itemsets and maximal frequent itemsets by modifying LCM. We show by computational experiments that our implements of LCM, LCMfreq and LCMmax perform above the other algorithms for many datasets, especially for sparse datasets. There is a possibility of speeding up LCM algorithms by developing more efficient maximality checking algorithms, or developing a hybrid of array and FP-tree like data structures.

Acknowledgment

This research is supported by joint-research funds of National Institute of Informatics.

References

[1] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," *In Proceedings of VLDB '94,* pp. 487–499, 1994.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen and A. I. Verkamo, "Fast Discovery of Association Rules," *In Advances in Knowledge Discovery and Data Mining,* MIT Press, pp. 307–328, 1996.

[3] R. J. Bayardo Jr., "Efficiently Mining Long Patterns from Databases", In *Proc. SIGMOD'98*, pp. 85–93, 1998.

[4] E. Boros, V. Gurvich, L. Khachiyan, and K. Makino, "On the Complexity of Generating Maximal Frequent and Minimal Infrequent Sets," *STACS 2002*, pp. 133-141, 2002.

[5] D. Burdick, M. Calimlim, J. Gehrke, "MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases," In *Proc. ICDE 2001*, pp. 443-452, 2001.

[6] D. Burdick, M. Calimlim, J. Flannick, J. Gehrke, and T. Yiu, "MAFIA: A Performance Study of Mining Maximal Frequent Itemsets," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003. (Available as CEUR Workshop Proc. series, Vol. 90, http://ceur-ws.org/vol-90)

[7] B. Goethals, *the FIMI'03 Homepage*, `http://fimi.cs.helsinki.fi/`, 2003.

[8] G. Grahne and J. Zhu, "Efficiently Using Prefix-trees in Mining Frequent Itemsets," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003. (Available as CEUR Workshop Proc. series, Vol. 90, http://ceur-ws.org/vol-90)

[9] J. Han, J. Pei, Y. Yin, "Mining Frequent Patterns without Candidate Generation," *SIGMOD Conference 2000*, pp. 1-12, 2000

[10] R. Kohavi, C. E. Brodley, B. Frasca, L. Mason and Z. Zheng, "KDD-Cup 2000 Organizers' Report: Peeling the Onion," *SIGKDD Explorations,* 2(2), pp. 86-98, 2000.

[11] Guimei Liu, Hongjun Lu, Jeffrey Xu Yu, Wang Wei, and Xiangye Xiao, "AFOPT: An Efficient Implementation of Pattern Growth Approach," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003. (Available as CEUR Workshop Proc. series, Vol. 90, http://ceur-ws.org/vol-90)

[12] S. Orlando, C. Lucchese, P. Palmerini, R. Perego and F. Silvestri, "kDCI: a Multi-Strategy Algorithm for Mining Frequent Sets," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003. (Available as CEUR Workshop Proc. series, Vol. 90, http://ceur-ws.org/vol-90)

[13] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, *Efficient Mining of Association Rules Using Closed Itemset Lattices*, *Inform. Syst.*, 24(1), 25–46, 1999.

[14] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, *Discovering Frequent Closed Itemsets for Association Rules*, In *Proc. ICDT'99*, 398-416, 1999.

[15] J. Pei, J. Han, R. Mao, "CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets," *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery 2000*, pp. 21-30, 2000.

[16] A. Pietracaprina and D. Zandolin, "Mining Frequent Itemsets using Patricia Tries," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003. (Available as CEUR Workshop Proc. series, Vol. 90, http://ceur-ws.org/vol-90)

[17] S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirakawa, "A New Algorithm for Generating All the Maximum Independent Sets," *SIAM Journal on Computing,* Vol. 6, pp. 505–517, 1977.

[18] T. Uno, T. Asai, Y. Uchida, H. Arimura, "LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets," In *Proc. IEEE ICDM'03 Workshop FIMI'03*, 2003. (Available as CEUR Workshop Proc. series, Vol. 90, http://ceur-ws.org/vol-90)

[19] T. Uno, T. Asai, Y. Uchida, H. Arimura, "An Efficient Algorithm for Enumerating Closed Patterns in Transaction Databases," to appear in Proc. of Discovery Science 2004, 2004.

[20] M. J. Zaki, C. Hsiao, "CHARM: An Efficient Algorithm for Closed Itemset Mining," *2nd SIAM International Conference on Data Mining (SDM'02)*, pp. 457-473, 2002.

[21] Z. Zheng, R. Kohavi and L. Mason, "Real World Performance of Association Rule Algorithms," *KDD 2001*, pp. 401-406, 2000.