TCS-TR-A-09-37

TCS Technical Report

Distinctive Frequent Itemset Mining from Time Segmented Databases Using ZDD-Based Symbolic Processing

by

Shin-ichi Minato and Takeaki Uno

Division of Computer Science

Report Series A

January 12, 2009

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

Distinctive Frequent Itemset Mining from Time Segmented Databases Using ZDD-Based Symbolic Processing

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

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

January 12, 2009

(Abstract) Frequent itemset mining is one of the fundamental techniques for data mining and knowledge discovery. Recently, Minato et al. proposed a fast algorithm "LCM over ZDDs" for generating very large-scale frequent itemsets using Zero-suppressed BDDs (ZDDs), a compact graph-based data structure. Their method is based on LCM algorithm, one of the most efficient state-of-the-art techniques for itemset mining, and directly generates compact output data structures on the main memory, to be efficiently post-processed by using ZDD-based algebraic operations.
In this paper, we propose a novel method of finding distinctive frequent itemsets from time segmented (e.g. daily, weekly, monthly) sequential transaction databases. We define "frequency pattern chart" using regular expressions for specifying distinctive frequency patterns in time segmented databases. Our method efficiently extracts all itemsets satisfying a given frequency pattern chart using LCM over ZDDs algorithm and ZDD-based symbolic processing of finite automata. Experimental results show that our method is applicable to very large-scale problems, for example, we can find a small number of distinctive itemsets from a huge number (more than 10^{44}) of frequent itemsets in a few seconds. Time segmented databases often appear in many real-life problems, so our new method will have a significant impact to various practical applications.

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 problems for data mining and knowledge discovery. Since the pioneering work by Agrawal *et al.*[1], various algorithms have been proposed to solve the frequent itemset mining problem (cf., e.g., [3, 12]). Among those state-of-the-art algorithms, *Linear time Closed itemset Miner (LCM)* [11] by Uno et al. has a feature of the theoretical bound as output linear time. Their open source code [10] is known as one of the fastest implementations of a frequent itemset mining program.

LCM and most of the other itemset mining algorithms focus on only enumerating or listing the itemsets that satisfy the given conditions, and how to store and index the result of itemsets for a more efficient data analysis was a different matter. If we want to post-process the mining results by setting various conditions or restrictions, we have to dump the frequent itemsets into storage at least once. Even though LCM is an output linear time algorithm, it may require impracticable time and space if the number of frequent itemsets gets too large.

For representing very large-scale frequent itemsets, S. Minato proposed a method using *Zero-suppressed Binary Decision Diagrams (ZDDs, or ZBDDs)* [7], an efficient graph-based data structure. ZDD is a variant of a *Binary Decision Dia-*

gram (BDD) [2], which was originally developed in the VLSI logic design area, but later it has been applied to data mining problems [8, 6]. Recently, Minato et al. [9] proposed a nice combination of an LCM algorithm and a ZDD-based data structure. Their method, called "LCM over ZDDs," can generate a huge number of frequent itemsets on the main memory with a very small overhead of computational time when compared with the original LCM algorithm. The mining result can be efficiently post-processed by using algebraic ZDD operations.

LCM over ZDDs algorithm is developed for combinatorial (non-sequential) itemset mining, but in this paper, we propose a novel idea of applying this algorithm to a sequential data mining problem. Our method finds distinctive itemsets which frequently appear in "time segmented" sequential transaction databases, where the time segmented databases means a list of transaction databases, each of which is indexed by a sequential segment number (as $D_1, D_2, \ldots, D_N$). Daily (or weekly, monthly) databases are the typical cases of this model. Here we do not define the sequential order of the transactions in a same time segment.

Now, we are interested in the distinctive itemsets which have a specific sequential behavior of occurrnces in the time segmeneted databases, for example, an itemset is always infrequent in D_1 to D_k (for some k), and after that it is always frequent from D_{k+1} to end. It will be a very powerful tool if we can extract all the itemsets satisfying such a given sequential frequency pattern. To deal with this problem, we define "frequency pattern chart" using regular expressions to describe various frequency patterns. Our method efficiently extracts all itemsets satisfying a given frequency pattern chart using LCM over ZDDs algorithm and ZDD-based symbolic data processing of finite automata. Experimental results show that our method is applicable to very large-scale problems, for example, we can find a small number of distinctive itemsets from a huge number (more than 10^{44}) of frequent itemsets in a few seconds. Time segmented databases often appear in many real-life problems, so our new method will have a significant impact to various practical applications.

In the rest of this paper, we start with preliminaries for itemset mining and ZDD data structure in Section 2. We then review the LCM over ZDDs algorithm in Section 3. Section 4 describes our new method of finding distinctive frequent itemsets from time segmented databases. Experimental results are shown in Section 5, followed by summary of this paper.

2 Preliminaries

2.1 Transaction Databases and Itemset Mining

Let $\mathcal{E} = \{1, 2, \ldots, n\}$ be the set of *items*. A *transaction database* on $\mathcal{E}$ is a multiset $D = \{T_1, T_2, \ldots, T_m\}$ where each T_i is included in $\mathcal{E}$. Each T_i is called a *transaction* (or *tuple*). We denote the sum of sizes of all transactions in D, with $||D||$ that is, the size of database D. A set $P \subseteq \mathcal{E}$ is called an *itemset*. The maximum element of P is called the *tail* of P, and is 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 $Q = P \setminus tail(P)$, and therefore, 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 an *occurrence* of P. The *denotation* of P, which is denoted by $Occ(P)$, is the set of the occurrences of P. $|Occ(P)|$ is the *frequency* of P, and is denoted by $frq(P)$. In particular, for an item e, $frq(\{e\})$ is the frequency of e. For a given constant θ, called a *minimum support*, itemset P is *frequent* if $frq(P) \geq \theta$. If a frequent itemset P is not included in any other frequent itemset, P is *maximal*. We define the *closure* of itemset P in D, denoted by $clo(P)$, with $\bigcap_{T \in Occ(P)} T$. An itemset P is *closed* if $P = clo(P)$.

The problem of frequent itemset mining is to enumerate all (or maximal/closed) frequent itemsets for a given database D and a parameter θ. In other words, it is to generate a family of itemsets for a given setting. Frequent itemset mining is one of the fundamental problems for data mining and knowledge discovery. Since the pioneering work by

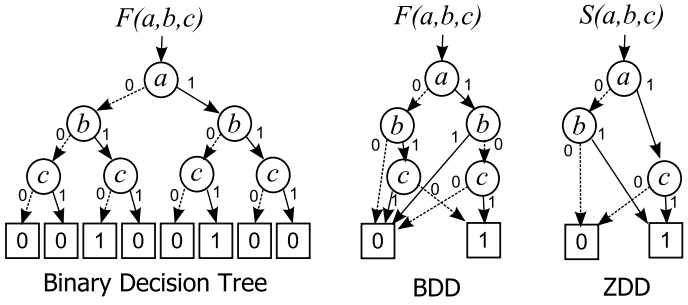

Figure 1: Binary Decision Tree, BDD and ZDD

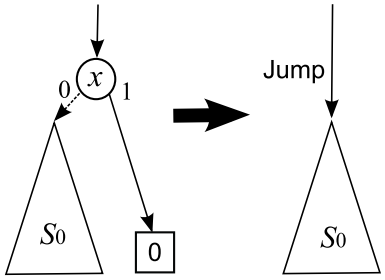

Figure 3: ZDD reduction rule.

a	b	c	F	
0	0	0	0	$\rightarrow S$
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	

As a Boolean function:
$F(a,b,c) = a\overline{b}c \vee \overline{a}b\overline{c}$

As a family of itemsets:
$S(a,b,c) = \{ac,\ b\}$

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

Agrawal *et al.*[1], various algorithms have been proposed to solve the frequent itemset mining problem (cf., e.g., [3, 12].

2.2 Zero-suppressed Binary Decision Diagrams (ZDDs)

Next, we briefly review the data structures of decision diagrams. A *Binary Decision Diagram (BDD)* is a graph representation for a Boolean function. An Example is shown in Fig. 1 for $F(a,b,c) = a\overline{b}c \vee \overline{a}b\overline{c}$. 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 family of itemsets 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 families of itemsets. However, one can even obtain a more efficient representation by using *Zero-suppressed* BDDs (ZDDs, or ZBDDs)[7].

If there are many similar itemsets then the subgraphs are shared resulting in a smaller representation. In addition, ZDDs 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 itemset are automatically deleted as shown in Fig.1. This ZDD reduction rule is extremely effective if we handle a family of sparse itemsets. If the average appearance ratio of each item is 1%, ZDDs are possibly more compact than ordinary BDDs, up to 100 times.

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

Figure 1 summarizes the primitive operations of the ZDDs. In these operations, "$\emptyset$," "**1**," and S.top can be obtained in a constant time. S.offset(k), S.onset(k), and S.change(k) operations require a constant time if item k is at the root node of S, otherwise they consume linear time for the number of ZDD nodes located at a higher position than item k. The union, intersection, and difference operations can be performed in almost linear time to

Table 1: Primitive ZDD operations

"$\emptyset$"	Returns empty family. (0-termial node)
"**1**"	Returns singleton family of null-itemset. (1-terminal node)
S.top	Returns item-ID at root node of S.
S.offset(k)	Sub-family of itemsets not including item k.
S.onset(k)	Gets $S - S$.offset(k) and then deletes item k from each itemset.
S.change(k)	Inverts existence of item k (add / delete) on each itemset.
$S_1 \cup S_2$	Returns union of the two families.
$S_1 \cap S_2$	Returns intersection of the two families.
$S_1 - S_2$	Returns difference of the two families. (in S_1 but not in S_2.)
S.count	Counts cardinality of S.

the size of the ZDDs. S.count is also linear to the ZDD size, not depends on the cardinality.

Recently, Prof. D. E. Knuth presented a draft version of the upcoming fascicle of his famous book series [5]. This new fascicle (total 140 pages, including 236 exercises) is entirely devoted for BDDs, and ZDD is also discussed minutely using 30 pages including 70 exercises. Prof. Knuth re-arranged a set of primitive ZDD operations and named it "Family Algebra." He has developed his own BDD/ZDD package, which is available in his home page.

3 LCM over ZDDs for Large-Scale Itemset Mining

We briefly review LCM over ZDDs algorithm to efficiently generate a huge number of frequent itemsets on the main memory.

3.1 LCM Algorithm

As described above, various algorithms have been proposed to solve the frequent itemset mining problem. Among those state-of-the-art algorithms, *LCM (Linear time Closed itemset Miner)*[11] by Uno et al. has a feature of the theoretical bound as output linear time[1]. Their open source code[10] is known as one of the fastest implementation of frequent itemset mining program.

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

```
LCM_Backtrack(P: itemset)
{
  Output P
  For e = n to tail(P) + 1 step −1 do
    If P ∪ {e} is frequent
      LCM_Backtrack(P ∪ {e})
}
```

Figure 4: Basic structure of LCM algorithm.

```
ZDD LCMovZDD(P: itemset)
{
  ZDD F ← P
  For e = n to tail(P) + 1 step −1 do
    If P ∪ {e} is frequent {
      F′ ← LCMovZDD(P ∪ {e})
      F ← F ∪ F′
    }
  Return F
}
```

Figure 5: Basic structure of LCM over ZDDs.

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 a sequential file storage. Even LCM is an output linear time algorithm, it may require impracticable time and space if the number of frequent itemsets becomes enormous.

Table 2: Comparison of LCM over ZDDs with original LCM.

Database name	#Item	#Transaction	$\|\|D\|\|$ (size of D)	Min. support	#Frequent itemsets	LCM over ZDDs |ZDD|	LCM over ZDDs Time(s)	LCM-count Time(s)	LCM-dump Time(s)
mushroom	119	8,124	186,852	1,000	123,287	760	0.50	0.49	0.64
				300	5,259,786	4,412	2.25	2.22	9.96
				100	66,076,586	11,584	5.06	4.87	114.21
				50	198,169,866	17,830	8.17	7.86	357.27
BMS-WebView-1	497	59,602	149,639	50	8,192	3,415	0.11	0.11	0.12
				40	48,544	10,755	0.18	0.18	0.22
				36	461,522	28,964	0.49	0.42	0.98
				34	4,849,466	49,377	1.30	1.07	8.58
				32	1,531,980,298	71,574	31.90	29.73	3,843.06
BMS-WebView-2	3,340	77,512	358,278	5	26,946,004	353,091	4.84	3.62	51.28
T10I4D100K	870	100,000	1,010,228	2	19,561,715	3,270,977	9.68	5.09	22.66
chess	75	3,196	118,252	1,000	29,442,849	53,338	197.58	197.10	248.18
connect	129	67,557	2,904,951	40,000	23,981,184	3,067	5.42	5.40	49.21
pumsb	2,113	49,046	3,629,404	32,000	7,733,322	5,443	60.65	60.42	75.29

(2.4GHz Core2Duo E6600 PC, 2 GB memory, SuSE Linux 10, GNU C++)

3.2 LCM over ZDDs

Recently, Minato et al. [9] proposed a fast algorithm for generating huge number of frequent itemsets using ZDDs. Their method, "LCM over ZDDs" is based on LCM algorithm and generating a compact output data using ZDDs on the main memory. The result can efficiently be post-processed by using algebraic ZDD operations.

LCM over ZDDs does not touch the core algorithm of LCM, and just generates a ZDD for the solutions obtained by LCM. Figure 4 shows the basic structure of the original LCM algorithm. Here we omit detailed techniques used in the original LCM for 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 ZDDs constructs a ZDD which is the union of all the itemsets found in the backtracking search, and finally returns a pointer to the root node of the ZDD. A basic modification can be described as Fig. 5.

The performance of LCM over ZDDs algorithm is shown in [9]. In the Table 2, the benchmark datasets are chosen from FIMI2003 repository [4]. |ZDD| represents the number of ZDD nodes representing all the frequent itemsets. The column "LCM-count" shows the computation time of the original LCM when counting only the number of itemsets, and 'LCM-dump" represents the time for listing all the itemsets into the output file (using /dev/null). "LCM over ZDD" show the time for generating the results of the ZDD on the main memory, including the time for counting the ZDD nodes.

We can observe that LCM over ZDDs is more efficient than the original LCM-dump. The difference becomes significant when very large numbers of itemsets are generated. The original LCM-dump is known as an output linear time algorithm, but LCM over ZDDs requires a sub-linear time for the number of itemsets. The computational time of LCM over ZDDs is almost the same as executing an LCM-count. We must emphasize that LCM-count does not store the itemsets, but only counts the number of solutions. On the other hand, LCM over ZDDs generates all the solutions and stores them on the main memory as a compact ZDD. This is an important difference.

After executing LCM over ZDDs, we can apply various algebraic ZDD operations to filter or analyze the frequent itemsets [8]. In the following sections, we present a novel method of finding distinctive frequent itemsets from time segmented

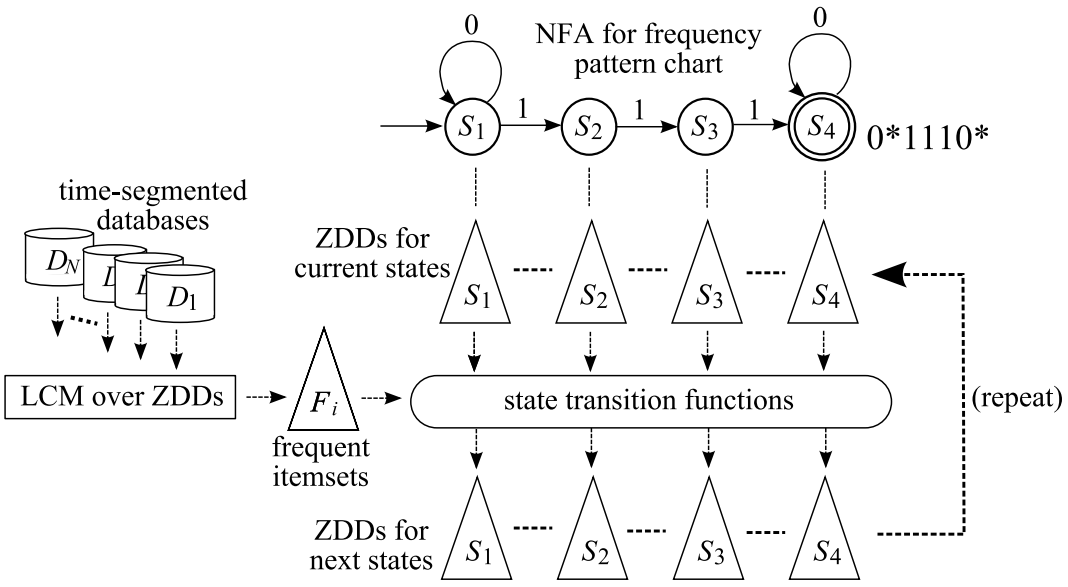

Figure 6: ZDD-based symbolic processing for finite automata.

databases.

4 Distinctive Frequent Itemset Mining from Time Segmented Databases

LCM over ZDDs algorithm is developed for combinatorial (non-sequential) itemset mining, but in this paper, we propose a novel idea of applying the algorithm to a sequential data mining problem. Our method finds distinctive itemsets which frequently appear in "time segmented" sequential transaction databases. This method will be useful for many types of real-life databases, such as, daily (or weekly, monthly, etc.) market sales, traffic accidents, web click streams, influenza virus types, etc.

First we define the time segmented sequential databases. This is a list of (the same type of) transaction databases D_i, each of which is indexed by a sequential segment number i ($i = 1, 2, \ldots, N$). Here we do not define the sequential order of the transactions in a same time segment.

4.1 Itemsets with Distinctive Sequential Frequency Patterns

In this data model, we generate a sequence of the families of frequent itemsets F_i from respective databases D_i with a given common minimum support ratio α. We then want to extract a family of distinctive frequent itemsets $\mathcal{F}$ having a sequential frequency pattern, for example:

- Each itemset in $\mathcal{F}$ is always included in F_i ($i > k$) but not in F_i ($i \leq k$), for some k ($1 \leq k < N$). In other words, the itemset suddenly become frequent on some day and then continuously frequent. We do not know k, when it becomes frequent.
- Each itemset in $\mathcal{F}$ is included only in F_k, F_{k+1}, F_{k+2}, for some k ($1 \leq k \leq N-2$). In other words, the itemset is frequent only in the consecutive three days, and never been frequent in other days. We do not know k, when it starts being frequent.
- Each itemset in $\mathcal{F}$ is included in F_i when ($i < k$) or ($i \geq l$), for some k, l ($1 < k < l < N$). In other words, the itemset is frequent on the first and the last days, however it is infrequent only in one consecutive period. We do not mind when it starts and ends being infrequent.

4.2 Frequency Pattern Charts Using Regular Expressions

We propose a way to represent "frequency pattern chart" by a string of '0' and '1'. For example, those three example can be represented by the string "0...01...1", "0...01110...0" and 1...10...01...1", where '0' means infrequent and '1' means frequent. More precisely, we can use regular expressions to represent the frequency pattern charts as follows:

- 00*11* (suddenly becomes frequent and continues.)
- 0*1110* (frequent only for three consecutive segments.)
- 11*00*11* (frequent, infrequent and frequent again.)

Using a regular expression, we can easily and accurately represent the sequence of frequency patterns which we are interested in. So, our goal is to extract a family of all itemsets satisfying the given frequency pattern chart represented by a regular expression.

4.3 ZDD-Based Symbolic Data Processing for Frequency Pattern Charts

The above problem is efficiently solved by ZDDs representing for families of itemsets. The overall process for "0*1110*" is illustrated in Fig. 6. First we construct a Nondeterministic Finite Automaton (NFA) to accept the same strings as the given regular expression. In this automaton, each state has at most two edges labeled '0' or '1'. When an itemset is staying at a certain state, and if the itemset is included in next F_i (i.e. frequent), then it choose '1' edge, otherwise choose '0' edge. If no corresponding edge exists, the itemset disappears. It is a well-known process to construct the NFA by a systematic way with a computation time linearly to the length of the regular expression. Next, we determine the state transition function for each state of the NFA.

In the case of "0*1110*", the state transition functions can be written as follows.

$$\begin{cases} \mathrm{Next}(\overline{S_1}) \leftarrow \overline{S_1} \cup F_i \\ \mathrm{Next}(S_2) \leftarrow F_i - \overline{S_1} \\ \mathrm{Next}(S_3) \leftarrow S_2 \cap F_i \\ \mathrm{Next}(S_4) \leftarrow (S_4 - F_i) \cup (S_3 \cap F_i) \end{cases}$$

In this formulas, S_j represents a family of itemsets staying at the state. Then, the itemsets included in F_i (i.e. $S_j \cap F_i$) travarse the 1-edge, and the itemsets not in F_i (i.e. $S_j - F_i$) traverse the 0-edge. In this way, the itemsets in the next states can be calculated as set operations.

Here we use the complement set $\overline{S_1}$ instead of S_1, because S_1 corresponds to "0*", that means itemsets which are always infrequent for all F_i's. However, we cannot directly get infrequent itemsets from F_i's, so we manipulate $\overline{S_1}$. As the regular language is closed under the complement operation, we can see that it has no significant impact to the computability.

We then simulate the NFA's action symbolically by using algebraic ZDD operations. In the simulation process, we prepare one ZDD for each state S_j of the NFA ($0 \leq j < M$, where M is the number of NFA states), and all S_j's are initialized to the empty family. Then the family of frequent itemsets F_1 is computed from D_1 by using LCM over ZDDs, and the F_1 is given as the first input to the NFA. The next states of S_j's can be computed by the state transition functions with the current S_j's and F_1 by using algebraic ZDD operations. This computation process is repeated sequentially until all F_i's $(i = 1, 2, \ldots N)$ are generated and given to the NFA. Finally, the ZDDs of the accepting states (S_4) contains the solutions of itemsets $\mathcal{F}$ whose frequency patterns satisfy the given frequency pattern chart. We remark that this method is based on a similar idea as the "symbolic simulation" used in VLSI formal verification method.

5 Experimental Results

To evaluate our method, we conducted experiments for the benchmark datasets. For example, "BMS-WebView-1" is known as a set of click streams for

Table 3: Number of itemsets and ZDD size for each state of NFA in symbolic processing.

i	F_i		$\overline{S_1}$		S_2		S_3		$S_4(=\mathcal{F})$	
(seg.)	#Itemset	|ZDD|	#Itemset	|ZDD|	#Itemset	|ZDD|	#Itemset	|ZDD|	#Itemset	|ZDD|
0	–	–	0	0	0	0	0	0	0	0
1	345,095	1,483	345,095	1,483	345,095	1,483	0	0	0	0
2	131,908	447	476,592	1,660	131,417	316	491	314	0	0
3	340	228	476,557	1,688	45	50	23	31	233	172
4	701	374	476,774	1,792	217	190	8	12	39	52
5	1,393	562	477,528	2,005	754	368	59	70	18	29
6	2,230	797	478,867	2,444	1,339	611	109	96	37	53
7	1,668	666	479,695	2,741	828	479	199	150	33	55
8	$(1.49 \cdot 10^{20})$	12,147	$(1.49 \cdot 10^{20})$	15,082	$(1.49 \cdot 10^{20})$	15,862	500	322	186	136
9	328	260	$(1.49 \cdot 10^{20})$	15,096	10	16	29	43	174	150
	...	...	...	...	...	...	...	...	...	...
55	1,598	737	$(7.14 \cdot 10^{44})$	26,499	361	248	109	82	878	313
56	425	336	$(7.14 \cdot 10^{44})$	26,516	19	25	3	4	877	320
57	443	322	$(7.14 \cdot 10^{44})$	26,560	17	26	0	0	872	318
58	455	342	$(7.14 \cdot 10^{44})$	26,607	57	63	5	8	866	312
59	1,610	723	$(7.14 \cdot 10^{44})$	26,817	676	287	17	18	852	302

Table 4: Experimental results for performance evaluation.

Dataset	α	Freq. pat.	#Itemset		Time
(#Segment)	(%)	chart	F_i's	$\mathcal{F}$	(sec)
BMS-WebView-1	0.5	0*1110*	$7.21 \cdot 10^{16}$	37	0.40
(59 segments)	0.4	0*1110*	$7.14 \cdot 10^{44}$	852	4.51
	0.3	0*1110*	$1.18 \cdot 10^{46}$	$3.57 \cdot 10^{44}$	42.00
	0.5	11*00*11*	$7.21 \cdot 10^{16}$	7	0.40
	0.4	11*00*11*	$7.14 \cdot 10^{44}$	6	4.41
	0.3	11*00*11*	$1.18 \cdot 10^{46}$	19	42.90
BMS-WebView-2	0.4	0*1110*	666,654	300	1.75
(77 segments)	0.3	0*1110*	9,236,264	1,493	2.69
	0.2	0*1110*	$1.44 \cdot 10^{17}$	38,895	7.04

(2.4GHz Core2Duo PC, 2 GB mem., SuSE 10, GNU C++)

an online shopping site. Each transaction shows a set of visiting web pages by one customer's consecutive action. This dataset consists of 59,602 transactions without any timing information, but we assume that they are sorted by time, and we partitioned them with 1,000 transactions per segment, i.e. we have 59 segments in total. (the last fragment is omitted.) In this way, we artificially generated time segmented databases.

We then apply our method to extract distinctive itemsets $\mathcal{F}$. Table.3 shows the result of the number of itemsets and ZDD sizes during the symbolic simulation for "BMS-WebView-1" with $\alpha = 0.4\%$ and the frequency pattern chart "0*1110*". We observed that 852 itemsets are finally extracted from 7136238463529799405291431334516279847981840 96 ($\approx 7.14 \cdot 10^{44}$) of frequent itemsets. The total computation time is less than five seconds, including the execution of LCM over ZDDs. To check the correctness, we confirmed that the itemset $\{18631, 18643\} \in \mathcal{F}$ is frequent only in D_{50}, D_{51} and D_{52}. Similarly, $\{46293, 46285, 46281\} \in \mathcal{F}$ is frequent only in D_6, D_7 and D_8. This means that we have discovered some local event only seen in these time segments.

Our method can be applied flexibly for various settings. Table 4 shows the performances for different datasets, parameters and frequency pattern charts.

We remark that our method repeats the same procedure for each time segment, so the computation time is basically linear to the number of time segments N, but it depends on ZDD sizes for representing F_i's and S_j's. We can apply this method for large N, if we appropriately control the problm size by the minimum support ratio α.

As related work on itemset data compression, the techniques of closed/maximal itemset mining are known. However, the distinctive itemsets in $\mathcal{F}$ may not closed/maximal for each F_i, so we cannot compute $\mathcal{F}$ correctly if we use the closed/maximal sets of F_i's. ZDD-based data structure has a great advantage that it can compress any family of itemsets and can apply symbolic operations for all itemsets without decompression. Our ZDD-based method will make a breakthrough to large-scale sequential data mining.

6 Summary

we presented a novel method of finding distinctive frequent itemsets from time segmented databases. We proposed "frequency pattern chart" using regular expressions for specifying distinctive frequency patterns in time segmented databases. Our method efficiently extracts all itemsets satisfying a given frequency pattern chart using LCM over ZDDs algorithm and ZDD-based symbolic processing of finite automata.

BDD-based symbolic processing techniques for sequential systems have been developed in the area of VLSI formal verification. Our method is the first practical result of applying such symbolic method to a sequential data mining problem. Time segmented databases often appear in many real-life problems, so our new method will have a significant impact to various practical applications.

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] D. E. Knuth. A draft of section 7.1.4: Binary decision diagrams. In *The Art of Computer Programming*, volume 4, pre-fascicle 1B. Addison-Wesley, 2008. http://www-cs-faculty.stanford.edu/ knuth/fasc1b.ps.gz.

[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 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.

[9] S. Minato, T. Uno, and H. Arimura. LCM over ZBDDs: Fast generation of very large-scale frequent itemsets using a compact graph-based representation. In *Proc. of 12-th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2008),*

(LNAI 5012, Springer), pages 234–246, 5 2008.

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

[11] 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/.

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