TCS-TR-A-07-30

TCS Technical Report

Fast Generation of Very Large-Scale Frequent Itemsets Using a Compact Graph-Based Representation

by

Shin-ichi Minato, Takeaki Uno, and Hiroki Arimura

Division of Computer Science

Report Series A

October 10, 2007

Hokkaido University
Graduate School of
Information Science and Technology

Email: minato@ist.hokudai.ac.jp

Phone: +81-011-706-7682

Fax: +81-011-706-7682

Fast Generation of Very Large-Scale Frequent Itemsets Using a Compact Graph-Based Representation

Shin-ichi Minato
Division of Computer Science
Hokkaido University
Sapporo 060-0814, Japan

Takeaki Uno
National Institute of Informatics
Tokyo 101–8430, Japan.

Hiroki Arimura
Division of Computer Science
Hokkaido University
Sapporo 060-0814, Japan

October 10, 2007

(Abstract) Frequent itemset mining is one of the fundamental techniques for data mining and knowledge discovery. In the last decade, a number of efficient algorithms for frequent itemset mining have been presented, but most of them focused on just enumerating the itemsets which satisfy the given conditions, and it was a different matter how to store and index the mining result for efficient data analysis.
In this paper, we propose a fast algorithm for generating very large-scale all/closed/maximal frequent itemsets using Zero-suppressed BDDs (ZBDDs), a compact graph-based data structure. Our method, "LCM over ZBDDs," is based on one of the most efficient state-of-the-art algorithms proposed before, and not only enumerating/listing the itemsets but also generating a compact output data structure on the main memory. The result can efficiently be post-processed by using algebraic ZBDD operations. The original LCM is known as an output linear time algorithm, but our new method requires a sub-linear time to the number of frequent patterns when the ZBDD-based data compression works well. Our method may greatly accelerate the data mining process and will lead a new style of on-memory processing for knowledge discovery problems.

1 Introduction

Discovering useful knowledge from large-scale databases has attracted a considerable attention during the last decade. Frequent itemset mining is one of the fundamental data mining problems. Since the pioneering paper by Agrawal *et al.* [1] various algorithms have been proposed to solve the frequent pattern mining problem (cf., e.g., [3, 5, 16]. Among those state-of-the-art algorithms, *LCM (Linear time Closed itemset Miner)*[15, 13, 14] by Uno et al. has a feature of the theoretical bound as output linear time. Their open source code[12] is known as one of the fastest implementation of frequent itemset mining program.

LCM and most of the other itemset mining algorithms focus on just enumerating or listing the itemsets which satisfy the given conditions, and it was a different matter how to store and index the result of itemsets for efficient data analysis. If we want to post-process the mining results by applying various conditions or restrictions, once we have to dump the frequent itemsets into storage. Even LCM is an output linear time algorithm, it may require impracticable time and space if the number of frequent itemsets becomes enormous. Usually we control the output size with the minimum support threshold in ad hoc setting, but we do not know if it may lose some important information.

For representing very large-scale frequent itemsets, one proposed a method of using *Zero-suppressed Binary Decision Diagrams (ZBDDs)*[7], an efficient graph-based data structure. ZBDD is a variant of *Binary Decision Diagram (BDD)*[2], which was originally developed in VLSI logic design area, but recently applied to data mining problems[9, 6, 8]. Last year, Minato et al. presented *ZBDD-growth*[11] algorithm to compute all/closed/maximum frequent itemsets based on ZBDD operations, and generate a compressed output data structure on the main memory. Unfortunately, the overhead of ZBDD-based frequency computation is not small in their algorithm, so the computational advantage is limited to only the examples where ZBDD-based data compression rate is extremely high. Otherwise, for example, when the number of frequent itemsets is not very large, ordinary LCM algorithm is much faster than ZBDD-growth.

In this paper, we propose a good combination of LCM algorithm and a ZBDD-based data structure. Our method, "LCM over ZBDDs," can generate very large-scale frequent itemsets on the main memory with a very small overhead of computation time compared with the original LCM algorithm. The mining result can be post-processed efficiently by using algebraic ZBDD operations. The original LCM is an output linear time algorithm, but our new method requires a sub-linear time to the number of frequent itemsets when the ZBDD-based data compression works well. Our method may greatly accelerate the data mining process and will lead a new style of on-memory processing for knowledge discovery problems.

2 Preliminaries

Let $\mathcal{E} = \{1, 2, \ldots, n\}$ be the set of *items*. A *transaction database* on $\mathcal{E}$ is a set $\mathcal{T} = \{T_1, T_2, \ldots, T_m\}$ such that each T_i is included in $\mathcal{E}$. Each T_i is called a *transaction* (or *tuple*). 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 $P \subseteq \mathcal{E}$ is called an *itemset* (or *pattern*). The maximum element of P is called the *tail* of P, and denoted by $tail(P)$. An itemset Q is a *tail extension* of P if and only if both $Q \setminus P = \{e\}$ and $e > tail(P)$ hold for an item e. An itemset $P \neq \emptyset$ is a tail extension of Q if and only if $Q = P \setminus tail(P)$, hence Q is unique, i.e., any non-empty itemset is a tail extension of a unique itemset.

For itemset P, a transaction including P is called an *occurrence* of P. The *denotation* of P, denoted by $Occ(P)$ set of the occurrences of P. $|Occ(P)|$ is called the *frequency* of P, and denoted by $frq(P)$. In particular, for an item e, $frq(\{e\})$ is called the frequency of e. 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*. We define the *closure* of itemset P in $\mathcal{T}$, denoted by $clo(P)$, by $\bigcap_{T \in Occ(P)} T$. An itemset P is called *closed* if $P = clo(P)$.

3 LCM and ZBDDs

In this section, we briefly explain LCM algorithm and ZBDD-based techniques for representing frequent itemsets.

3.1 LCM algorithm

LCM is a series of algorithms for enumerating frequent itemsets, developed by Uno et al. These algorithms have a feature of the theoretical bound as output linear time. The first LCM algorithm is presented in FIMI2003[15], and the second version of LCM demonstrated its remarkable efficiency in FIMI2004[13]. The original LCM was developed for enumerating closed itemsets, and then LCMfreq and LCMmax are presented for mining all frequent itemsets and maximal itemsets[1]. Now the three variants are integrated into one program. Those implementations are available at the developer's web page[12] as open source software.

In general, frequent itemset mining algorithms are classified into the two categories: *apriori-like* (or *level-by-level*) algorithms[1] and *backtracking* (or *depth-first*) algorithms[16, 5]. LCM algorithms belong to backtracking style.

Backtracking algorithm is based on recursive calls. The algorithm inputs a frequent itemset P, and generates new itemsets by adding one of the unused items to P. Then, for each itemset being frequent among them, it generates recursive calls with respect to it. To avoid duplications, an iteration of backtracking algorithms adds items with

[1]The complexity has been proved theoretically in generating all/closed itemsets, but still open (only experimental) for maximal one.

indices larger than the tail of P. We describe the framework of backtracking algorithms as follows.

```
ALGORITHM Backtracking ($P$: itemset)
  Output $P$
  For each $e \in \mathcal{E}$, $e > tail(P)$ do
    If $P \cup \{e\}$ is frequent then
      call Backtracking ($P \cup \{e\}$)
```

LCM algorithms are based on backtracking algorithms, and use acceleration techniques for the frequency counting, called *occurrence deliver* and *anytime database reduction.* Hence, LCM algorithms efficiently compute the frequency. Here we omit detailed techniques used in LCM, as they are described in [13, 14].

Although LCM can efficiently enumerate large-scale frequent itemsets, it is a different matter how to store and index the result of itemsets for efficient data analysis. Even LCM is an output linear time algorithm, it may require impracticable time and space if the number of frequent itemsets becomes enormous. Usually we control the output size with the minimum support threshold in ad hoc setting, but we do not know if it may lose some important information to be discovered.

3.2 ZBDDs

A *Binary Decision Diagram (BDD)* is a graph representation for a Boolean function. An Example is shown in Fig. 1 for $F(a, b, c) = a\bar{b}c \vee \bar{a}b\bar{c}$. Given a variable ordering (in our example a, b, c), one can use Bryant's algorithm[2] to construct the BDD for any given Boolean function. For many Boolean functions appearing in practice this algorithm is quite efficient and the resulting BDDs are much more efficient representations than binary decision trees.

BDDs were originally invented to represent Boolean functions. But we can also map a set of combinations into Boolean space of n variables, where n is the cardinality of $\mathcal{E}$ (see Fig. 2). So, one could also use BDDs to represent sets of combinations. However, one can even obtain a more efficient representation by using *Zero-suppressed* BDDs (ZBDDs)[7].

If there are many similar combinations then the subgraphs are shared resulting in a smaller representation. In addition, ZBDDs have a special type of node deletion rule. As shown in Fig. 3, All nodes whose 1-edge directly points to the 0-terminal node are deleted. Because of this, the nodes of items that do not appear in any sets of combinations are automatically deleted as shown in Fig.1. This ZBDD reduction rule is extremely effective if we handle a set of sparse combinations. If the average appearance ratio of each item is 1%, ZBDDs are possibly more compact than ordinary BDDs, up to 100 times.

ZBDD representation has another good property that each path from the root node to the 1-terminal node corresponds to each combination in the set. Namely, the number of such paths in the ZBDD equals to the number of combinations in the set. This beautiful property indicates that, even if there are no equivalent nodes to be shared, the ZBDD structure explicitly stores all items of each combination, as well as using an explicit linear linked list data structure. In other words, (the order of) ZBDD size never exceeds the explicit representation. If more nodes are shared, the ZBDD is more compact than linear list.

Table 1 summarizes the most of primitive operations of ZBDDs. In these operations, "$\emptyset$," "$\mathbf{1}$," and $P.top$ can be obtained in a constant time. $P.offset(v)$, $P.onset(v)$, and $P.change(v)$ operations require a constant time if v is the top variable of P, otherwise they consume linear time to the number of ZBDD nodes located higher position than v. The union, intersection, and difference operations can be performed in almost linear time to the size of ZBDDs.

3.3 ZBDD-growth algorithm

Using a ZBDD-based compact data structure, we can efficiently manipulate large-scale itemset databases on the main memory. Recently, Minato et al. have developed *ZBDD-growth* algorithm to generate all/closed/maximal frequent itemsets for given databases. The details of the algorithm is written in the article[11]. As well as LCM algorithms, ZBDD-growth is based on the backtracking

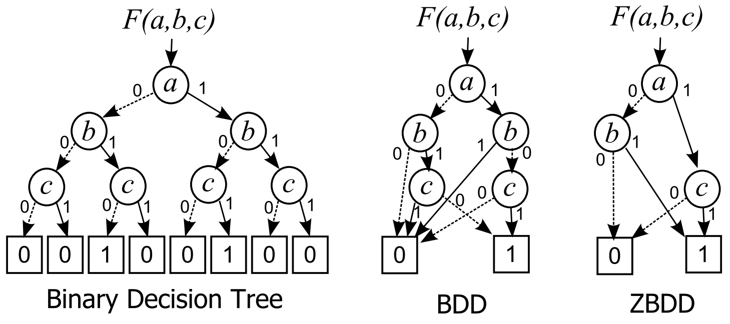

Figure 1: Binary Decision Tree, BDDs and ZBDDs

a	b	c	F	
0	0	0	0	
0	0	1	0	
0	1	0	1	$\rightarrow b$
0	1	1	0	
1	0	0	0	
1	0	1	1	$\rightarrow ac$
1	1	0	0	
1	1	1	0	

$\rightarrow S$

As a Boolean function:
$F = a\bar{b}c \vee \bar{a}b\bar{c}$

As a set of combinations:
$S = \{ac,\ b\}$

Figure 2: Correspondence of Boolean functions and sets of combinations.

algorithm using recursive calls. This algorithm has the two technical features as follows:

(i) using of ZBDDs for the internal data structure, and

(ii) using of ZBDDs for the output data structure.

In the first feature, the internal data structure means that the given transaction database is converted to ZBDD-based representation on the main memory. On each recursive step of backtracking, frequency counting for the conditional (or restricted) database is performed by ZBDD operations. This is similar manner as *FP-growth*[5] algorithm, which manipulates *FP-tree* in the backtracking algorithm.

Since ZBDDs are representation of sets of combinations, a simple ZBDD distinguishes only existence of each itemset in the database. In order to count the integer numbers of frequency, ZBDD-growth algorithm uses an m-digits of ZBDD vector $\{F_0, F_1, \ldots, F_{m-1}\}$ to represent integers up to $(2^m - 1)$, as shown in Fig. 4. The numbers are encoded into binary digital code, as F_0 represents a set of itemsets appearing odd times (LSB = 1), F_1 represents a set of itemsets whose appearance number's second lowest bit is 1, and similar way we define the set of each digit up to F_{m-1}. Notice that this ZBDD vector is used only for the internal data structure in ZBDD-growth algorithm. The output data is represented by a simple ZBDD because the result is just a set of frequent itemsets. (not keeping frequency of each itemset.)

ZBDD-growth algorithm manipulate ZBDDs for both the internal and output data structures, so the advantage of ZBDD-based data compression is fully employed. There are examples where billions of frequent itemsets can be represented by only thousands of ZBDD nodes. The mining result can be post-processed efficiently by using algebraic ZBDD operations.

However, ZBDD-growth has an overhead for frequency computing using ZBDD vectors. Arithmetic operations of ZBDD vectors are performed by a series of ZBDD operations on each binary digit, so it requires more steps than ordinary 32-

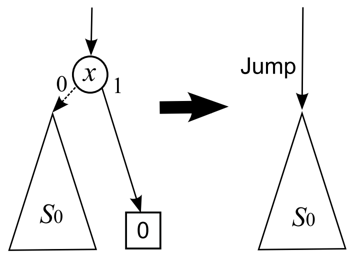

Figure 3: ZBDD reduction rule.

Table 1: Primitive ZBDD operations

"$\emptyset$"	Returns empty set. (0-termial node)
"**1**"	Returns the set of only null-combination. (1-terminal node)
$P.top$	Returns the item-ID at the root node of P.
$P.offset(v)$	Subset of combinations not including item v.
$P.onset(v)$	Gets $P - P.offset(v)$ and then deletes v from each combination.
$P.change(v)$	Inverts existence of v (add / delete) on each combination.
$P \cup Q$	Returns union set.
$P \cap Q$	Returns intersection set.
$P - Q$	Returns difference set. (in P but not in Q.)
$P.count$	Counts number of combinations.

or 64-bit arithmetic operations in the CPU. Unless the ZBDD-based data compression rate is very high, the overhead becomes obvious. There are the two typical cases where ZBDD is not very effective.

- The number of itemsets is enough small to be handled easily in anyway.
- The database is completely random and no similar itemsets are included.

In many practical cases, ZBDD-growth algorithm is not faster than previous algorithms. As shown in the experimental result in this paper, ZBDD-growth is 10 to 100 times slower than ordinary LCM when the output size is small. ZBDD-growth wins only when a huge number of frequent itemsets are generated.

4 LCM over ZBDDs

In this section, we discuss a good combination of LCM and ZBDDs. Fortunately, we can observe a number of common properties in LCM algorithms and ZBDD manipulation, as follows:

- Both are based on the backtracking (depth-first) algorithm.
- All the items used in the database have a fixed variable ordering.
- In the algorithm, we choose an item one by one according to the variable ordering, and then call the algorithm itself recursively.
- In the current implementation of LCM, the variable ordering is decided at the beginning of the algorithm, and the ordering is never changed until the end of execution.

These common properties indicate that LCM and ZBDDs may have a really good combination. Our algorithm, "LCM over ZBDDs," does not touch the core algorithm of LCM, and just generates a ZBDD for the solutions obtained by LCM. In this way, we aim to generate very large-scale frequent itemsets efficiently with a very small overhead of ZBDD manipulation. Here we describe the techniques in the new method.

tuple	frequency	F_2	F_1	F_0
abc	5 (101)	1	0	1
ab	3 (011)	0	1	1
bc	2 (010)	0	1	0
c	1 (001)	0	0	1

F_0 = {abc, ab, c}
F_1 = {ab, bc}, F_2 = {abc}

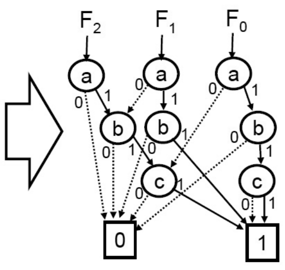

Figure 4: ZBDD vector for frequency counting.

```
LCM_Backtrack($P$: itemset)
{
   **Output** $P$
   **For** $e = n$ **to** $tail(P) + 1$ **step** $-1$ **do**
      **If** $P \cup \{e\}$ is frequent
         LCM_Backtrack($P \cup \{e\}$)
}
```

Figure 5: Basic structure of LCM algorithm.

```
ZBDD LCMovZBDD_Naive($P$: itemset)
{
   ZBDD $F \leftarrow P$
   **For** $e = n$ **to** $tail(P) + 1$ **step** $-1$ **do**
      **If** $P \cup \{e\}$ is frequent {
         $F' \leftarrow$ LCMovZBDD_Naive($P \cup \{e\}$)
         $F \leftarrow F \cup F'$
      }
   **Return** $F$
}
```

Figure 6: Naive modification for generating ZBDDs.

4.1 ZBDD construction in LCM procedure

We recall the basic structure of the original LCM algorithm in Fig. 5. Here we omit detailed techniques used in checking frequency of each itemset, but basically the algorithm explores all the candidate of itemsets in a backtracking (or depth-first) manner, and when a frequent itemset is found, it is appended one by one to the output file. On the other hand, "LCM over ZBDDs" constructs a ZBDD which is the union of all the itemsets found in the backtracking search, and finally returns a pointer to the root node of the ZBDD. A naive modification can be described as Fig. 6. However, this naive algorithm has a problem in its efficiency.

In the LCM procedure, a ZBDD grows by repeating union operations of the frequent itemsets found in the depth-first search. If we look at the sequence of itemsets generated by the algorithm, the consecutive itemsets are quite similar to each other in most cases, namely, only a few items near by the tail are different and the other top items are completely identical. The ZBDD union operations becomes as Fig. 7, only a few bottom levels are different and the almost all other part is the same. Since the procedure of ZBDD operation is executed recursively from the top node to the bottom node, the computation of a union operation requires $O(n)$ steps while only a few bottom items are meaningful. Namely, this algorithm will become n times slower. It is unacceptable loss of efficiency because n may be more than hundred in practical datasets.

To address this problem, we improved the algorithm as shown in Fig. 8. On each recursive step, we construct a ZBDD only for the lower items, and after returning from the subsidiary recursive call, we put the top item on the current result of ZBDD. In this way, we can avoid redundant traversals in the ZBDD union operation, as shown in Fig. 9. If we use the variable ordering of ZBDDs as same as the LCM's item ordering, each ZBDD operation requires only a constant time, and the total overhead of ZBDD generation can be bounded by a constant factor compared with the original LCM.

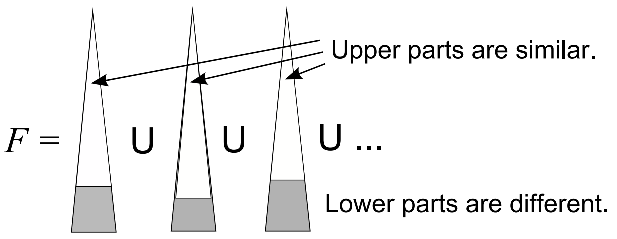

Figure 7: ZBDD union operations in naive LCM over ZBDDs.

```
ZBDD LCMovZBDD($P$: itemset)
{
  ZBDD $F \leftarrow$ "1"
  **For** $e = n$ **to** $tail(P) + 1$ **step** $-1$ **do**
    **If** $P \cup \{e\}$ is frequent {
      $F' \leftarrow$ LCMovZBDD($P \cup \{e\}$)
      $F \leftarrow F \cup F'.change(e)$
    }
  **Return** $F$
}
```

Figure 8: Improved version of "LCM over ZBDD."

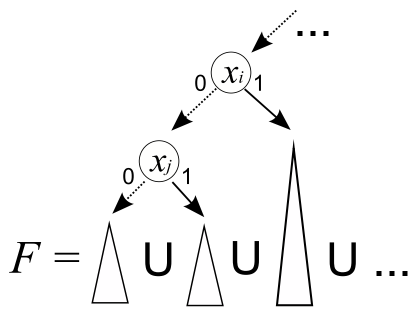

Figure 9: Efficient ZBDD construction in LCM over ZBDDs.

4.2 Employing hypercube decomposition

The original LCM finds a number of frequent itemsets at once for reducing the computation time by using the technique of *hypercube decomposition*[15] (or, also called *equisupport*). For a frequent itemset P, let $H(P)$ be the set of items e satisfying $e > tail(P)$ and $Occ(P) = Occ(P \cup \{e\})$. Then, for any $Q \subseteq H(P)$, $Occ(P) = Occ(P \cup Q)$ holds, and $P \cup Q$ is frequent. The original LCM avoids duplicated backtracking with respect to items included in $H(P)$, by passing $H(P)$ to the subsidiary recursive calls. The algorithm is shown in Fig. 10.

Current LCM implementations have the two output options, (i) printing out all the solutions to the output file, or (ii) just counting the total number of solutions. When counting the number of itemsets, we accumulate 2's power to the hypercube size for each solution, without generating all candidates derived from the hypercube. This technique greatly reduces the computation time since the LCM algorithm is dominated by the output size.

Also in LCM over ZBDDs, we can employ the hypercube decomposition technique. The algorithm is described in Fig. 11. A remarkable advantage of our method is that we can efficiently generate a ZBDD including all the solutions, within a similar computation time as the original LCM only counting the number of the solutions. The original LCM is known as an output linear time algorithm, but our method can generate all the solutions in a sub-linear time to the number of solutions if the hypercubes often appear.

4.3 Closed/maximal itemset mining

The original LCM can also generate closed/maximal itemsets. Our method does not touch the core algorithm of LCM, and just generates ZBDDs for the solutions obtained by LCM. Therefore, a ZBDD for closed/maximal itemsets can be generated as well as the original LCMs. The technique of hypercube decomposition should slightly be modified to generate closed/maximal one, but it is a similar technique used in the original LCMs.

```
LCM_Backtrack_H($P, S$: itemset)
{
  $S' \leftarrow S \cup H(P)$
  **Output** itemsets including $P$
        and included in $P \cup S'$
  **For** $e = n$ **to** $tail(P) + 1$ **step** $-1$ **do**
    **If** $e \notin S'$ and $P \cup \{e\}$ is frequent
      LCM_Backtrack_H($P \cup \{e\}, S'$)
}
```

Figure 10: Original LCM with hypercube decomposition.

```
ZBDD LCMovZBDD_H($P, S$: itemset)
{
  $S' \leftarrow S \cup H(P)$
  ZBDD $F \leftarrow$ "1"
  **For** $e = n$ **to** $tail(P) + 1$ **step** $-1$ **do**
    **If** $e \in S'$
      $F \leftarrow F \cup F.change(e)$
    **Else if** $P \cup \{e\}$ is frequent {
      $F' \leftarrow$ LCMovZBDD_H($P \cup \{e\}, S'$)
      $F \leftarrow F \cup F'.change(e)$
    }
  **Return** $F$
}
```

Figure 11: LCM over ZBDDs with hypercube decomposition.

5 Experimental Results

Based on the above ideas, we implemented LCM over ZBDDs by modifying the open software, LCM ver. 5[12]. We composed about 50 lines modifications or additions to the main file of the original LCM, and compiled it with our own ZBDD package, which consists of about 2,300 lines of C codes. We used a 2.4GHz Core2Duo E6600 PC, 2 GB of main memory, with SuSE Linux 10 and GNU C++ compiler. On this platform, we can manipulate up to 40,000,000 nodes of ZBDDs with up to 65,000 different items.

For evaluating the performance, we applied our method to the practical size of datasets chosen from FIMI2003 repository[4] with various minimum support thresholds. We compared our results with the original LCM[12] and ZBDD-growth[11]. In the datasets, "mushroom" is known as an example where ZBDD-growth is effective since the ZBDD-based data compression works well. "T10I4D100K" is known as an opposite case, an artificial database consists of randomly generated combinations. In this case, ZBDD-based data compression is quite ineffective. "BMS-WebView-1" has an intermediate property between the two.

Table 2 shows our experimental results. In this table, |ZBDD| means the number of ZBDD nodes representing all frequent itemsets. The column "LCM-count" shows the computation time of the original LCM only counting the number of solutions, and 'LCM-dump" means the time for listing all itemset data to the output file (using /dev/null). "LCMoverZBDD" and "ZBDD-growth" show the time for generating the result of ZBDD on the main memory, including the time for counting the ZBDD nodes.

From the experimental results, we can clearly see that LCM over ZBDDs is more efficient than ZBDD-growth in most cases. Larger advantage of our method can be observed when a smaller number of solutions are generated. ZBDD-growth shows comparable performances to our method only in "mushroom" with very low minimum support, but for all the other cases, our method overwhelms ZBDD-growth.

We can also observe that LCM over ZBDDs is more efficient than the original LCM-dump. The difference becomes significant when very large number of itemsets are generated. The original LCM-dump is known as an output linear time algorithm, but our LCM over ZBDDs requires a sublinear time to the number of itemsets. The computation time of our method is almost same as executing LCM-count. We must emphasize that LCM-count does not store the itemsets but only count the number of solutions. On the other hand, LCM over ZBDDs generates all the solutions and store them on the main memory as a compact ZBDD. This is an important point.

After executing LCM over ZBDDs, we can apply various algebraic operations to the ZBDD for filtering or analyzing the frequent itemsets[11]. Storing

Table 2: Comparison of LCM over ZBDDs with the previous methods.

Dataset name:	#Frequent	LCMoverZBDDs		LCM-count	LCM-dump	ZBDD-growth
min. support	itemsets	|ZBDD|	Time(s)	Time(s)	Time(s)	Time(s)
mushroom: 1,000	123,287	760	0.50	0.49	0.64	1.78
500	1,442,504	2,254	1.32	1.30	3.29	3.49
300	5,259,786	4,412	2.25	2.22	9.96	5.11
200	18,094,822	6,383	3.21	3.13	31.63	6.24
100	66,076,586	11,584	5.06	4.87	114.21	6.72
70	153,336,056	14,307	7.16	7.08	277.15	6.97
50	198,169,866	17,830	8.17	7.86	357.27	6.39
T10I4D100K: 100	27,533	8,482	0.85	0.85	0.86	209.82
50	53,386	16,872	0.97	0.92	0.98	242.31
20	129,876	58,413	1.13	1.08	1.20	290.78
10	411,366	173,422	1.55	1.36	1.64	332.22
5	1,923,260	628,491	2.86	2.08	3.54	370.54
3	6,169,854	1,576,184	5.20	3.15	8.14	386.72
2	19,561,715	3,270,977	9.68	5.09	22.66	384.60
BMS-WebView-1: 50	8,192	3,415	0.11	0.11	0.12	29.46
40	48,544	10,755	0.18	0.18	0.22	48.54
36	461,522	28,964	0.49	0.42	0.98	67.16
35	1,177,608	38,164	0.80	0.69	2.24	73.64
34	4,849,466	49,377	1.30	1.07	8.58	83.36
33	69,417,074	59,119	3.53	3.13	144.98	91.62
32	1,531,980,298	71,574	31.90	29.73	3,843.06	92.47
chess: 1,000	29,442,849	53,338	197.58	197.10	248.18	1,500.78
connect: 40,000	23,981,184	3,067	5.42	5.40	49.21	212.84
pumsb: 32,000	7,733,322	5,443	60.65	60.42	75.29	4,189.09
BMS-WebView-2: 5	26,946,004	353,091	4.84	3.62	51.28	118.01

the result as a ZBDD will be more useful than having a large dump file of all frequent itemsets.

Finally, we show the experimental results for generating closed itemsets in Table 3. We compred our results with the original LCM and ZBDD-growthC[10], a variation of ZBDD-growth to generate closed itemsets. Since the closed (or maximal) itemsets are a very small subset of all frequent itemsets, in this case, the performances of LCM-count and LCM-dump are not so different. Anyway, LCM over ZBDDs can efficiently generate the clesed itemsets with a very small overhead of ZBDD manipulation. As well as the ZBDD of all frequent itemsets, various post-processing is applicable to the ZBDD of closed itemsets. For example, we can easily obtain all "non-closed" itemsets by using ZBDD-based difference operation between all frequent itemsets and closed itemsets.

6 Conclusion

In this paper, we proposed "LCM over ZBDDs" algorithm for efficiently generating very large-scale all/closed/maximal frequent itemsets using ZBDDs. Our method is based on LCM, one of the most efficient state-of-the-art algorithms proposed before, and not only enumerating the itemsets but also generating a compact output data structure on the main memory. The result can efficiently be post-processed by using algebraic ZBDD operations.

The original LCM is known as an output linear time algorithm, but our new method requires a sub-linear time to the number of frequent patterns when the ZBDD-based data compression works well. Our experimental results indicate that the ZBDD-based method will greatly accelerate the data mining process and will lead a new style of on-memory processing for knowledge discovery problems.

Acknowledgment

This research was partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for on Priority Area "Information explosion" (Area

Table 3: Generating closed itemsets.

Dataset name: min. support	#Closed itemsets	LCMoverZBDDs |ZBDD|	LCMoverZBDDs Time(s)	LCM-count Time(s)	LCM-dump Time(s)	ZBDD-growthC Time(s)
mushroom: 1,000	3,427	1,059	0.58	0.55	0.55	1.86
500	9,864	2,803	1.28	1.24	1.24	3.62
100	45,944	9,884	3.06	2.93	2.40	6.54
50	68,468	12,412	3.48	3.35	3.50	8.71
T10I4D100K: 100	26,806	8,548	0.89	0.89	0.92	1,931.21
50	46,993	16,995	1.03	0.99	1.03	2,455.22
10	283,397	164,773	1.69	1.54	1.75	(>5,000)
2	2,270,195	1,476,698	6.62	4.76	6.47	(>5,000)
BMS-WebView-1: 50	7,811	3,477	0.12	0.12	0.13	32.09
40	29,489	11,096	0.24	0.22	0.26	58.44
35	76,260	29,553	0.84	0.79	0.88	102.87
32	110,800	46,667	1.94	1.86	1.98	138.22

#456), and Joint Research Project by National Institute of Informatics, Japan.

References

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In P. Buneman and S. Jajodia, editors, *Proc. of the 1993 ACM SIGMOD International Conference on Management of Data, Vol. 22(2) of SIGMOD Record*, pages 207–216, 1993.

[2] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[3] B. Goethals. Survey on frequent pattern mining, 2003. `http://www.cs.helsinki.fi/u/goethals/publications/survey.ps`.

[4] B. Goethals and M. J. Zaki. Frequent itemset mining dataset repository, 2003. Frequent Itemset Mining Implementations (FIMI'03), `http://fimi.cs.helsinki.fi/data/`.

[5] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: a frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87, 2004.

[6] E. Loekit and J. Bailey. fast mining of high dimensional expressive contrast patterns using zero-suppressed binary decision diagrams. In *Proc. The Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD2006)*, pages 307–316, 2006.

[7] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. of 30th ACM/IEEE Design Automation Conference*, pages 272–277, 1993.

[8] S. Minato. Symmetric item set mining based on zero-suppressed BDDs. In *the 9th International Conference on Discovery Science (DS-2006), (LNAI 4265, Springer)*, pages 321–326, 10 2006.

[9] S. Minato and H. Arimura. Efficient combinatorial item set analysis based on zero-suppressed BDDs. In *Proc. IEEE/IEICE/IPSJ International Workshop on Challenges in Web Information Retrieval and Integration (WIRI-2005)*, pages 3–10, 4 2005.

[10] S. Minato and H. Arimura. frequent closed item set mining based on zero-suppressed BDDs. *Trans. of the Japanese Society of Artificial Intelligence*, 22(2):165–172, 2007.

[11] S. Minato and H. Arimura. frequent pattern mining and knowledge indexing based on zero-suppressed BDDs. In *In Knowledge Discovery in Inductive Databases, 5th International Workshop, KDID 2006 Revised Selected and*

Invited Papers, LNCS 4747, pages 152–169, 9 2007.

[12] T. Uno and H. Arimura. Program codes of takeaki uno and hiroki arimura, 2007. `http://research.nii.ac.jp/~uno/codes.htm`.

[13] T. Uno, M. Kiyomi, and H. Arimura. LCM ver.2: efficient mining algorithms for frequent/closed/maximal itemsets. In *Proc. IEEE ICDM'04 Workshop FIMI'04 (International Conference on Data Mining, Frequent Itemset Mining Implementations)*, 2004.

[14] T. Uno, M. Kiyomi, and H. Arimura. LCM ver.3: collaboration of array, bitmap and prefix tree for frequent itemset mining. In *Proc. Open Source Data Mining Workshop on Frequent Pattern Mining Implementations 2005*, 2005.

[15] T. Uno, Y. Uchida, T. Asai, and H. Arimura. LCM: an efficient algorithm for enumerating frequent closed item sets. In *Proc. Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, 2003. `http://fimi.cs.helsinki.fi/src/`.

[16] M. J. Zaki. Scalable algorithms for association mining. *IEEE Trans. Knowl. Data Eng.*, 12(2):372–390, 2000.