LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets

Takeaki Uno[1], Tatsuya Asai[2], Yuzo Uchida[2], Hiroki Arimura[2]
[1] National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430, Japan
e-mail: uno@nii.jp
[2] Department of Informatics, Kyushu University, 6-10-1 Hakozaki, Fukuoka 812-0053, JAPAN
e-mail:{t-asai,y-uchida,arim}@i.kyushu-u.ac.jp

Abstract:

In this paper, we propose three algorithms LCM-freq, LCM, and LCMmax for mining all frequent sets, frequent closed item sets, and maximal frequent sets, respectively, from transaction databases. The main theoretical contribution is that we construct tree-shaped transversal routes composed of only frequent closed item sets, which is induced by a parent-child relationship defined on frequent closed item sets. By traversing the route in a depth-first manner, LCM finds all frequent closed item sets in polynomial time per item set, without storing previously obtained closed item sets in memory. Moreover, we introduce several algorithmic techniques using the sparse and dense structures of input data. Algorithms for enumerating all frequent item sets and maximal frequent item sets are obtained from LCM as its variants. By computational experiments on real world and synthetic databases to compare their performance to the previous algorithms, we found that our algorithms are fast on large real world datasets with natural distributions such as KDD-cup2000 datasets, and many other synthetic databases.

1. Introduction

Frequent item set mining is one of the fundamental problems in data mining and has many applications such as association rule mining [1], inductive databases [9], and query expansion [12].

Let E be the universe of *items*, consisting of items $1, ..., n$. A subset X of E is called an *item set*. $\mathcal{T}$ is a set of *transactions* over E, i.e., each $T \in \mathcal{T}$ is composed of items of E. For an item set X, let $\mathcal{T}(X) = \{ t \in \mathcal{T} \mid X \subseteq t \}$ be the set of transactions including X. Each transaction of $\mathcal{T}(X)$ is called an *occurrence* of X. For a given constant $\alpha \geq 0$, an item set X is called *frequent* if $|\mathcal{T}(X)| \geq \alpha$. If a frequent item set is included in no other frequent set, it is said to be *maximal*. For a transaction set $\mathcal{S} \subseteq \mathcal{T}$, let $I(\mathcal{S}) = \bigcap_{T \in \mathcal{S}} T$. If an item set X satisfies $I(\mathcal{T}(X)) = X$, then X is called a *closed item set*. We denote by $\mathcal{F}$ and $\mathcal{C}$ the sets of all frequent itemsets and all frequent closed item sets, respectively.

In this paper, we propose an efficient algorithm LCM for enumerating all frequent closed item sets. LCM is an abbreviation of *Linear time Closed item set Miner*. Existing algorithms for this task basically enumerate frequent item sets with cutting off unnecessary frequent item sets by pruning. However, the pruning is not complete, hence the algorithms operate unnecessary frequent item sets, and do something more. In LCM, we define a parent-child relationship between frequent closed item sets. The relationship induces tree-shaped transversal routes composed only of all the frequent closed item sets. Our algorithm traverses the routes, hence takes linear time of the number of frequent closed item sets. This algorithm is obtained from the algorithms for enumerating maximal bipartite cliques [14, 15], which is designed based on reverse search technique [3, 16].

In addition to the search tree technique for closed item sets, we use several techniques to speed-up the update of the occurrences of item sets. One technique is *occurrence deliver*, which simutaneously computes the occurrence sets of all the successors of the current item set during a single scan on the current occurrence set. The other is *diffsets* proposed in [18]. Since there is a trade-off between these two methods that the former is fast for sparse data while the latter is fast for dense data, we developed the *hybrid algorithm* combining them. In some iterations, we make a decision based of the estimation of their computation time, hence our algorithm can use appropriate one for dense parts and sparse parts of the input.

We also consider the problems of enumerating all frequent sets, and maximal frequent sets, and derive two algorithms LCMfreq and LCMmax from LCM. LCMmax is obtained from LCM by adding the explicit check of maximality. LCMfreq is not merely a LCM without the check of closedness, but also achives substantial speed-up using closed itemset discovery techniques because it enumerates only the representatives of groups of frequent item sets, and generate other frequent item sets from the representatives.

From computer experiments on real and artificial datasets with the previous algorithms, we observed that our algorithms LCMfreq, LCM, and LCMmax significantly outperform the previous algorithms on real world datasets with natural distributions such as BMS-Web-View-1 and BMS-POS datasets in the KDD-CUP 2000 datasets as well as large synthesis datasets such as IBM T10K4D100K. The performance of our algorithms is similar to other algorithms for hard datasets such as Connect and Chess datasets from UCI-Machine Learning Repository, but less significant than MAFIA, however LCM works with small memory rather than other algorithms.

The organization of the paper is as follows. In Section 2, we explain our tree enumeration method for frequent closed item sets and our algorithm LCM. In Section 3, we describe several algorithmic techniques for speeding up and saving memory. Then, Section 4 and 5 give LCMmax and LCMfreq for maximal and all frequent item sets, respectively. Techniques for implementation is described in Section 6, and the results of computational experiments are reported in Section 7. Finally, we conclude in Section 8.

2. Enumerating Frequent Closed Item Sets

In this section, we introduce a parent-child relationship between frequent closed item sets in $\mathcal{C}$, and describe our algorithm LCM for enumeration them.

Recent efficient algorithms for frequent item sets, e.g.,[4, 17, 18], use a tree-shaped search structure for $\mathcal{F}$, called the *set enumeration tree* [4] defined as follows. Let $X = \{x_1, \ldots, x_n\}$ be an itemset as an ordered sequence such that $x_1 < \cdots < x_n$, where the *tail* of X is $tail(X) = x_n \in E$. Let X, Y be item sets. For an index i, $X(i) = X \cap \{1, \ldots, i\}$. X is a

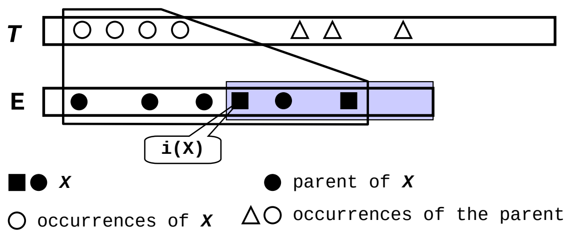

Figure 1: An example of the parent of X: The parent of X is obtained by deleting items larger than $i(X)$ (in the gray area) and take a closure.

prefix of Y if $X = Y(i)$ holds for $i = tail(X)$. Then, the parent-child relation $\mathcal{P}$ for the set enumeration tree for $\mathcal{F}$ is define as $X = \mathcal{P}(Y)$ iff $Y = X \cup \{i\}$ for some $i > tail(X)$, or equivalently, $X = Y \backslash \{tail(Y)\}$. Then, the whole search space for $\mathcal{F}$ forms a *prefix tree* (or *trie*) with this edge relation $\mathcal{P}$.

Now, we define the parent-child relation $\mathcal{P}$ for closed item sets in $\mathcal{C}$ as follows. For $X \in \mathcal{C}$, we define the *parent* of X by $\mathcal{P}(X) = I(\mathcal{T}(X(i(X) - 1)))$, where $i(X)$ be the minimum item i such that $\mathcal{T}(X) = \mathcal{T}(X(i))$ but $\mathcal{T}(X) \neq \mathcal{T}(X(i-1))$. If Y is the parent of X, we say X is a *child* of Y. Let $\perp = I(\mathcal{T}(\emptyset))$ be the smallest item set in $\mathcal{C}$ called the *root*. For any $X \in \mathcal{C} \setminus \{\perp\}$, its parent $\mathcal{P}(X)$ is always defined and belongs to $\mathcal{C}$. An illustration is given in Fig. 1.

For any $X \in \mathcal{C}$ and its parent Y, the proper inclusion $Y \subset X$ holds since $\mathcal{T}(X(i(X) - 1)) \subset \mathcal{T}(X)$. Thus, the relation $\mathcal{P}$ is acyclic, and its graph representation forms a tree. By traversing the tree in a depth-first manner, we can enumerate all the frequent closed item sets in linear time in the size of the tree, which is equal to the number of the frequent closed item sets in $\mathcal{C}$. In addition, we need not store the tree in memory. Starting from the root $\perp$, we find a child X of the root, and go to X. In the same way, we go to a child of X. When we arrive at a leaf of the tree, we backtrack, and find another child. Repeating this process, we eventually find all frequent closed item set in $\mathcal{C}$.

To find the children of the current frequent closed item set, we use the following lemma. For an item set X and an index i, let $X[i] = X \cup H$ where H is the set of the items $j \in I(\mathcal{T}(X \cup \{i\}))$ satisfying $j \geq i$.

Lemma 1 *X' is a child of $X \in \mathcal{C}$ ($X' \in \mathcal{C}$ and the parent of X' is X) if and only if*
(cond1) $X' = X[i]$ for some $i > i(X)$,
(cond2) $X' = I(\mathcal{T}(X'))$ (X' is a closed item set)
(cond3) X' is frequent

Proof : Suppose that $X' = X[i]$ satisfies the conditions (cond1), (cond2) and (cond3). Then, $X' \in \mathcal{C}$. Since $\mathcal{T}(X(i-1)) = \mathcal{T}(X)$ and $\mathcal{T}(X(i-1) \cup \{i\}) = \mathcal{T}(X')$ holds thus $i(X') = i$ holds. Hence, X' is a child of X. Suppose that X' is a child of X. Then, (cond2) and (cond3) hold. From the definition of $i(X')$, $\mathcal{T}(X(i(X')) \cup \{i(X')\}) = \mathcal{T}(X')$ holds. Hence, $X' = X[i(X')]$ holds. We also have $i(X') > i(X)$ since $\mathcal{T}(X'(i(X')-1)) = \mathcal{T}(X)$. Hence (cond1) holds. ∎

Clearly, $\mathcal{T}(X[i]) = \mathcal{T}(X \cup \{i\})$ holds if (cond2) $I(\mathcal{T}(X[i])) = X[i]$ holds. Note that no child X' of X satisfies $X[i] = X[i'], i \neq i'$, since the minimum item of $X[i] \setminus X$ and $X[i'] \setminus X$ are i and i', respectively. Using this lemma, we construct the following algorithm scheme for, given a closed itemset X, enumerating all descendants in the search tree for closed itemsets.

```
Algorithm LCM ($X$ : frequent closed item set)
1. output $X$
2. For each $i > i(X)$ do
3.   If $X[i]$ is frequent and $X[i] = I(\mathcal{T}(X[i]))$ then
       Call LCM( $X[i]$ )
4. End for
```

Theorem 1 *Let $0 < \sigma < 1$ be a minimum support. Algorithm LCM enumerates, given the root closed item set $\perp = I(\mathcal{T}(\emptyset))$, all frequent closed item sets in linear time in the number of frequent closed item sets in $\mathcal{C}$.* ∎

The existing enumeration algorithm for frequent closed item sets are based on backtrack algorithm, which traverse a tree composed of all frequent item sets in $\mathcal{F}$, and skip some item sets by pruning the tree. Since the pruning is not complete, however, these algorithms generate unnecessary frequent item sets. On the other hand, the algorithm in [10] directly generates only closed item sets with the closure operation $I(\mathcal{T}(\cdot))$ as ours, but their method may generate duplicated closed item sets and needs expensive duplicate check.

On the other hand, our algorithm traverses a tree composed only of frequent closed item sets, and each iteration is not as heavy as the previous algorithms. Hence, our algorithm runs fast in practice. If we consider our algorithm as a modification of usual backtracking algorithm, each iteration of our algorithm re-orders the items larger than $i(X)$ such that the items not included in X follow the items included in X. Note that the parent X is not a prefix of $X[i]$ in a recursive call. The check of (cond2) can be considered as a pruning of non-closed item sets.

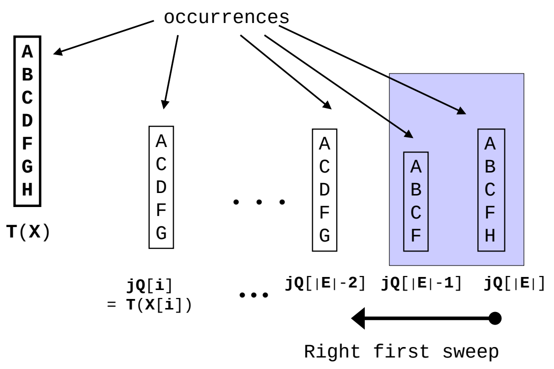

Figure 2: Occurrence deliver and right first sweep: In the figure, $\mathcal{J}[i]$ is written as $JQ[i]$. For each occurrence T of X, occurrence deliver inserts T to $\mathcal{J}[i]$ such that $i \in T$. When the algorithm generates a recursive call respect to $X[|E|-2]$, the recursive calls respect to $X[|E|-1]$ and $X[|E|]$ have been terminated, $\mathcal{J}[|E|-1]$ and $\mathcal{J}[|E|]$ are cleared. The recursive call of $X[|E|-2]$ uses only $\mathcal{J}[|E|-1]$ and $\mathcal{J}[|E|]$, and hence the algorithm re-uses them in the recursive call.

3. Reducing Computation Time

The computation time of LCM described in the previous section is linear in $|\mathcal{C}|$, with a factor depending on $\mathcal{T}(X)$ for each closed item set $X \in \mathcal{C}$. However, this still takes long time if it is implemented in a straightforward way. In this section, we introduce some techniques based on sparse and dense structures of the input data.

Occurrence Deliver. First, We introduce the technique called the *occurrence deliver* for reducing the construction time for $\mathcal{T}(X[i])$, which is needed to check (cond3). This technique is particularly efficient in the case that $|\mathcal{T}(X[i])|$ is much smaller than $|\mathcal{T}(X)|$. In a usual way, $\mathcal{T}(X[i])$ is obtained from $\mathcal{T}(X)$ in $O(|\mathcal{T}(X)|)$ time by removing all transactions not including i based on the equiality $\mathcal{T}(X[i]) = \mathcal{T}(X \cup \{i\}) = \mathcal{T}(X) \cap \mathcal{T}(\{i\})$ (this method is known as *down-project*). Thus, the total computation for all children takes $|E|$ scans and $O(|\mathcal{T}(X)| \cdot |E|)$ time.

Instead of this, we build for all $i = i(X), \ldots, |E|$ the occurrence lists $\mathcal{J}[i] \overset{\text{def}}{=} \mathcal{T}(X[i])$ simultaneously by scanning the transactions in $\mathcal{T}(X)$ at once as follows. We initialize $\mathcal{J}[i] = \emptyset$ for all $i = i(X), \ldots, |E|$. For each $T \in \mathcal{T}(X)$ and for each $i \in T$ $(i > i(X))$, we insert T to $\mathcal{J}[i]$. See Fig. 2 for explanation, where we write $jQ[i]$ for $\mathcal{J}[i]$. This correctly computes $\mathcal{J}[i]$ for all i in the total time $O(|\mathcal{T}(X)|)$. Furthermore, we need not make recursive call of **LCM** for $X[i]$ if $\mathcal{T}(X[i]) = \emptyset$ (this is often called *lookahead* [4]). In our experiments on BMS instances, the occurrence deliver reduces the computation time up to 1/10 in some cases.

Right-first sweep. The occurrence deliver method needs eager computation of the occurrence sets $\mathcal{J}[i] = \mathcal{T}(X[i])$ for all children before expanding one of them. A simple implementation of it may require much memory than the ordinary lazy computation of $\mathcal{T}(X[i])$ as in [17]. However, we can reduce the memory usage using a method called the *right-first sweep* as follows.

Given a parent X, we make the recursive call for $X[i]$ in the decreasing order for each $i = |E|, \ldots, i(X)$ (See Fig. 2). At each call of $X[i]$, we collect the memory allocated before for $\mathcal{J}[i+1], \ldots, \mathcal{J}[|E|]$ and then re-use it for $\mathcal{J}[i]$. After terminating the call for $X[i]$, the memory for $\mathcal{J}[i]$ is released for the future use in $\mathcal{J}[j]$ for $j < i$. Since $|\mathcal{J}[i]| = |\mathcal{T}(X[i])| \leq |\mathcal{T}(\{i\})|$ for any i and X, the total memory $\sum_i |\mathcal{J}[i]|$ is bounded by the input size $||\mathcal{T}|| = \sum_{T \in \mathcal{T}} |T|$, and thus, it is sufficient to allocate the memory for $\mathcal{J}$ at once as a global variable.

Diffsets. In the case that $|\mathcal{T}(X[i])|$ is nearly equal to $|\mathcal{T}(X)|$ we use the *diffset* technique proposed in [18]. The diffset for index i is $\mathcal{DJ}[i] = \mathcal{T}(X) \setminus \mathcal{T}(X[i])$, where $\mathcal{T}(X[i]) = \mathcal{T}(X \cup \{i\})$. Then, the frequency of $X[i]$ is obtained by $|\mathcal{T}(X[i])| = |\mathcal{T}(X)| - |\mathcal{DJ}[i]|$. When we generate a recursive call respect to $X[i]$, we update $\mathcal{DJ}[j], j > i$ by setting $\mathcal{DJ}[j]$ to be $\mathcal{DJ}[j] \setminus \mathcal{DJ}[i]$ in time $O(\sum_{i > i(X), X[i] \in \mathcal{F}}((|\mathcal{T}(X)| - |\mathcal{T}(X[i])|)$. Diffsets are needed for only i such that $X[i]$ is frequent. By diffsets, the computation time for instances such as connect, chess, pumsb are reduced to 1/100, where $|\mathcal{T}(X[i])|$ is as large as $|\mathcal{T}(X)|$.

Hybrid Computation. As we saw in the preceding subsections, our occurrence deliver is fast when $|\mathcal{T}(X[i])|$ is much smaller than $|\mathcal{T}(X)|$ while the diffset of [18] is fast when $|\mathcal{T}(X[i])|$ is nearly close to $|\mathcal{T}(X)|$. Therefore, our LCM dinamically decides which of occurrence deliver and diffsets we will use. To do this, we compare two quantities on X:

$A(X) = \sum_i |\mathcal{T}(X \cup \{i\})|$ and

$B(X) = \sum_{i: X \cup \{i\} \in \mathcal{F}} (|\mathcal{T}(X)| - |\mathcal{T}(X \cup \{i\})|)$.

For some fixed constant $\alpha > 1$, we decide to use the occurrence deliver if $A(X) < \alpha B(X)$ and the diffset otherwise. We make this decision only at the child iterations of the root set $\perp$ since this decision takes much time. Empirically, restricting the range $i \in \{1, \ldots, |E|\}$ of the the index i in $A(X)$ and $B(X)$ to $i \in \{i(X)+1, \ldots, |E|\}$ results significant speed-up. By experiments on BMS instances, we observe that the hybrid technique reduces the computation time up to 1/3. The hybrid technique is also useful in reducing the memory space in diffset as follows. Although the memory $B(X)$ used by diffsets is not bounded by the input size $||\mathcal{T}||$ in the worst case, it is ensured in hybrid that $B(X)$ does not exceed $A(X) \leq ||\mathcal{T}||$ because the diffset is chosen only when $A(X) \geq \alpha B(X)$.

Checking the closedness in occrrence deliver. Another key is to efficiently check the closedness $X[i] = I(\mathcal{T}(X[i]))$ (cond 2). The straightforward computation of the closure $I(\mathcal{T}(X[i]))$ takes much time since it requires the access to the whole sets $\mathcal{T}(X[j]), j < i$ and i is usually as large as $|E|$.

By definition, (cond 2) is violated iff there exists some $j \in \{1, \ldots, i-1\}$ such that $j \in T$ for every $T \in \mathcal{T}(X \cup \{i\})$. We first choose a transaction $T^*(\cup\{i\}) \in \mathcal{T}(X \cup \{i\})$ of minimum size, and tests if $j \in T$ for increasing $j \in T^*(\cup\{i\})$. This results $O(\sum_{j \in T^*(X \cup \{i\})} m(X[i], j))$ time algorithm, where $m(X', j)$ is the maximum index m such that all of the first m transactions of $\mathcal{T}(X')$ include j, which is much faster than the straightforward algorithm with $O(\sum_{j<i} |\mathcal{T}(X \cup \{i\} \cup \{j\}])|)$ time.

In fact, the efficient check requires the adjacency matrix (sometime called *bitmap*) representing the inclusion relationship between items and transactions. However, the adjacency matrix requires $O(|\mathcal{T}| \times |E|)$ memory, which is quite hard to store for large instances. Hence, we make columns of adjacency matrix for only transactions of size larger than $(\sum_{T \in \mathcal{T}} |T|)/\delta$. Here δ is a constant. This uses at most $O(\delta \times \sum_{T \in \mathcal{T}} |T|)$, linear in the input size.

Checking the closedness in diffsets. In the case that $|\mathcal{T}(X[i])|$ is nearly equal to $|\mathcal{T}(X)|$, the above check is not done in short time. In this case, we keep diffset $\mathcal{DJ}[j]$ for all $j < i, i \notin X$ such that $X[i]$ is frequent. To maintain $\mathcal{DJ}$ for all i is a heavy task, thus we discard unnecessary $\mathcal{DJ}$'s as follows. If $\mathcal{T}(X \cup \{j\})$ includes an item included in no $\mathcal{T}(X[i']), i' > i(X)$, then for any descendant X' of X, $j \notin I(\mathcal{T}(X'[j']))$ for any $j' > i(X')$. Hence, we no longer have to keep $\mathcal{DJ}[j]$ for such j. Let $NC(X)$ be the set of items j such that $X[j]$ is frequent and any item of $\mathcal{T}(X) \setminus \mathcal{T}(X \cup \{j\})$ is included in some $\mathcal{T}(X[j']), j' > i(X)$. Then, the computation time for checking (cond2) is written as $O(\sum_{j \in NC(X), j < i} |\mathcal{T}(X) \setminus \mathcal{T}(X \cup \{j\})|)$. By checking (cond2) in these ways, the computation time for checking (cond2) is reduced from 1/10 to 1/100.

Detailed Algorithm. We present below the description of the algorithm **LCM**, which recursively computes $(X, \mathcal{T}(X), i(X))$, simultaneously.

```
global: $\mathcal{J}, \mathcal{DJ}$   /* Global sets of lists */
Algorithm LCM()
1. $X := I(\mathcal{T}(\emptyset))$   /* The root $\perp$ */
2. For $i := 1$ to $|E|$
3.   If $X[i]$ satisfies (cond2) and (cond3) then
       Call LCM_Iter( $X[i], \mathcal{T}(X[i]), i$ ) or
       Call LCMd_Iter2( $X[i], \mathcal{T}(X[i]), i, \mathcal{DJ}$ )
         based on the decision criteria
4. End for

LCM_Iter( $X, \mathcal{T}(X), i(X)$ )   /* occurrence deliver */
1. output $X$
2. For each $T \in \mathcal{T}(X)$
     For each $j \in T, j > i(X)$, insert $t$ to $\mathcal{J}[j]$
4. For each $j, \mathcal{J}[j] \neq \emptyset$ in the decreasing order
5.   If $|\mathcal{J}[j]| \geq \alpha$ and (cond2) holds then
       LCM_Iter( $\mathcal{T}(\mathcal{J}[j]), \mathcal{J}[j], j$ )
6.   Delete $\mathcal{J}[j]$
7. End for

LCM_Iter2( $X, \mathcal{T}(X), i(X), \mathcal{DJ}$ )   /* diffset */
1. output $X$
2. For each $i, X[i]$ is frequent
3.   If $X[i]$ satisfies (cond2) then
4.     For each $j, X[i] \cup \{j\}$ is frequent,
         $\mathcal{DJ}'[j] := \mathcal{DJ}[j] \setminus \mathcal{DJ}[i]$
5.     LCM_Iter2( $\mathcal{T}(\mathcal{J}[j]), \mathcal{J}[j], j, \mathcal{DJ}'$ )
6.   End if
7. End for
```

Theorem 2 *Algorithm LCM enumerates all frequent closed item sets in* $O(\sum_{j>i(X)} |\mathcal{T}(X[j])| + \sum_{j>i(X), X[j]\in\mathcal{F}} \sum_{j'\in T^*(X)} m(X[j], j'))$ *time, or* $O(\sum_{i>i(X), X[i]\in\mathcal{F}}((|\mathcal{T}(X)| - |\mathcal{T}(X[i])|) + \sum_{j\in NC(X), j<i} |\mathcal{T}(X) \setminus \mathcal{T}(X \cup \{j\})|))$ *time for each frequent closed item set* X*, with memory linear to the input size.* ▮

4. Enumerating Maximal Frequent Sets

In this section, we explain an enumeration algorithm of maximal frequent sets with the use of frequent closed item set enumeration. The main idea is very simple. Since any maximal frequent item set is a frequent closed item set, we enumerate frequent closed item sets and output only those being maximal frequent sets. For a frequent closed item set X, X is a maximal frequent set if and only if $X \cup \{i\}$ is infrequent for any $i \notin X$. By adding this check to LCM, we obtain LCMmax.

This modification does not increase the memory complexity but increase the computation time. In the case of occurrence deliver, we generate $\mathcal{T}(X \cup \{j\})$ for all j in the same way as the occurrence deliver, and check the maximality. This takes $O(\sum_{j<i(X)} |\mathcal{T}(X \cup \{j\}|)$ time. In the case of difference update, we do not discard diffsets unnecessary for closed item set enumeration. We keep diffsets $\mathcal{DJ}$ for all j such that $X \cup \{j\}$ is frequent. To update and maintain this, we spend $O(\sum_{j, X\cup\{j\}\in\mathcal{F}} |\mathcal{T}(X) \setminus \mathcal{T}(X \cup \{j\})|)$ time. Note that we are not in need of check the maximality if X has a child.

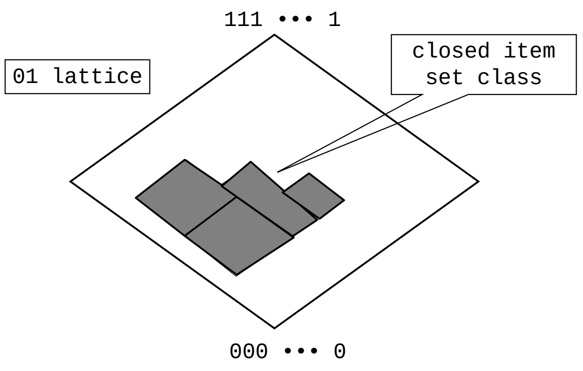

Figure 3: Hypercube decomposition: **LCMfreq** decomposes a closed item set class into several sublattices (gray rectangles).

Theorem 3 *Algorithm LCMmax enumerates all maximal frequent item sets in* $O(\sum_i |\mathcal{T}(X \cup \{i\})|)$ *time, or* $O(\sum_{i, X\cup\{i\}\in\mathcal{F}}((|\mathcal{T}(X)| - |\mathcal{T}(X \cup \{i\})|))$ *time for each frequent closed item set* X*, with memory linear in the input size.* ▮

5. Enumerating Frequent Sets

In this section, we describe an enumeration algorithm for frequent item sets. The key idea of our algorithm is that we classify the frequent item sets into groups and enumerate the representative of each group. Each group is composed of frequent item sets included in the class of a closed item set. This idea is based on the following lemma.

Lemma 2 *Suppose that frequent item sets* X *and* $S \supset X$ *satisfy* $\mathcal{T}(X) = \mathcal{T}(S)$*. Then, for any item set* X' *including* X*,* $\mathcal{T}(X') = \mathcal{T}(X' \cup S)$*.* ▮

Particularly, $\mathcal{T}(X') = \mathcal{T}(R)$ holds for any $X' \subseteq R \subseteq X' \cup S$, hence all R are included in the same class of a closed item set. Hence, any frequent item set X'

is generated from $X' \setminus (S \setminus X)$. We call $X' \setminus (S \setminus X)$ *representative.*

Let us consider a backtracking algorithm finding frequent item sets which adds items one by one in lexicographical order. Suppose that we currently have a frequent item set X, and find another frequent item set $X \cup \{i\}$. Let $S = X[i]$. Then, according to the above lemma, we can observe that for any frequent item set X' including X and not intersecting $S \setminus X$, any item set including X' and included in $X' \cup S$ is also frequent. Conversely, any frequent item set including X is generated from X' not intersecting $S \setminus X$. Hence, we enumerate only representatives including X and not intersecting $S \setminus X$, and generate other frequent item sets by adding each subset of $S \setminus X$. This method can be considered that we "decompose" classes of closed item sets into several sublattices (hypercubes) each of whose maximal and minimal elements are S and X', respectively (see Fig. 3). We call this technique *hypercube decomposition.*

Suppose that we are currently operating a representative X' including X, and going to generate a recursive call respect to $X' \cup \{j\}$. Then, if $(X'[i] \setminus X') \setminus S \neq \emptyset$, X' and $S \cup (X'[i] \setminus X')$ satisfies the condition of Lemma 2. Hence, we add $X'[i] \setminus X'$ to S.

We describe LCMfreq as follows.

```
Algorithm LCMfreq ( $X$ : representative,
                    $S$ : item set, $i$ : item )
1. Output all item sets $R, X \subseteq R \subseteq X \cup S$
2. For each $j > i, j \notin X \cup S$
3.   If $X \cup \{j\}$ is frequent then
       Call LCMfreq ( $X \cup \{j\}, S \cup (X[j] \setminus (X \cup \{j\})), j$)
4. End for
```

For some synthetic instances such that frequent closed item sets are fewer than frequent item sets, the average size of S is up to 5. In these cases, the algorithm finds $2^{|S|} = 32$ frequent item sets at once, hence the computation time is reduced much by the improvement.

To check the frequency of all $X \cup \{j\}$, we can use occurrence deliver and diffsets used for LCM. LCMfreq does not require the check of (cond2), hence The computation time of each iteration is $O(\sum_{j>i(X)} |\mathcal{T}(X[j])|)$ time for occurrence deliver, and $O(\sum_{j>i(X),X[j]\in\mathcal{F}} |\mathcal{T}(X) \setminus \mathcal{T}(X[j])|)$ for diffsets. Since the computation time change, we use another estimators for hybrid. In almost all cases, if once $\sum_{j>i(X),X[j]\in\mathcal{F}} |\mathcal{T}(X) \setminus \mathcal{T}(X[j])|$ becomes smaller than $\sum_{j>i(X)} |\mathcal{T}(X[j])|$, the condition holds in any iteration generated by a recursive call. Hence, the algorithm first starts with occurrence deliver, and compares them in each iteration. If $\sum_{j>i(X),X[j]\in\mathcal{F}} |\mathcal{T}(X) \setminus \mathcal{T}(X[j])|$ becomes smaller, then we change to diffsets. Note that these estimators can computed in short time by using the result of occurrence deliver.

Theorem 4 *LCMfreq enumerates all frequent sets of $\mathcal{F}$ in $O(\sum_{j>i(X)} |\mathcal{T}(X[j])|)$ time or $O(\sum_{j>i(X),X[j]\in\mathcal{F}} |\mathcal{T}(X) \setminus \mathcal{T}(X[j])|)$ time for each frequent set X, within $O(\sum_{T\in\mathcal{T}} |T|)$ space.* ∎

Particularly, LCMfreq requires one integer for each item of any transaction, which is required to store the input data. Other memory LCMfreq uses is bounded by $O(|\mathcal{T}| + |E|)$.

Experimentally, an iteration of LCMfreq inputting frequent set X takes $O(|\mathcal{T}(X)| + |X|)$ or O((size of diffset) + $|X|$) steps in average. In some sense, this is optimal since we have to take $O(|X|)$ time to output, and $O(|\mathcal{T}(X)|)$ time or O((size of diffset)) time to check the frequency of X.

6. Implementation

In this section, we explain our implementation. First, we explain the data structure of our algorithm. A transaction T of input data is stored by an array with length $|T|$. Each cell of the array stores the index of an item of T. For example, $t = \{4, 2, 7\}$ is stored in an array with 3 cells, [2, 4, 7]. We sort the elements of the array so that we can take $\{i, ..., |E|\} \cap T$ in linear time of $\{i, ..., |E|\} \cap T$. $\mathcal{J}$ is also stored in arrays in the same way. We are not in need of doubly linked lists or binary trees, which take much time to be operated.

To reduce the practical computation time, we sort the transactions by their sizes, and items by the number of transactions including them. Experimentally, this reduces $\sum_{j>i(X)} |\mathcal{T}(X \cup \{j\})|$. In some cases, the computation time has been reduced by a factor of 1/3.

7. Computational Experiments

To examine the practical efficiency of our algorithms, we run the experiments on the real and synthetic datasets, which are made available on FIMI'03 site. In the following, we will report the results of the experiments.

Table 1: The datasets. AvTrSz means the average transaction size

Dataset	#items	#Trans	AvTrSz	#FI	#FCI	#MFI	Minsup (%)
BMS-Web-View1	497	59,602	2.51	3.9K–NA	3.9K–1241K	2.1K–129.4K	0.1–0.01
BMS-Web-View2	3,340	77,512	4.62	24K–9897K	23K–755K	3.9K–118K	0.1–0.01
BMS-POS	1,657	517,255	6.5	122K–33400K	122K–21885K	30K–4280K	0.1–0.01
T10I4D100K	1,000	100,000	10.0	15K–335K	14K–229K	7.9K–114K	0.15–0.025
T40I10D100K	1,000	100,000	39.6	-	-	-	2–0.5
pumsb	7,117	49,046	74.0	-	-	-	95–60
pumsb_star	7,117	49,046	50.0	-	-	-	50–10
mushroom	120	8,124	23.0	-	-	-	20–0.1
connect	130	67,577	43.0	-	-	-	95–40
chess	76	3196	37.0	-	-	-	90–30

7.1 Datasets and Methods

We implemented our algorithms our three algorithms LCMfreq (LCMfreq), LCM (LCM), LCMmax (LCMmax) in C and compiled with gcc3.2.

The algorithms were tested on the datasets shown in Table 1. available from the FIMI'03 homepage[1], which include: T10I4D100K, T40I10D100K from IBM Almaden Quest research group; chess, connect, mushroom, pumsb, pumsb_star from UCI ML repository[2] and PUMSB; BMS-WebView-1, BMS-WebView-2, BMS-POS from KDD-CUP 2000[3].

We compare our algorithms LCMfreq, LCM, LCMmax with the following frequent item set mining algorithms: Implementations of Fp-growth [7], Eclat [17], Apriori [1, 2] by Bart Goethals [4]; We also compare the LCM algorithms with the implementation of Mafia [6], a fast maximal frequent pattern miner, by University of Cornell's Database group [5]. This versions of mafia with frequent item sets, frequent closed item sets, and maximal frequent item sets options are denoted by mafia-fi, mafia-fci, mafia-mfi, respectively. Although we have also planned to make the performance comparison with Charm, the state-of-the-art frequent closed item set miner, we gave up the comparison in this time due to the time constraint.

All experiments were run on a PC with the configuration of Pen4 2.8GHz, 1GB memory, and RPM 7200 hard disk of 180GB. In the experiments, LCMfreq, LCM and LCMmax use at most 123MB, 300MB, and 300MB of memory, resp. Note that LCM and LCMmax can save the memory use by decreasing δ.

[1]`http://fimi.cs.helsinki.fi/testdata.html`

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

[3]`http://www.ecn.purdue.edu/KDDCUP/`

[4]`http://www.cs.helsinki.fi/u/goethals/software/`

[5]University of Cornell Database group, Himalaya Data Mining Tools, `http://himalaya-tools.sourceforge.net/`

7.2 Results

Figures 6 through Figure 12 show the running time with varying minimum supports for the seven algorithms, namely LCMfreq, LCM, LCMmax, FP-growth, eclat, apriori, mafia-mfi on the nine datasets described in the previous subsection. In the following, we call all, maximal, closed frequent item set mining simply by all, maximal, closed.

Results on Synthetic Data

Figure 4 shows the running time with minimum support ranging from 0.15% to 0.025% on IBM-Artificial T10I4D100K datasets. From this plot, we see that most algorithms run within around a few 10 minutes and the behaviors are quite similar when minimum support increases. In Figure 4, All of LCMmax, LCM, and LCMfreq are twice faster than FP-growth on IBM T10I4D100K dataset. On the other hand, Mafia-mfi, Mafia-fci, and Mafia-fi are slower than every other algorithms. In Figure 5, Mafia-mfi is fastest for maximal, and LCMfreq is fastest for all, for minimum support less than 1% on IBM T10I4D100K dataset.

Results on KDD-CUP datasets

Figures 6 through Figure 8 show the running time with range of minimum supports from ranging from 0.1% to 0.01% on three real world datasets BMS-WebView-1, BMS-WebView-2, BMS-POS datasets. In the figure, we can observe that LCM algorithms outperform others in almost cases, especially for lower minimum support. In particular, LCM was best among seven algorithms in a wide range of minimum support from 0.1% to 0.01% on all datasets.

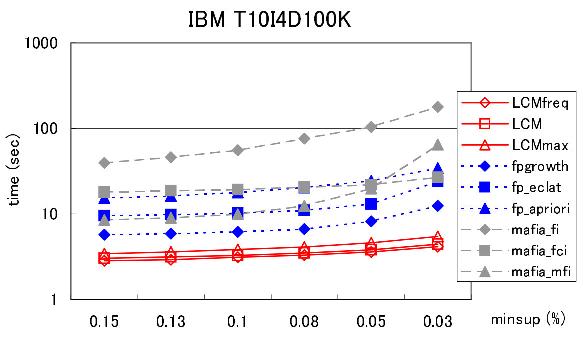

Figure 4: Running time of the algorithms on IBM-Artificial T10I40D100K

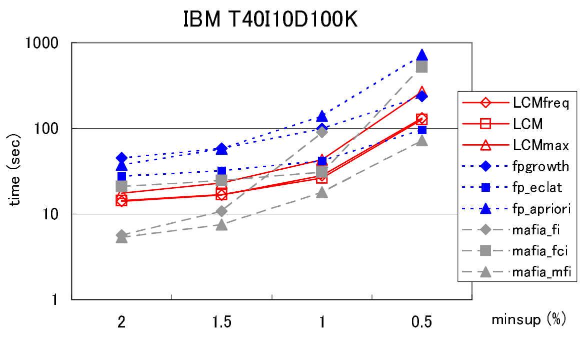

Figure 5: Running time of the algorithms on IBM-Artificial T40I10D100K

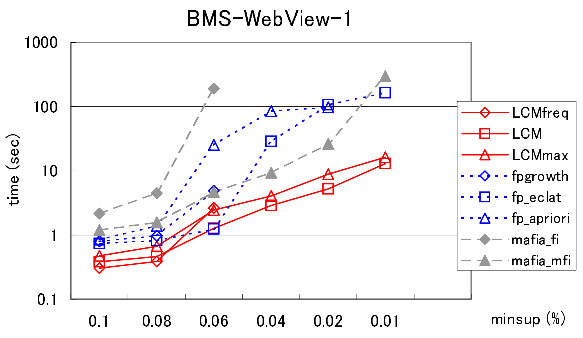

Figure 6: Running time of the algorithms on BMS-WebView-1

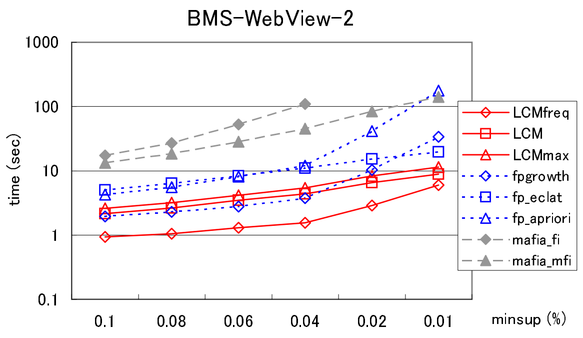

Figure 7: Running time of the algorithms on BMS-WebView-2

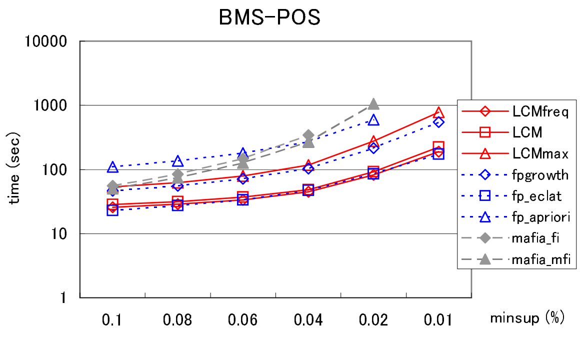

Figure 8: Running time of the algorithms on BMS-POS

For higher minimum support ranging from 0.1% to 0.06%, the performances of all algorithms are similar, and LCM families have slightly better performance. For lower minimum support ranging from 0.04% to 0.01%, Eclat and Apriori are much slower than every other algorithms. LCM outperforms others. Some frequent item set miners such as Mafia-fi, and Mafia-fci runs out of 1GB of main memory for these minimum supports on BMS-WebView-1, BMS-WebView-2, BMS-POS datasets. LCMfreq works quite well for higher minimum support, but takes more than 30 minutes for minimum support above 0.04% on BMS-Web-View-1. In these cases, the number of frequent item sets is quite large, over 100,000,000,000. Interestingly, Mafia-mfi's performance is stable in a wide range of minimum support from 0.1% to 0.01%.

In summary, LCM family algorithms significantly perform well on real world datasets BMS-WebView-1, BMS-WebView-2, BMS-POS datasets.

Results on UCI-ML repository and PUMSB datasets

Figures 9 through Figure 12 show the running time on middle sized data sets pumsb and pumsb*, kosarak and small sized datasets connect, chess, and mushroom. These datasets taken from machine learning domains are small but hard datasets for frequent pattern mining task since they have many frequent patterns even with high minimum supports, e.g., from 50% to 90%. These datasets are originally build for classification task and have slightly different characteristics than large business datasets such

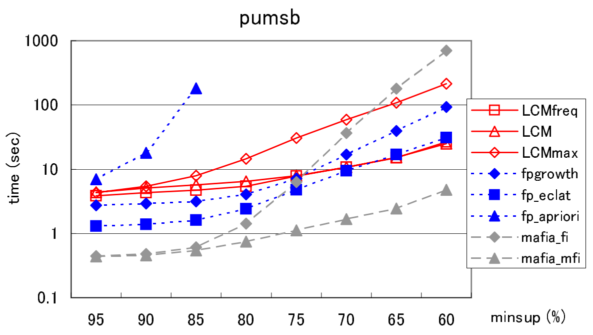

Figure 9: Running time of the algorithms on pumsb

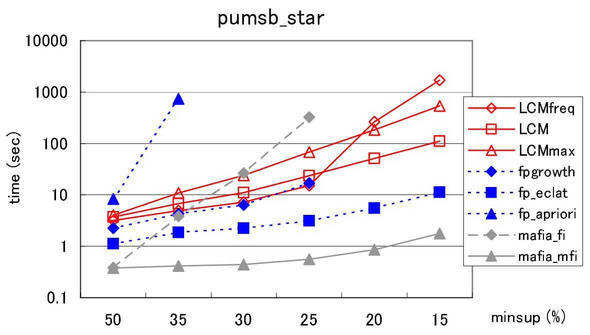

Figure 10: Running time of the algorithms on pumsb*

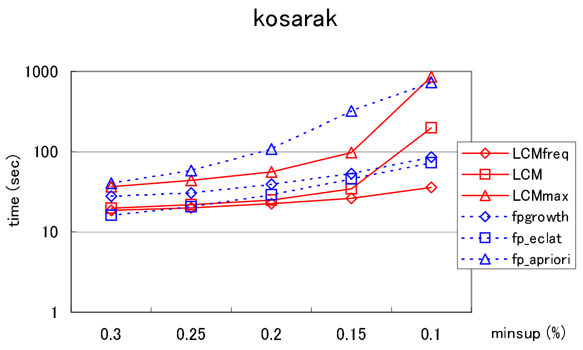

Figure 11: Running time of the algorithms on kosarak

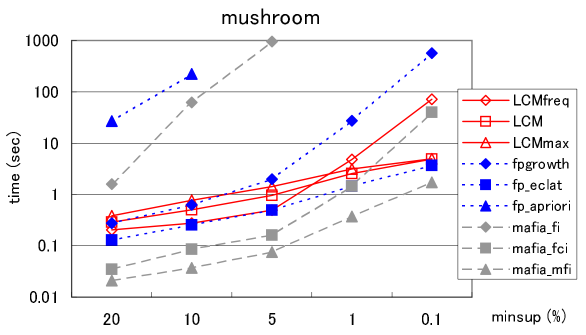

Figure 12: Running time of the algorithms on mushroom

as BMS-Web-View-1 or BMS-POS.

In these figures, we see that Mafia-mfi constantly outperforms every other maximal frequent item sets mining algorithms for wide range of minimum supports except on pumsb*. On the other hand, Apriori is much slower than other algorithm. On the mining of all frequent item sets, LCMfreq is faster than the others algorithms. On the mining of frequent closed item sets, there seems to be no consistent tendency on the performance results. However, LCM does not store the obtained solutions in the memory, while the other algorithms do. Thus, in the sense of memory-saving, LCM has an advantage.

8 Conclusion

In this paper, we present an efficient algorithm LCM for mining frequent closed item sets based on parent-child relationship defined on frequent closed item sets. This technique is taken from the algorithms for enumerating maximal bipartite cliques [14, 15] based on reverse search [3]. In theory, we demonstrate that LCM exactly enumerates the set of frequent closed item sets within polynomial time per closed item set in the total input size. In practice, we show by experiments that our algorithms run fast on several real world datasets such as BMS-WebView-1. We also showed variants LCMfreq and LCMmax of LCM for computing maximal and all frequent item sets. LCMfreq uses new schemes hybrid and hypercube decomposition, and the schemes work well for many problems.

Acknowledgement

We gratefully thank to Prof. Ken Satoh of National Institute of Informatics. This research had been supported by group research fund of National Institute of Informatics, JAPAN.

References

[1] R. Agrawal and R. Srikant, “Fast Algorithms for Mining Association Rules in Large Databases,” In *Proc. VLDB ’94*, pp. 487–499, 1994.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen and A. I. Verkamo, “Fast Discovery of Associa-

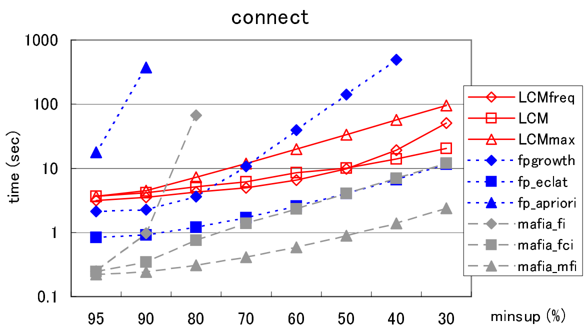

Figure 13: Running time of the algorithms on connect

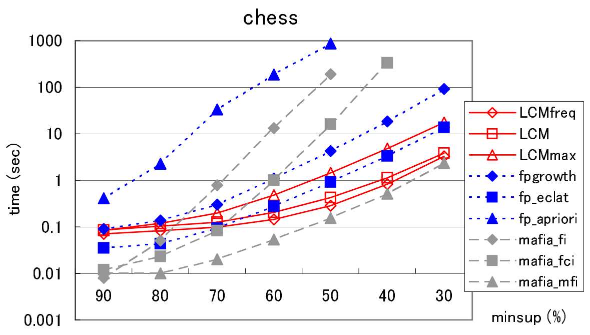

Figure 14: Running time of the algorithms on chess

tion Rules," In *Advances in Knowledge Discovery and Data Mining,* MIT Press, pp. 307–328, 1996.

[3] D. Avis and K. Fukuda, "Reverse Search for Enumeration," *Discrete Applied Mathematics,* Vol. 65, pp. 21–46, 1996.

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

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

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

[7] J. Han, J. Pei, Y. Yin, "Mining Frequent Patterns without Candidate Generation," In *Proc. SIGMOD'00*, pp. 1-12, 2000

[8] 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.

[9] H. Mannila, H. Toivonen, "Multiple Uses of Frequent Sets and Condensed Representations," In *Proc. KDD'96*, pp. 189–194, 1996.

[10] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, Discovering frequent closed itemsets for association rules, In *Proc. ICDT'99*, pp. 398-416, 1999.

[11] 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.

[12] B. Possas, N. Ziviani, W. Meira Jr., B. A. Ribeiro-Neto, "Set-based model: a new approach for information retrieval," In *Proc. SIGIR'02*, pp. 230-237, 2002.

[13] 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.

[14] Takeaki Uno, "A Practical Fast Algorithm for Enumerating Cliques in Huge Bipartite Graphs and Its Implementation," 89th Special Interest Group of Algorithms, Information Processing Society Japan, 2003,

[15] Takeaki Uno, "Fast Algorithms for Enumerating Cliques in Huge Graphs," Research Group of Computation, IEICE, Kyoto University, pp.55-62, 2003

[16] Takeaki Uno, "A New Approach for Speeding Up Enumeration Algorithms," In *Proc. ISAAC'98*, pp. 287–296, 1998.

[17] M. J. Zaki, "Scalable algorithms for association mining," *Knowledge and Data Engineering,* 12(2), pp. 372–390, 2000.

[18] M. J. Zaki, C. Hsiao, "CHARM: An Efficient Algorithm for Closed Itemset Mining," In *Proc. SDM'02*, SIAM, pp. 457-473, 2002.

[19] Z. Zheng, R. Kohavi and L. Mason, "Real World Performance of Association Rule Algorithms," In *Proc. SIGKDD-01*, pp. 401-406, 2001.