TCS-TR-A-**06-12**

TCS Technical Report

ZBDD-growth: An Efficient Method for Frequent Pattern Mining and Knowledge Indexing

by

Shin-ichi Minato and Hiroki Arimura

Division of Computer Science

Report Series A

April 10, 2006

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

ZBDD-growth: An Efficient Method for Frequent Pattern Mining and Knowledge Indexing

Shin-ichi Minato
Division of Computer Science, Hokkaido University
North 14, West 9, Sapporo, 060-0814 Japan

Hiroki Arimura
Division of Computer Science, Hokkaido University
North 14, West 9, Sapporo, 060-0814 Japan

April 10, 2006

(Abstract) Frequent pattern mining is one of the fundamental techniques for knowledge discovery and data mining. In the last decade, a number of efficient algorithms for frequent pattern mining have been presented, but most of them focused on just enumerating the patterns which satisfy the given conditions, and it was a different matter how to store and index the result of patterns for efficient data analysis. In this paper, we propose a fast algorithm of extracting all/maximal frequent patterns from transaction databases and simultaneously indexing the result of huge patterns using Zero-suppressed BDDs (ZBDDs). Our method, ZBDD-growth, is fast as competitive to the existing state-of-the-art algorithms, and not only enumerating/listing the patterns but also indexing the output data compactly on the memory. After mining, the result of patterns can efficiently be analyzed by using algebraic operations. The data structures of BDDs have already been used in VLSI logic design systems successively, but our method will be the first practical work of applying the BDD-based techniques for data mining area.

1 Introduction

Frequent pattern mining is one of the fundamental techniques for knowledge discovery and data mining. Since the introduction by Agrawal et al.[1], the frequent pattern mining and association rule analysis have been received much attentions from many researchers, and a number of papers have been published about the new algorithms or improvements for solving such mining problems[7, 9, 19]. However, most of such pattern mining algorithms focused on just enumerating or listing the patterns which satisfy the given conditions and it was a different matter how to store and index the result of patterns for efficient data analysis.

In this paper, we propose a fast algorithm of extracting all/maximal frequent patterns from transaction databases, and simultaneously indexing the result of huge patterns on the computer memory using Zero-suppressed BDDs. Our method does not only enumerate/list the patterns but also indexes the output data compactly on the memory. After mining, the result of patterns can efficiently be analyzed by using algebraic operations.

The key of our method is to use BDD-based data structure for representing sets of patterns. BDDs[4] are graph-based representation of Boolean functions, now widely used in VLSI logic design and verification area. For the data mining applications, it is important to use Zero-suppressed BDDs (ZBDDs)[12], a special type of BDDs, which are suitable for handling large-scale sets of combinations. Using ZBDDs, we can implicitly enumerate combinatorial item set data and efficiently compute set operations over the ZBDDs. The preliminary idea of using ZBDDs is presented in our last workshop paper[15], and after that we developed a fast pattern mining algorithm based on this data structure. Our work will be the first practical result of applying the BDD-based technique for data mining area.

For a related work, *FP-growth*[9] is received a great deal of attention because it supports fast manipulation of large-scale item set data using com-

pact tree structure on the main memory. Our method is a similar approach to handle sets of combinations on the main memory, but will be more efficient in the following points:

- ZBDDs are a kind of DAGs for representing item sets, while FP-growth uses a tree representation for the same objects. In general, DAGs can be more compact than trees.
- ZBDD-growth uses ZBDDs not only as internal data structure but also as output data structure. It provides an efficient knowledge index for consequent analysis.

ZBDD-growth algorithm is based on a recursive depth-first search of the database represented by ZBDDs. We show two versions of algorithms, generating all frequent patterns and generating maximal ones. Our experimental result shows that ZBDD-growth is fast as competitive to the existing state-of-the-art algorithms, such as FP-growth. Especially for the cases where the ZBDD nodes are well shared, exponential speed up are observed comparing to the existing algorithms based on explicit table/tree representation.

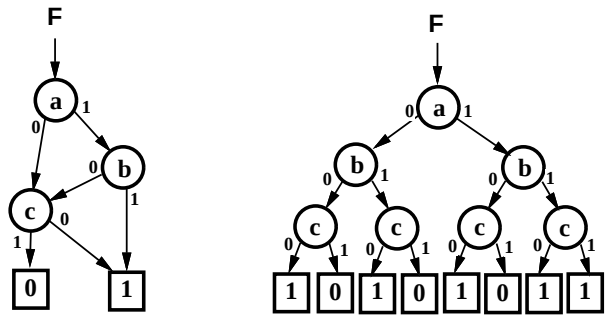
Recently, the data mining methods are often discussed in the context of Inductive Databases[3, 11], the integrated processes of knowledge discovery. In this paper, we place the ZBDD-based method as a basis of integrated discovery processes to efficiently execute various operations finding interest patterns and analyzing information involved in large-scale combinatorial item set databases.

2 BDDs and Zero-suppressed BDDs

Here we briefly describe the basic techniques of BDDs and Zero-suppressed BDDs for representing sets of combinations efficiently.

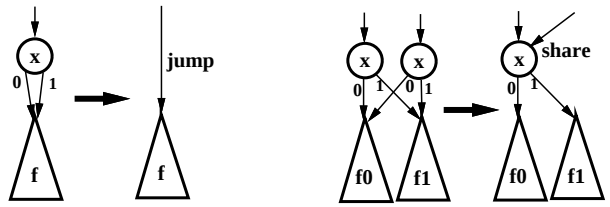
2.1 BDDs

BDD (Binary Decision Diagram) is a directed graph representation of the Boolean function, as illustrated in Fig. 1(a). It is derived by reducing



(a) BDD. (b) Binary tree.

Figure 1: BDD and binary tree: $F = (a \wedge b) \vee \overline{c}$.



(a) Node deletion. (b) Node sharing.

Figure 2: Reduction rules of ordinary BDDs

a binary tree graph representing recursive *Shannon's expansion*, indicated in Fig. 1(b). The following reduction rules yield a *Reduced Ordered BDD (ROBDD)*, which can efficiently represent the Boolean function. (see [4] for details.)

- Delete all redundant nodes whose two edges point to the same node. (Fig. 2(a))
- Share all equivalent sub-graphs. (Fig. 2(b))

ROBDDs provide canonical forms for Boolean functions when the variable order is fixed. Most researches on BDDs are based on the above reduction rules. In the following sections, ROBDDs will be referred to as BDDs (or ordinary BDDs) for the sake of simplification.

As shown in Fig. 3, a set of multiple BDDs can be shared each other under the same fixed variable ordering. In this way, we can handle a number of Boolean functions simultaneously in a monolithic memory space.

Using BDDs, we can uniquely and compactly represent many practical Boolean functions includ-

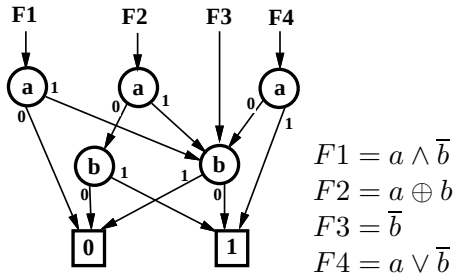


Figure 3: Shared multiple BDDs.

a	b	c	F	
0	0	0	0	
1	0	0	0	
0	1	0	0	
1	1	0	1	→ ab
0	0	1	1	→ c
1	0	1	1	→ ac
0	1	1	0	
1	1	1	0	

As a Boolean function:
F(a,b,c) = (a b ~c) V (~b c)

As a set of combinations:
F(a,b,c) = {ab, ac, c}

Figure 4: Set of combinations and Boolean function.

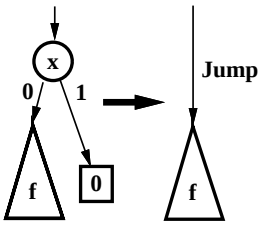


Figure 5: ZBDD reduction rule.

ing AND, OR, parity, and arithmetic adder functions. Using Bryant's algorithm[4], we can efficiently construct a BDD for the result of a binary logic operation (i.e. AND, OR, XOR), for given a pair of operand BDDs. This algorithm is based on hash table techniques, and the computation time is almost linear to the data size unless the data overflows the main memory. (see [13] for details.)

Based on these techniques, a number of BDD packages have been developed in 1990's and widely used for large-scale Boolean function manipulation, especially popular in VLSI CAD area.

2.2 Sets of Combinations and ZBDDs

BDDs are originally developed for handling Boolean function data, however, they can also be used for implicit representation of sets of combinations. Here we call "sets of combinations" for a set of elements each of which is a combination out of n items. This data model often appears in real-life problems, such as combinations of switching devices(ON/OFF), fault combinations, and sets of paths in the networks.

A combination of n items can be represented by an n-bit binary vector, $(x_1 x_2 \ldots x_n)$, where each bit, $x_k \in \{1, 0\}$, expresses whether or not the item is included in the combination. A set of combinations can be represented by a list of the combination vectors. In other words, a set of combinations is a subset of the power set of n items.

A set of combinations can be mapped into Boolean space by using n-input variables for each bit of the combination vector. If we choose any one combination vector, a Boolean function determines whether the combination is included in the set of combinations. Such Boolean functions are called *characteristic functions*. For example, Fig. 4 shows a truth-table representing a Boolean function $(ab\overline{c}) \vee (\overline{b}c)$, but also represents a set of combination $\{ab, ac, c\}$. Using BDDs for characteristic functions, we can implicitly and compactly represent sets of combinations. The logic operations AND/OR for Boolean functions correspond to the set operations intersection/union for sets of combinations. By using BDDs for characteristic functions, we can manipulate sets of combinations efficiently. They can be generated and manipulated within a time roughly proportional to the BDD size. When we handle many combinations including similar patterns (sub-combinations), BDDs are greatly reduced by node sharing effect, and sometimes an exponential reduction benefit can be obtained.

Zero-suppressed BDD (ZBDD)[12, 14] is a special type of BDDs for efficient manipulation of sets of combinations. ZBDDs are based on the following special reduction rules.

- Delete all nodes whose 1-edge directly points

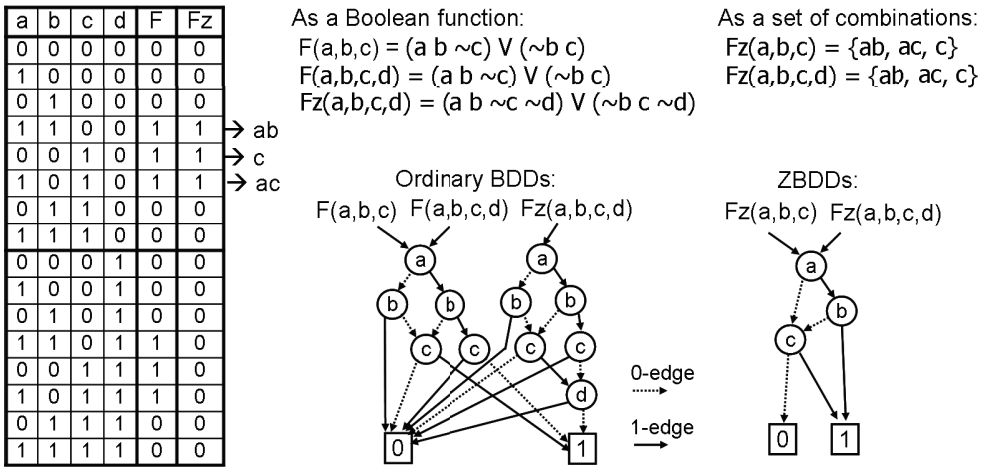


Figure 6: Effect of ZBDD reduction rule.

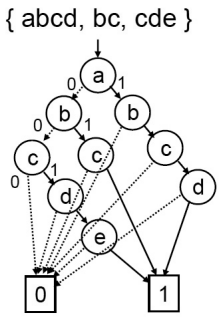


Figure 7: Explicit representation by ZBDD.

to the 0-terminal node, and jump through to the 0-edge's destination, as shown in Fig. 5.

- Share equivalent nodes as well as ordinary BDDs.

Notice that we do not delete the nodes whose two edges point to the same node, which used to be deleted by the original rule. The zero-suppressed deletion rule is asymmetric for the two edges, as we do not delete the nodes whose 0-edge points to a terminal node. It is proved that ZBDDs are also gives canonical forms as well as ordinary BDDs under a fixed variable ordering.

Here we summarize the features of ZBDDs.

- In ZBDDs, the nodes of irrelevant items (never chosen in any combination) are automatically deleted by ZBDD reduction rule. In ordinary BDDs, irrelevant nodes still remain and they may spoil the reduction benefit of sharing nodes.

  An example is shown in Fig. 6. In this case, the item d is irrelevant, but ordinary BDD for characteristic function $Fz(a,b,c)$ and $Fz(a,b,c,d)$ become different forms. On the other hand, ZBDDs for $Fz(a,b,c)$ and $Fz(a,b,c,d)$ become identical forms and completely shared.

- ZBDDs are especially effective for representing sparse combinations. For instance, sets of combinations selecting 10 out of 1000 items can be represented by ZBDDs up to 100 times more compact than ordinary BDDs.

- 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. In ordinary BDDs, this property does not always hold.

- When no equivalent nodes exist in a ZBDD, that is the worst case, the ZBDD structure explicitly stores all items in all combinations, as well as using an explicit linear linked list data structure. An example is shown in Fig. 7. Namely, (the order of) ZBDD size never exceeds the explicit representation. If more

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)	Selects the subset of combinations each of which does not include item v.
P.onset(v)	Selects the subset of combinations including item v, and then delete 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.

nodes are shared, the ZBDD is more compact than linear list. Ordinary BDDs have larger overhead to represent sparser combinations while ZBDDs have no such overhead.

Table 1 shows the most of primitive operations of ZBDDs. In these operations, $\emptyset$, **1**, *P.top* are executed in a constant time, and the others are almost linear to the size of graph. We can describe various processing on sets of combinations by composing of these primitive operations.

2.3 ZBDD-based Database Analysis

In this paper, we discuss the method of manipulating large-scale transaction databases using ZBDDs. Here we consider binary item set databases, each record of which holds a combination of items chosen from a given item list. Such a combination is called a *tuple* (or a *transaction*).

For analyzing those large-scale transaction databases, frequent pattern mining[2] and *maximum frequent pattern mining*[5] are especially important and they have been discussed actively in the last decade. Since the introduction by Agrawal et al.[1], a number of papers have been published about the new algorithms or improvements for solving such mining problems[7, 9, 19]. Recently, graph-based methods, such as FP-growth[9], are received a great deal of attention, since they can quickly manipulate large-scale tuple data by constructing compact graph structure on the main memory.

ZBDD-based method is a similar approach to handle sets of combinations on the main memory, but will be more efficient because ZBDDs are a kind of DAGs for representing item sets, while FP-growth uses a tree representation for the same objects. In general, DAGs can be more compact than trees.

Another important point is that our method uses ZBDDs not only as internal data structure but also as output data structure. The most of existing state-of-the-art pattern mining algorithms focused on just enumerating or listing the patterns which satisfy the given conditions, and it was a different matter how to store and index the result of patterns for efficient data analysis. In this paper, we present a fast algorithm of pattern mining and simultaneously indexing the result of huge patterns compactly on the main memory for consequent analysis. The results can be analyzed flexibly by using algebraic operations implemented on ZBDDs.

In addition, we show here why we use ZBDDs instead of ordinary BDDs in this application. Table 2 lists the basic statistics of typical benchmark data[7] often used for data mining/analysis problems. $\#I$ shows the number of items used in the data, $\#T$ is the number of tuples included in the data, $avg|T|$ is the average number of items per tuple, and $avg|T|/\#I$ is the average appearance ratio of each item. From this table, we can observe that the item's appearance ratio is very small in many cases. This is reasonable as considering real-life problems, for example, the number of items in a basket purchased by one customer is usually much less than all the items displayed in a shop. This observation means that we often handle very sparse combinations in many practical data mining/analysis problems, and in such cases, the

Table 2: Statistics of typical benchmark data.

Data name	$\#I$	$\#T$	$total\|T\|$	$avg\|T\|$	$avg\|T\|/\#I$
T10I4D100K	870	100,000	1,010,228	10.1	1.16%
T40I10D100K	942	100,000	3,960,507	39.6	4.20%
chess	75	3,196	118,252	37.0	49.30%
connect	129	67,557	2,904,951	43.0	33.33%
mushroom	119	8,124	186,852	23.0	19.32%
pumsb	2,113	49,046	3,629,404	74.0	3.50%
pumsb_star	2,088	49,046	2,475,947	50.5	2.42%
BMS-POS	1,657	515,597	3,367,020	6.5	0.39%
BMS-WebView-1	497	59,602	149,639	2.5	0.51%
BMS-WebView-2	3,340	77,512	358,278	4.6	0.14%
accidents	468	340,183	11,500,870	33.8	7.22%

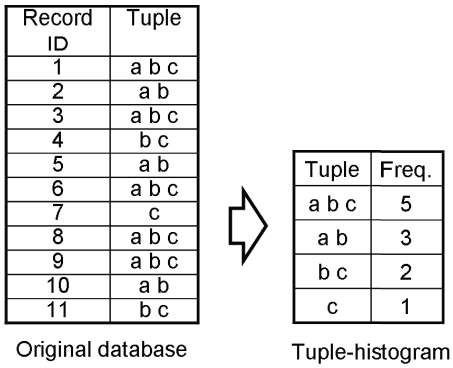


Figure 8: Database example and tuple-histogram.

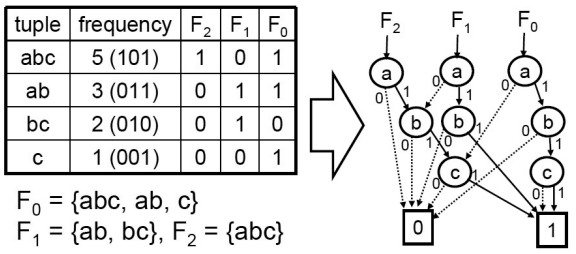


Figure 9: ZBDD vector for tuple-histogram.

ZBDD reduction rule is extremely effective. If the average appearance ratio of each item is 1%, ZBDDs may be more compact than ordinary BDDs up to 100 times. In the literature, there is a first report by Jiang et al.[10] applying BDDs to data mining problems, but the result seems not excellent due to the overhead of ordinary BDDs. We must use ZBDDs in stead of ordinary BDDs for success in many practical data mining/analysis problems.

3 ZBDD-growth: ZBDD-based pattern mining algorithm

In this section, we describe our new algorithm, ZBDD-growth, which extract all frequent patterns from a given transaction database using ZBDDs.

3.1 Tuple-Histograms and ZBDD vectors

A *Tuple-histogram* is the table for counting the number of appearance of each tuple in the given database. An example of tuple-histogram is shown in Fig. 8. This is just a compressed table of the database to combine the same tuples appearing more than once into one line with the frequency.

Our pattern mining algorithm manipulates ZBDD-based tuple-histogram representation as the internal data structure. Here we describe how to represent tuple-histograms using ZBDDs. Since ZBDDs are representation of sets of combinations, a simple ZBDD distinguishes only existence of each tuple in the database. In order to represent the numbers of tuple's appearances, we decompose the number into 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. 9. Namely, we encode the appearance numbers into binary digital code, as F_0 represents a set of tuples appearing odd times (LSB

= 1), F_1 represents a set of tuples whose appearance number's second lowest bit is 1, and similar way we define the set of each digit up to F_{m-1}.

In the example of Fig. 9, The tuple frequencies are decomposed as: $F_0 = \{abc, ab, c\}$, $F_1 = \{ab, bc\}$, $F_2 = \{abc\}$, and then each digit can be represented by a simple ZBDD. The three ZBDDs are shared their sub-graphs each other.

Now we explain the procedure for constructing a ZBDD-based tuple-histogram from original database. We read a tuple data one by one from the database, and accumulate the single tuple data to the histogram. More concretely, we generate a ZBDD of T for a single tuple picked up from the database, and accumulate it to the ZBDD vector. The ZBDD of T can be obtained by starting from "**1**" (a null-combination), and applying "Change" operations several times to join the items in the tuple. Next, we compare T and F_0, and if they have no common parts, we just add T to F_0. If F_0 already contains T, we eliminate T from F_0 and carry up T to F_1. This ripple carry procedure continues until T and F_k have no common part. After finishing accumulations for all data records, the tuple-histogram is completed.

Using the notation F.add(T) for addition of a tuple T to the ZBDD vector F, we describe the procedure of generating tuple-histogram H for given database D.

$H = \mathbf{0}$
forall $T \in D$ **do**
  $H = H$.add(T)
return H

When we construct a ZBDD vector of tuple-histogram, the number of ZBDD nodes in each digit is bounded by total appearance of items in all tuples. If there are many partially similar tuples in the database, the sub-graphs of ZBDDs are shared very well, and compact representation is obtained. The bit-width of ZBDD vector is bounded by $\log S_{max}$, where S_{max} is the appearance of most frequent items.

Once we have generated a ZBDD vector for the tuple-histogram, various operations can be executed efficiently. Here are the instances of operations used in our pattern mining algorithm.

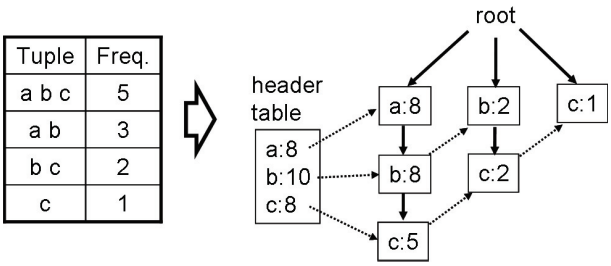


Figure 10: Example of FP-tree.

- H.factor0(v): Extracts sub-histogram of tuples without item v.
- H.factor1(v): Extracts sub-histogram of tuples including item v and then delete v from the tuple combinations. (also considered as the quotient of H/v)
- $v \cdot H$: Attaches an item v on each tuple combinations in the histogram F.
- $H_1 + H_2$: Generates a new tuple-histogram with sum of the frequencies of corresponding tuples.
- H.tuplecount: The number of tuples appearing at least once.

These operations can be composed as a sequence of ZBDD operations. The result is also compactly represented by a ZBDD vector. The computation time is bounded by roughly linear to total ZBDD sizes.

3.2 ZBDD vectors and FP-trees

FP-growth[9], one of the state-of-the-art algorithm, constructs "FP-tree" for a given transaction database, and then searches frequent patterns using this data structure. An example of FP-tree is shown in Fig. 10. We can see that FP-tree is a trie of tuples with their frequencies. In other words, **FP-growth is based on the tree representation of tuple-histograms.** Namely, ZBDD-growth is based on logically same internal data structure as FP-growth. This is the reason why we call this algorithm ZBDD-growth. However, ZBDD-based method will be more efficient

```
ZBDDgrowth(H, α)
{
  if(H has only one item v)
    if(v appears more than α ) return v ;
    else return "0" ;
  F ← Cache(H) ;
  if(F exists) return F ;
  v ← H.top ; /* Top item in H */
  H1 ← H.factor1(v) ;
  H0 ← H.factor0(v) ;
  F1 ←ZBDDgrowth(H1, α) ;
  F0 ←ZBDDgrowth(H0 + H1, α) ;
  F ← (v · F1) ∪ F0 ;
  Cache(H) ← F ;
  return F ;
}
```

Figure 11: ZBDD-growth algorithm.

```
ZBDDgrowthMax(H, α)
{
  if(H has only one item v)
    if(v appears more than α ) return v ;
    else return "0" ;
  F ← Cache(H) ;
  if(F exists) return F ;
  v ← H.top ; /* Top item in H */
  H1 ← H.factor1(v) ;
  H0 ← H.factor0(v) ;
  F1 ←ZBDDgrowthMax(H1, α) ;
  F0 ←ZBDDgrowthMax(H0 + H1, α) ;
  F ← (v · F1) ∪ (F0 − F0.permit(F1)) ;
  Cache(H) ← F ;
  return F ;
}
```

Figure 12: ZBDD-growth-max algorithm.

because ZBDDs can share the equivalent subgraphs and computation time is bounded by the ZBDD size. The benefit of ZBDDs is especially remarkable when a huge number of patterns are produced.

3.3 Frequent Pattern Mining Algorithm

Our algorithm, ZBDD-growth, is based on a recursive depth-first search over the ZBDD-based tuple-histogram representation. The basic algorithm is shown in Fig. 11.

In this algorithm, we choose an item v used in the tuple-histogram H, and compute the two sub-histograms H_1 and H_0. (Namely, $H = (v \cdot H_1) \cup H_0$.) As v is the top item in the ZBDD vector, H_1 and H_0 can be obtained just by referring the 1-edge and 0-edge of the highest ZBDD-node, so the computation time is constant for each digit of ZBDD.

The algorithm consists of the two recursive calls, one of which computes the subset of patterns including v, and the other computes the patterns excluding v. The two subsets of patterns can be obtained as a pair of pointers to ZBDDs, and then the final result of ZBDD is computed. This procedure may require an exponential number of recursive calls, however, we prepare a hash-based cache to store the result of each recursive call. Each entry in the cache is formed as pair (H, F), where H is the pointer to the ZBDD vector for a given tuple-histogram, and F is the pointer to the result of ZBDD. On each recursive call, we check the cache to see whether the same histogram H has already appeared, and if so, we can avoid duplicate processing and return the pointer to F directly. By using this technique, the computation time becomes almost linear to the total ZBDD sizes.

In our implementation, we use some simple techniques to prune the search space. For example, if H_1 and H_0 are equivalent, we may skip to compute F_0. For another case, we can stop the recursive calls if total frequencies in H is no more than α. There are some other elaborate pruning techniques, but they needs additional computation cost for checking the conditions, so sometimes effective but not always.

3.4 Extension for Maximal Pattern Mining

We can extend the ZBDD-growth algorithm to extract only the maximal frequent patterns[5], each of which is not included in any other frequent patterns. The algorithm is shown in Fig. 12.

The difference from the original algorithm is only one line, written in the frame box. In this part, we check each pattern in F_0, and delete it if the pattern is included in one of patterns of F_1. In this way, we can generate only maximal frequent patterns. This is basically the same approach as used in MAFIA[5].

```
P.permit(Q)
{
  if(P ="0" or Q ="0") return "0" ;
  if(P = Q) return F ;
  if(P ="1") return "1" ;
  if(Q ="1")
    if(P include "1" ) return "1" ;
    else return "0" ;
  R ← Cache(P,Q) ;
  if(R exists) return R ;
  v ←TopItem(P,Q) ; /* Top item in P,Q */
  (P0,P1) ←factors of P by v ;
  (Q0,Q1) ←factors of Q by v ;
  R ← (v · P1.permit(Q1)) ∪ (P0.permit(Q0 ∪ Q1))
;
  Cache(P,Q) ← R ;
  return R ;
}
```

Figure 13: Permit operation.

The process of deleting non-maximal patterns is basically a very time consuming task, however, we found that one of the ZBDD-based operation, called *permit* operation by Okuno et al.[17], can be used for solving this problem[1]. P.permit(Q) returns a set of combinations in P each of which is a subset of some combinations in Q. For example, when $P = \{ab, abc, bcd\}$ and $Q = \{abc, bc\}$, then P.permit(Q) returns $\{ab, abc\}$. The permit operation is efficiently implemented as a recursive procedure of ZBDD manipulation, as shown in Fig. 13. The computation time of permit operation is almost linear to the ZBDD size.

4 Experimental Results

Here we show the experimental results to evaluate our new method. We used a Pentium-4 PC, 800MHz, 1.5GB of main memory, with SuSE Linux 9. We can deal with up to 20,000,000 nodes of ZBDDs in this machine.

4.1 Experiment for Mathematical Example

First, we present the experiment for a set of artificial examples where ZBDD-growth is extremely effective. The database, named "one-pair-missing," has the following form for a given integer $n > 0$.

	$a_2b_2a_3b_3\cdots a_{n-1}b_{n-1}a_nb_n$		
a_1b_1	$a_3b_3\cdots a_{n-1}b_{n-1}a_nb_n$		
$a_1b_1a_2b_2$	$\cdots a_{n-1}b_{n-1}a_nb_n$		
$\vdots$	$\ddots$		$\vdots$
$a_1b_1a_2b_2a_3b_3\cdots$			a_nb_n
$a_1b_1a_2b_2a_3b_3\cdots a_{n-1}b_{n-1}$			

Namely, this database has n records each of which contains $(n-1)$ pairs of items but only one pair is missing. It may produce an exponential number of frequent patterns. The experimental results with frequency threshold $\alpha = 1$ are shown in Table 3. We can observe the exponential explosion of the number of patterns, but only linear size of ZBDDs are needed for representing such a huge number of patterns. In such cases, ZBDD-growth runs extremely fast, while FP-growth requires exponential time depending on the output data size.

Table 3: Results for "one-pair-missing" with $\alpha = 1$.

n	#Patterns	(output) |ZBDD|	ZBDD-growth Time(sec)	FP-growth Time(sec)
8	58,974	35	0.01	0.11
9	242,460	40	0.01	0.47
10	989,526	45	0.01	1.93
11	4,017,156	50	0.01	7.81
12	16,245,774	55	0.01	32.20
13	65,514,540	60	0.01	131.15
14	263,652,486	65	0.02	518.90
15	1,059,392,916	70	0.02	1966.53
16	4,251,920,574	75	0.02	(timeout)

4.2 Experiments for Benchmark Examples

Next we show the results for the benchmark examples[8], written in previous section.

Table 4 shows the time and space for generating ZBDD vectors of tuple-histograms. In this table, $\#T$ shows the number of tuples, $total|T|$ is the total of tuple sizes (total appearances of items), and |ZBDD| is the number of ZBDD nodes for the tuple-histograms. We can see that tuple-histograms can be constructed for all instances in

[1]Permit operation is basically same as *SubSet* operation by Coudert et al.[6], defined for ordinary BDDs.

Table 4: Generation of tuple-histograms.

Data name	$\#T$	$total\|T\|$	|ZBDD Vector|	Time(s)
T10I4D100K	100,000	1,010,228	552,429	43.2
T40I10D100K	100,000	3,960,507	3,396,395	150.2
chess	3,196	118,252	40,028	1.4
connect	67,557	2,904,951	309,075	58.1
mushroom	8,124	186,852	8,006	1.5
pumsb	49,046	3,629,404	1,750,883	188.5
pumsb_star	49,046	2,475,947	1,324,502	123.6
BMS-POS	515,597	3,367,020	1,350,970	895.0
BMS-WebView-1	59,602	149,639	46,148	18.3
BMS-WebView-2	77,512	358,278	198,471	138.0
accidents	340,183	11,500,870	3,877,333	107.0

a feasible time and space. The ZBDD sizes are almost same or less than $total|T|$.

After generating ZBDD vectors for the tuple-histograms, we applied ZBDD-growth algorithm to generate frequent patterns. Table 5 show the results for the selected benchmark examples, "mushroom," "T10I4D100K," and "BMS-WebView-1." The execution time includes the time for generating the initial ZBDD vectors for tuple-histograms.

The results shows that ZBDD-growth is much faster than FP-tree for "mushroom," but not effective for "T10I4D100K." "T10I4D100K" is known as an artificial database, consists of randomly generated combinations, so there are almost no relationship between the tuples. In such cases, ZBDD nodes cannot be shared well, and only the overhead factor is revealed. For "BMS-WebView-1," ZBDD-growth is slower than FP-growth when the output size is small, however, an exponential factor of reduction is observed for the cases of generating huge patterns. Especially for $\alpha = 31, 30$, more than 1 Tera patterns are generated and compactly stored in the memory, that has never been possible by using conventional data structures.

4.3 Maximal Frequent Pattern Mining

We also show the experimental results of maximal frequent pattern mining using ZBDD-growth-max algorithm. In Table 6, we show the results for the same examples as used in the experiment of original ZBDD-growth. The last column $Time_{(max)}/Time_{(all)}$ shows the ratio of computation time between the ZBDD-growth-max and the original ZBDD-growth algorithm. We can observe that the computation time is almost the same (up to twice) between the two algorithms. In other words, the additional computation cost for ZBDD-growth-max is almost the same order as the original algorithm. Our ZBDD-based "permit" operation can efficiently filter the maximal patterns within a time depending on the ZBDD size, which is almost the same cost as manipulating ZBDD vectors of tuple-histograms.

5 Post Processing for Generated Frequent Patterns

Our ZBDD-based method features that the algorithm uses ZBDDs not only as internal data structure but also as output data structure indexing a huge number of patterns compactly on the main memory. The results can be analyzed flexibly by using algebraic operations implemented on ZBDDs. Here we show several examples of the post processing operations for the output data.

Sub-pattern matching for the frequent patterns

For the result of frequent patterns F, we can efficiently filter a subset S, such that each pattern in S contains a given sub-pattern P.

Table 5: Results for benchmark examples.

Data name: Min. freq. α	#Frequent patterns	(output) |ZBDD|	ZBDD-growth Time(s)	FP-growth Time(s)
mushroom:				
8,124	1	1	1.2	0.1
5,000	41	11	1.2	0.1
1,000	123,277	1,417	3.7	0.3
200	18,094,821	12,340	9.7	5.4
50	198,169,865	36,652	10.2	44.6
16	1,176,182,553	53,804	7.7	244.1
4	3,786,792,695	59,970	4.3	891.3
1	5,574,930,437	40,557	1.8	1,322.5
T10I4D100K:				
100,000	0	0	78.5	0.7
5,000	10	10	81.3	0.7
1,000	385	382	135.5	3.1
200	13,255	4,288	279.4	4.5
50	53,385	20,364	408.7	6.7
16	175,915	89,423	543.3	13.7
4	3,159,067	1,108,723	646.0	38.8
1	2,217,324,767	(mem.out)	–	317.1
BMS-WebView-1:				
59,602	0	0	27.3	0.2
1,000	31	31	27.8	0.2
200	372	309	31.3	0.4
50	8,191	3,753	49.0	0.8
40	48,543	12,176	46.6	1.2
36	461,521	34,790	102.4	2.7
35	1,177,607	47,457	111.4	4.2
34	4,849,465	64,601	120.8	8.3
33	69,417,073	80,604	130.0	28.1
32	1,531,980,297	97,692	133.7	345.3
31	8,796,564,756,112	117,101	138.1	(timeout)
30	35,349,566,550,691	152,431	143.9	(timeout)

$S = F$
forall $v \in P$ **do:**
  $S = S.\text{onset}(v).\text{change}(v)$
return S

Inversely, we can extract a subset of patterns not satisfying the given conditions. It is easily done by computing $F - S$. The computation time for the sub-pattern matching is much smaller than the time for frequent pattern mining.

The above operations are sometimes called constraint pattern mining. In conventional method, it is too time consuming to generate all frequent patterns before filtering. Therefore, many researchers consider the direct methods of constraint pattern mining without generating all patterns. However, using ZBDD-based method, a huge number of patterns can be stored and indexed compactly on the memory, so in many cases, it is possible to generate all frequent patterns and then processing them using algebraic ZBDD operations.

Extracting Long/Short Patterns

Sometimes we are interested in the long/short patterns, consists of a large/small number of items. Using ZBDDs, all combinations of less than k out of

Table 6: Results of maximal pattern mining.

Data name: Min. freq. α	#Maximal frequent patterns	(output) |ZBDD|	ZBDD-growth-max Time(s)	$Time_{(max)}$ /$Time_{(all)}$
mushroom:				
8,124	1	1	1.2	0.99
5,000	3	10	1.2	1.00
1,000	467	744	4.1	1.10
200	3,111	4,173	10.7	1.10
50	9,857	10,223	11.0	1.08
16	24,060	13,121	8.1	1.06
4	39,456	14,051	4.2	0.98
1	8,124	8,006	1.2	0.70
T10I4D100K:				
5,000	10	10	107.1	1.32
1,000	370	376	203.1	1.50
200	1,938	2,609	462.8	1.66
50	12,062	13,259	787.8	1.93
16	68,096	66,274	922.4	1.70
4	400,730	372,993	1141.2	1.77
1	77,443	532,061	140.5	–
BMS-WebView-1:				
1,000	29	30	34.9	1.25
200	264	289	41.2	1.32
50	3,546	3,064	71.2	1.45
40	9,827	8,260	110.1	1.44
36	15,179	14,345	149.5	1.46
35	15,725	15,713	161.5	1.45
34	15,877	16,854	173.1	1.43
33	15,753	16,854	183.8	1.41
32	15,252	17,680	196.6	1.47
31	13,639	17,383	208.7	1.51
30	11,371	16,323	219.7	1.53

n items are efficiently represented in a polynomial size, bounded by $O(k \cdot n)$. This ZBDD represents a length constraint of patterns. We then apply intersection (or difference) operation to the frequent patterns with the length constraint of ZBDD. In this way, we can easily extract a set of long/short frequent patterns.

Comparison of Two Sets of Frequent Patterns

Our ZBDD manipulation environment can efficiently store more than one results of frequent pattern mining together. So, we can compare the two sets of frequent patterns generated with different conditions. For example, if the database is gradually changing as time passing, the tuple-histograms and frequent patterns are not the same forever. Our ZBDD-based method can store and index a number of snapshot of pattern sets and easily show the intersection, union, and difference between any pair of snapshots. When many similar ZBDDs are generated, their ZBDD nodes are effectively shared into a monolithic multi-rooted graph, so the memory requeirement is much less than storing each ZBDD separately.

Calculating Statistical Data

After generating a ZBDD for a set of patterns, we can quickly count a number of patterns by using a primitive ZBDD operation S.count. The computation time is linearly bounded by ZBDD size, not depending on the amount of pattern counts. We can also efficiently calculate other statistical measures, such as *Support* and *Confidence*, which are often used in probabilistic analysis and machine learning.

Finding Disjoint Decompositions in Frequent Patterns

In the recent paper[16], we presented an efficient ZBDD-based method for finding all possible *simple disjoint decompositions* in a set of combinations. If a given set of patterns f can be decomposed as $f(X,Y) = g(h(X),Y)$, and X and Y has no common items, we then call it a simple disjoint decomposition.

This decomposition method extracts another aspect of hidden structures from complicated itemset data. The decomposition procedure is enough fast for handling large-scale sets of patterns. It will be a powerful tool for database analysis.

6 Conclusion

In this paper, we presented a new method of ZBDD-based frequent pattern mining algorithm. Our method generates a ZBDD for a set frequent patterns from the ZBDD vector for the tuple-histogram of a given transaction database. Our experimental result shows that our ZBDD-growth algorithm is fast as competitive to the existing state-of-the-art algorithms, such as FP-growth. Especially for the cases where the ZBDD nodes are well shared, exponential speed up are observed comparing to the existing algorithms based on explicit table/tree representation. On the other hand, for the cases where ZBDDs are not well shared, or number of patterns is very small, ZBDD-growth method is not effective and the overhead factor reveals.

However, we do not have to use ZBDD-growth algorithm for all instances. We may use the existing methods for the instances where they are more effective than ZBDD-growth. In addition, we can develop a hybrid program that using FP-tree or simple array for internal data structure, but the output is constructed as a ZBDD.

A ZBDD can be regarded as a compressed trie for representing a set of patterns. ZBDD-based method will be useful as a fundamental techniques for database analysis and knowledge indexing, and will be utilized for various data mining 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, edtors, *Proc. of the 1993 ACM SIGMOD International Conference on Management of Data*, Vol. 22(2) of SIGMOD Record, pp. 207–216, ACM Press, 1993.

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

[3] J.-F. Boulicaut, Proc. 2nd International Workshop on Knowledge Discovery in Inductive Databases (KDID'03), Cavtat-Dubrovnik, 2003.

[4] Bryant, R. E., Graph-based algorithms for Boolean function manipulation, IEEE Trans. Comput., C-35, 8 (1986), 677–691.

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

[6] O. Coudert, J. C. Madre, H. Fraisse, A new viewpoint on two-level logic minimization, *in Proc. of 30th ACM/IEEE Design Automation Conference*, pp. 625-630, 1993.

[7] B. Goethals, "Survey on Frequent Pattern Mining", Manuscript, 2003. `http://www.cs.helsinki.fi/u/goethals/publications/survey.ps`

[8] B. Goethals, M. Javeed Zaki (Eds.), Frequent Itemset Mining Dataset Repository, Frequent

Itemset Mining Implementations (FIMI'03), 2003. http://fimi.cs.helsinki.fi/data/

[9] J. Han, J. Pei, Y. Yin, R. Mao, Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach, Data Mining and Knowledge Discovery, 8(1), 53–87, 2004.

[10] L. Jiang, M. Inaba, and H. Imai: A BDD-based Method for Mining Association Rules, in Proceedings of 55th National Convention of IPSJ, Vol. 3, pp. 397-398, Sept. 1997, IPSJ.

[11] H. Mannila, H. Toivonen, Multiple Uses of Frequent Sets and Condensed Representations, In *Proc. KDD*, 189–194, 1996.

[12] S. Minato: Zero-suppressed BDDs for set manipulation in combinatorial problems, In Proc. 30th ACM/IEEE Design Automation Conf. (DAC-93), (1993), 272–277.

[13] S. Minato: "Binary Decision Diagrams and Applications for VLSI CAD", Kluwer Academic Publishers, November 1996.

[14] S. Minato, Zero-suppressed BDDs and Their Applications, International Journal on Software Tools for Technology Transfer (STTT), Springer, Vol. 3, No. 2, pp. 156–170, May 2001.

[15] S. Minato and H. Arimura: Efficient Combinatorial Item Set Analysis Based on Zero-Suppressed BDDs", In *Proc. of IEEE/IEICE/IPSJ International Workshop on Challenges in Web Information Retrieval and Integration (WIRI-2005)*, pp. 3-10, Apr., 2005.

[16] S. Minato: Finding Simple Disjoint Decompositions in Frequent Itemset Data Using Zero-suppressed BDD, In *Proc. of IEEE ICDM 2005 workshop on Computational Intelligence in Data Mining*, pp. 3-11, ISBN-0-9738918-5-8, Nov. 2005.

[17] H. Okuno, S. Minato, and H. Isozaki: On the Properties of Combination Set Operations, Information Procssing Letters, Elsevier, 66 (1998), pp. 195-199, 1998.

[18] Ricardo Baeza-Yates, Berthier Ribiero-Neto, "Modern Information Retrieval", Addison Wesley, 1999.

[19] M. J. Zaki, Scalable Algorithms for Association Mining, IEEE Trans. Knowl. Data Eng. 12(2), 372–390, 2000.