Frequent Pattern Mining and Knowledge Indexing Based on Zero-Suppressed BDDs

Shin-ichi Minato and Hiroki Arimura

Graduate School of Information Science and Technology,
Hokkaido University, Sapporo, 060-0814 Japan

Abstract. Frequent pattern mining is one of the fundamental techniques for knowledge discovery and data mining. During the last decade, several efficient algorithms for frequent pattern mining have been presented, but most algorithms have focused on enumerating the patterns that satisfy the given conditions, considering the storage and indexing of the pattern results for efficient inductive analysis to be a separate issue. In this paper, we propose a fast algorithm for extracting all/maximal frequent patterns from transaction databases and simultaneously indexing a huge number of patterns using Zero-suppressed Binary Decision Diagrams (ZBDDs). Our method is comparably fast as existing state-of-the-art algorithms and not only enumerates/lists the patterns but also compactly indexes the output data in main memory. After mining, the pattern results can be analyzed efficiently by using algebraic operations. BDD-based data structures have previously been used successfully in VLSI logic design, but our method is the first practical application of BDD-based techniques in the data mining area.

1 Introduction

Frequent pattern mining is one of the fundamental techniques for knowledge discovery and data mining. Since their introduction by Agrawal et al. [1], frequent pattern mining and association rule analysis have received much attention from researchers, and many papers have been published about new algorithms and improvements for solving such mining problems [10,12,24]. However, most of these pattern-mining algorithms have focused on enumerating or listing the patterns that satisfy the given conditions, considering the storage and indexing of the pattern results for efficient inductive analysis to be a separate issue.

In this paper, we propose a fast algorithm for extracting all/maximal frequent patterns from transaction databases and simultaneously indexing a huge number of result patterns in computer memory using Zero-suppressed Binary Decision Diagrams (ZBDDs). Our method not only enumerates/lists the patterns but also indexes the output data compactly in main memory. After mining, the pattern results can be analyzed efficiently by using algebraic operations.

The key to our method is the use of data structures based on Binary Decision Diagrams (BDDs) to represent sets of patterns. BDDs [5] are graph-based representations of Boolean functions and are now widely used in the VLSI logic design and verification area. For data mining applications, it is important to use the ZBDD

S. Džeroski and J. Struyf (Eds.): KDID 2006, LNCS 4747, pp. 152–169, 2007.

[16], a special type of BDD, which is suited to handling large-scale sets of combinations. Using ZBDDs, we can implicitly enumerate combinatorial itemset data and efficiently compute set operations over the ZBDDs. The preliminary idea of using ZBDDs was presented in our previous workshop paper [19]. In this paper, we propose a fast pattern mining algorithm based on this data structure. Our work is the first practical application of the BDD-based technique to the data mining area.

In related work, the *FP-tree* [12] receives a great deal of attention because it supports fast manipulation of large-scale itemset data using a compact tree structure in main memory. Our method uses a similar approach to handling sets of combinations in main memory but is more efficient in these respects:

- A ZBDD is a kind of Directed Acyclic Graph (DAG) for representing itemsets, while the FP-tree is a tree representation. In general, DAGs can be more compact than trees.
- Our method uses ZBDDs not only as the internal data structure but also as the output data structure. This provides an efficient knowledge index for subsequent inductive analysis.

Our mining algorithm is based on a recursive depth-first search of the database represented by ZBDDs. We show two versions of the algorithm, generating all frequent patterns and generating maximal frequent patterns. Experimental results show that our method is comparably fast as existing state-of-the-art algorithms, such as those based on FP-trees. Especially for cases where the ZBDD nodes are well shared, exponential speed-up is observed compared with existing algorithms based on explicit table/tree representations.

Recently, data mining methods have often been discussed in the context of Inductive Databases [3,14], the integrated processes of knowledge discovery. In this paper, we also show a number of examples of postprocessing following frequent pattern mining. We place the ZBDD-based method at the core of integrated discovery processes that efficiently execute various operations to find interest patterns and analyze the information included in large-scale combinatorial itemset databases.

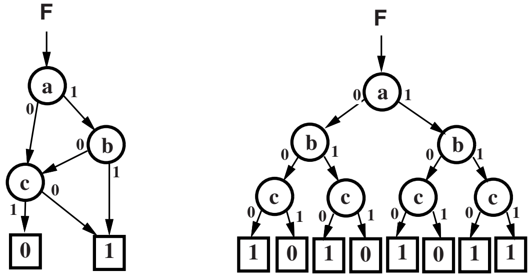
2 BDDs and Zero-Suppressed BDDs

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

2.1 BDDs

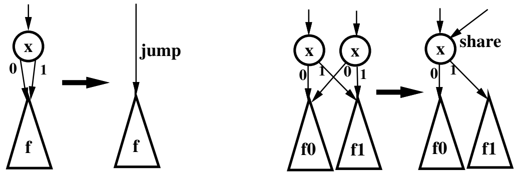
A BDD is a directed graph representation of a Boolean function, as illustrated in Fig. 1(a). It is derived by reducing a binary tree graph representing the recursive *Shannon's expansion*, shown in Fig. 1(b). The following reduction rules yield a *Reduced Ordered BDD (ROBDD)*, which can efficiently represent the Boolean function (see [5] for details).

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



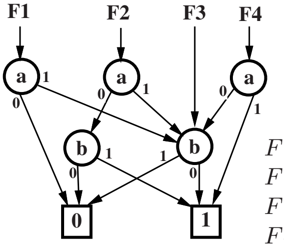
(a) BDD. (b) Binary tree.

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



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

Fig. 2. Reduction rules of ordinary BDDs



$F1 = a \wedge \overline{b}$
$F2 = a \oplus b$
$F3 = \overline{b}$
$F4 = a \vee \overline{b}$

Fig. 3. Shared multiple BDDs

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

As shown in Fig. 3, a set of multiple BDDs can share their subgraphs with 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 including AND, OR, parity, and arithmetic adder functions. Using Bryant's algorithm [5], we can efficiently construct a BDD for the result of a binary logic operation (e.g. AND, OR, XOR) on a given pair of operand BDDs. This

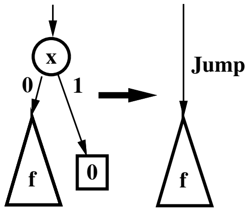


Fig. 4. ZBDD reduction rule

algorithm is based on hash table techniques, and the computation time is almost linearly related to the data size unless there is data overflow in main memory (see [5] or [17] for details).

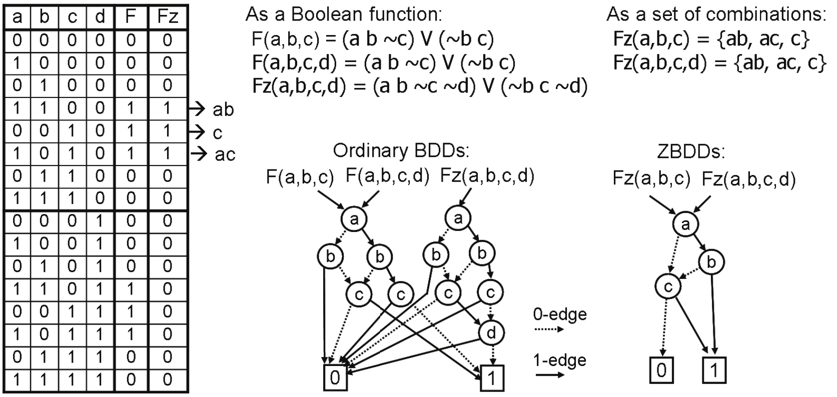
Based on these techniques, several BDD packages were developed in the 1990s and are widely used for large-scale Boolean function manipulation, especially in the VLSI CAD area.

2.2 Sets of Combinations and ZBDDs

BDDs were originally developed for handling Boolean function data. However, they can also be used for the implicit representation of sets of combinations. Here we use the term "sets of combinations" for a set of elements each of which is a combination 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 networks.

A combination of n items can be represented by an n-bit binary vector, $(x_1x_2 \ldots x_n)$, where each bit, $x_k \in \{1, 0\}$, expresses whether the item is included in the combination or not. 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 particular 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, the left side of Fig. 5 shows a truth table representing a Boolean function $(ab\overline{c}) \vee (\overline{b}c)$ but also represents a set of combinations $\{ab, ac, c\}$. 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 combinations that include many similar patterns (subcombinations), the BDDs are greatly reduced by the node-sharing effect, and sometimes an exponential reduction in processing time and space can be obtained.



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

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

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

Fig. 5. Effect of ZBDD reduction rule

The **ZBDD [16,18]** is a special type of BDD, used for the efficient manipulation of sets of combinations. ZBDDs are based on the following special reduction rules.

- Delete all nodes whose 1-edge directly points to the 0-terminal node, and jump through to the 0-edge's destination, as shown in Fig. 4.
- Share equivalent nodes, similarly to ordinary BDDs.

Note that we do not delete the nodes whose two edges point to the same node, which would have been 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 also give canonical forms as do ordinary BDDs under a fixed variable ordering.

Here we summarize the features of ZBDDs:

- In ZBDDs, the nodes of irrelevant items (i.e. never chosen in any combination) are automatically deleted by the ZBDD reduction rule. In ordinary BDDs, irrelevant nodes remain and may compromise the reduction available by sharing nodes. An example is shown in Fig. 5. In this case, the item d is irrelevant, but the ordinary BDDs for characteristic functions $Fz(a, b, c)$ and $Fz(a, b, c, d)$ have different forms. On the other hand, ZBDDs for $Fz(a, b, c)$ and $Fz(a, b, c, d)$ have identical forms and are therefore completely shared.
- Each path from the root node to the 1-terminal node corresponds to each combination in the set. That is, the number of such paths in the ZBDD equals the number of combinations in the set. In ordinary BDDs, this property does not always hold.
- When no equivalent nodes exist in a ZBDD, i.e. the worst case, the ZBDD structure explicitly stores all items in all combinations while also using an explicit linear linked list data structure. That is, (the order of) the ZBDD size never exceeds that of the explicit representation. If more nodes are shared, the ZBDD is more compact than the linear list.

Table 1. Primitive ZBDD operations

"$\emptyset$"	Returns the empty set. (0-terminal node)
"**1**"	Returns a 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.

Table 1 shows most of the primitive operations for ZBDDs. In these operations, The execution time for $\emptyset$, **1**, and *P.top* is a constant, and those for the remainder are almost linearly to the size of the graph. We can describe a variety of processing operations on sets of combinations by compositions of these primitive operations.

2.3 ZBDD-Based Database Analysis

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

For analyzing these large-scale transaction databases, frequent pattern mining [2] and *maximum frequent pattern mining* [6] are especially important and have been discussed actively during the last decade. Since their introduction by Agrawal et al. [1], many papers have been published about new algorithms and improvements for solving such mining problems [10,12,24]. Recently, graph-based methods, such as FP-growth [12], have received a great deal of attention, because they can quickly manipulate large-scale itemset data by constructing compact graph structures in main memory.

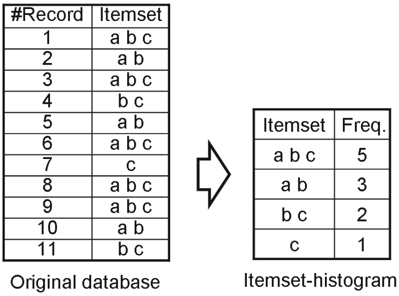
The ZBDD-based method is a similar approach to handling sets of combinations in main memory but is more efficient because ZBDD is a kind of DAG for representing itemsets, while FP-growth uses a tree representation for the same objects. In general, DAGs can be more compact than trees.

Another important point is that our method uses ZBDDs not only as the internal data structure but also as the output data structure. Most of the existing state-of-the-art pattern mining algorithms focus on enumerating or listing the patterns that satisfy the given conditions, and they consider the storage and indexing of the pattern results for efficient data analysis to be a separate issue. In this paper, we present a fast algorithm for pattern mining and simultaneously indexing a huge number of patterns compactly in main memory for subsequent analysis. The results can be analyzed flexibly by using algebraic operations implemented via ZBDDs.

In addition, we will now explain why we use ZBDDs instead of ordinary BDDs for this application. Table 2 lists the basic statistics of a typical data mining

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%
mushroom	119	8,124	186,852	23.0	19.32%
pumsb	2,113	49,046	3,629,404	74.0	3.50%
BMS-WebView-1	497	59,602	149,639	2.5	0.51%
accidents	468	340,183	11,500,870	33.8	7.22%



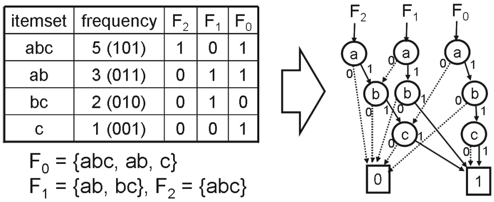
#Record	Itemset
1	a b c
2	a b
3	a b c
4	b c
5	a b
6	a b c
7	c
8	a b c
9	a b c
10	a b
11	b c

Original database

Itemset	Freq.
a b c	5
a b	3
b c	2
c	1

Itemset-histogram

Fig. 6. Example of itemset-histogram



itemset	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}

Fig. 7. ZBDD vector for itemset-histogram

benchmark data [10]. $\#I$ shows the number of items used in the data, $\#T$ is the number of itemsets included in the data, $avg|T|$ is the average number of items per itemset, 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 observation means that we often handle very sparse combinations in many practical data mining/analysis problems, and in such cases, the ZBDD reduction rule is extremely effective. If the average appearance ratio of each item is 1%, ZBDDs are potentially more compact than ordinary BDDs by a factor of up to 100. In the literature, there is an early report by Jiang et al. [13] applying BDDs to data mining problems, but the results seem less than excellent because of the overhead of using ordinary BDDs. Therefore, we should use ZBDDs instead of ordinary BDDs for success in many practical data mining/analysis problems.

3 A ZBDD-Based Pattern-Mining Algorithm

In this section, we first introduce the data structure of the itemset-histogram and ZBDD vectors [19], and then present our new algorithm, ZBDD-growth, which extracts all frequent patterns from a given transaction database using a ZBDD-based data structure.

3.1 Itemset-Histograms and ZBDD Vectors

An *Itemset-histogram* is a table that lists the number of appearances of each itemset in the given database. An example of itemset-histogram is shown in

Fig. 6. This is essentially a compressed table for the database that combines the itemsets appearing more than once into one line, together with the frequency of occurrence.

Our pattern mining algorithm uses a ZBDD-based itemset-histogram representation as the internal data structure, as presented in our previous paper [19]. Here we describe how to represent itemset-histograms using ZBDDs. Because ZBDDs are representations of sets of combinations, a simple ZBDD distinguishes only the existence of each itemset in the database. In order to represent the number of appearances of itemsets, we decompose the number into the m digits of a ZBDD vector $\{F_0, F_1, \ldots, F_{m-1}\}$ to represent integers up to $(2^m - 1)$, as shown in Fig. 7. That is, we encode the appearance numbers into binary digital code, with F_0 representing the itemsets appearing an odd number of times (LSB = 1), F_1 representing the itemsets whose appearance number's second lowest bit is 1, and similarly for each digit up to F_{m-1}.

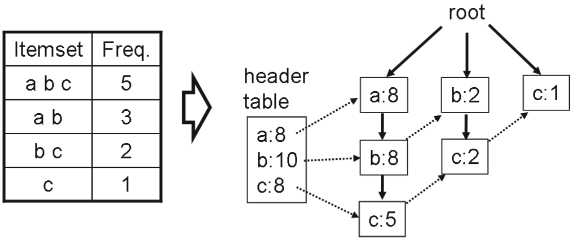
In the example of Fig. 7, the frequencies of itemsets are decomposed as: $F_0 = \{abc, ab, c\}$, $F_1 = \{ab, bc\}$, $F_2 = \{abc\}$, following which each digit can be represented by a simple ZBDD. The three ZBDDs share their subgraphs with each other.

Now we explain the procedure for constructing a ZBDD-based itemset-histogram from an original database. We read the itemset data one by one from the database and accumulate the single itemset data into the histogram. More precisely, we generate a ZBDD of T for a single itemset picked up from the database and accumulate it into 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 itemset. 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 T up 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 itemset-histogram is complete.

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

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

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



Itemset	Freq.
a b c	5
a b	3
b c	2
c	1

Fig. 8. Example of FP-tree

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

- H.factor0(v): Extracts sub-histogram of itemsets without item v.
- H.factor1(v): Extracts sub-histogram of itemsets including item v and then deletes v from the itemsets (also considered as the quotient of H/v).
- $v \cdot H$: Attaches an item v to each itemset in the histogram F.
- $H_1 + H_2$: Generates a new itemset-histogram with the sum of the frequencies of corresponding itemsets.
- H.count: The number of itemsets 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 bound is approximately linearly related to the total ZBDD size.

3.2 ZBDD Vectors and FP-Trees

FP-growth [12], one of the state-of-the-art algorithms, constructs an "FP-tree" for a given transaction database and then searches frequent patterns using this data structure. An example of an FP-tree is shown in Fig. 8. We can see that the FP-tree is a trie [9] of itemsets with their frequencies. In other words, **FP-growth is based on the tree representation of itemset-histograms.** That is, ZBDD-growth is logically based on the same internal data structure as FP-growth. This is the reason for calling this algorithm ZBDD-growth. However, the ZBDD-based method will be more efficient because ZBDDs can share equivalent subgraphs and the computation time is bounded by the ZBDD size. The benefit of ZBDDs is especially noticeable when large numbers of patterns are produced.

3.3 A Frequent Pattern Mining Algorithm

Our new algorithm, ZBDD-growth, is based on a recursive depth-first search over the ZBDD-based itemset-histogram representation. The basic algorithm is shown in Fig. 9.

```
ZBDDgrowth($H, \alpha$)
{
  **if**($H$ has only one item $v$)
     **if**($v$ appears more than $\alpha$ )
         **return** $v$ ;
     **else return** "0" ;
  $F \leftarrow$ Cache($H$) ;
  **if**($F$ exists) **return** $F$ ;
  $v \leftarrow H.top$ ; /* Top item in $H$ */
  $H_1 \leftarrow H$.factor1($v$) ;
  $H_0 \leftarrow H$.factor0($v$) ;
  $F_1 \leftarrow$ZBDDgrowth($H_1, \alpha$) ;
  $F_0 \leftarrow$ZBDDgrowth($H_0 + H_1, \alpha$) ;
  $F \leftarrow (v \cdot F_1) \cup F_0$ ;
  Cache($H$) $\leftarrow F$ ;
  **return** $F$ ;
}
```

Fig. 9. ZBDD-growth algorithm

```
ZBDDgrowthMax($H, \alpha$)
{
  **if**($H$ has only one item $v$)
     **if**($v$ appears more than $\alpha$ )
         **return** $v$ ;
     **else return** "0" ;
  $F \leftarrow$ Cache($H$) ;
  **if**($F$ exists) **return** $F$ ;
  $v \leftarrow H.top$ ; /* Top item in $H$ */
  $H_1 \leftarrow H$.factor1($v$) ;
  $H_0 \leftarrow H$.factor0($v$) ;
  $F_1 \leftarrow$ZBDDgrowthMax($H_1, \alpha$) ;
  $F_0 \leftarrow$ZBDDgrowthMax($H_0 + H_1, \alpha$) ;
  $\boxed{F \leftarrow (v \cdot F_1) \cup (F_0 - F_0.\mathrm{permit}(F_1)) ;}$
  Cache($H$) $\leftarrow F$ ;
  **return** $F$ ;
}
```

Fig. 10. ZBDD-growth-max algorithm

In this algorithm, we choose an item v used in the itemset-histogram H and compute the two sub-histograms H_1 and H_0 (i.e. $H = (v \cdot H_1) \cup H_0$). As v is the top item in the ZBDD vector, H_1 and H_0 can be obtained simply by referring to 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, while 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 ZBDD is computed. This procedure may theoretically require an exponential number of recursive calls. However, we can 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 itemset-histogram, and F is the pointer to the result of the ZBDD. On each recursive call, we check the cache to see if 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 linearly related to the total ZBDD size.

In our implementation, we include some simple techniques for pruning the search space. For example, if H_1 and H_0 are equivalent, we may skip the computation of F_0. In other cases, we can halt the recursive calls when the total of frequencies in H is no more than α. Other more elaborate pruning techniques exist, but they need additional computation cost for checking conditions, so they are sometimes but not always effective.

3.4 Extension for Maximal Pattern Mining

We can extend the ZBDD-growth algorithm to extract only the maximal frequent patterns [6], each of which is not included in any other frequent patterns. The algorithm is shown in Fig. 10.

```
$P$.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" ;
    $R \leftarrow$ Cache($P, Q$) ;
    if($R$ exists) return $R$ ;
    $v \leftarrow$TopItem($P, Q$) ; /* Top item in $P, Q$ */
    $(P_0, P_1) \leftarrow$factors of $P$ by $v$ ;
    $(Q_0, Q_1) \leftarrow$factors of $Q$ by $v$ ;
    $R \leftarrow (v \cdot P_1.\mathrm{permit}(Q_1)) \cup (P_0.\mathrm{permit}(Q_0 \cup Q_1))$ ;
    Cache($P, Q$) $\leftarrow R$ ;
    return $R$ ;
}
```

Fig. 11. Permit operation

The difference from the original algorithm is in only one line, shown in the frame box. Here, we check each pattern in F_0 and delete it if the pattern is included in one of the patterns of F_1. In this way, we generate only maximal frequent patterns. This is a similar approach to that used in MAFIA [6].

The process of deleting non-maximal patterns is a very time-consuming task. However, we found that one ZBDD-based operation, called the *permit* operation by Okuno et al. [21], can be used to solve this problem[1]. P.permit(Q) returns a set of combinations in P each of which is a subset of some combination 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. 3.4. The computation time of the permit operation is almost linearly related to the ZBDD size.

4 Experimental Results

Here we show the experimental results for evaluating 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 ZBDD nodes in this machine. In these experiments, our implementation of the ZBDD-growth algorithm does not print out the pattern list but constructs the ZBDD results in main memory and counts the number of patterns included in the ZBDD. Counting patterns requires only a time linearly related to the ZBDD size, even if an exponential number of patterns are contained.

For comparison, we also executed the FP-growth algorithm [12], using an implementation by Goethals [11]. This implementation also does not print out the pattern list, only counts the number of patterns.

[1] The Permit operation is similar to the *SubSet* operation of Coudert et al. [8], defined for ordinary BDDs.

Table 3. "One-pair-missing"

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

Table 4. Results for "one-pair-missing"

n	#Patterns	(output)	ZBDD-growth	FP-growth [12]
		|ZBDD|	Time(sec)	Time(sec)
8	58,974	35	0.01	0.11
10	989,526	45	0.01	1.93
12	16,245,774	55	0.01	32.20
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)

Table 5. Generation of itemset-histograms [19]

Data name	$\#T$	$total\|T\|$	|ZBDD Vector|	Time(s)
T10I4D100K	100,000	1,010,228	552,429	43.2
mushroom	8,124	186,852	8,006	1.2
pumsb	49,046	3,629,404	1,750,883	188.5
BMS-WebView-1	59,602	149,639	46,148	18.3
accidents	340,183	11,500,870	3,877,333	107.0

4.1 Experiment with a Mathematical Example

First, we present an experiment for a set of artificial examples where ZBDD-growth is extremely effective. The database, named "one-pair-missing," has the form shown in Table 3. That is, this database has n records, each of which contains $(n-1)$ pairs of items with only one pair missing. It may produce an exponentially increasing number of frequent patterns.

The experimental results with frequency threshold $\alpha = 1$ are shown in Table 4. We observe the exponential explosion of the number of patterns, compared with the linearly increasing sizes of the ZBDDs needed to represent such a large number of patterns. In such cases, ZBDD-growth runs extremely fast, while FP-growth requires a time exponentially related to the output data size.

4.2 Experiments for Benchmark Examples

Next, we show the results for the benchmark examples [11]. Table 5 shows the time and space required to generate ZBDD vectors of itemset-histograms [19] as the preprocessing of the ZBDD-growth algorithm. In this table, $\#T$ shows the number of itemsets, $total|T|$ is the total of itemset sizes (total appearances of items), and |ZBDD| is the number of ZBDD nodes for the itemset-histograms. We see that itemset-histograms can be constructed for all cases within a feasible time and space. The ZBDD sizes are similar to or less than $total|T|$.

After generating ZBDD vectors for the itemset-histograms, we applied the ZBDD-growth algorithm to generate frequent patterns. Table 6 shows 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 itemset-histograms. The results show that ZBDD-growth is much faster than FP-growth for "mushroom" but is not as effective for "T10I4D100K". This is a reasonable result because "T10I4D100K" is known to

Table 6. Results for benchmark examples

Data name:	#Frequent	(output)	ZBDD-growth	FP-growth [11]
Min. freq. α	patterns	$\lvert$ZBDD$\rvert$	Time(s)	Time(s)
mushroom: 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
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: 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
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-WebView1: 1,000	31	31	27.8	0.2
200	372	309	31.3	0.4
50	8,191	3,753	49.0	0.8
34	4,849,465	64,601	120.8	8.3
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)

be an artificial database comprising completely random combinations, so there are very few relationships between the itemsets. In such cases, the compression of ZBDDs is not effective, 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 in cases that generate many patterns. Especially for $\alpha = 31$ and 30, more than one trillion patterns are generated and compactly stored in memory, which has not been possible when using conventional data structures.

4.3 Maximal Frequent Pattern Mining

We also show the experimental results for maximal frequent pattern mining using the ZBDD-growth-max algorithm. In Table 7, we show the results for the same examples used in the original ZBDD-growth experiment. The last column $Time_{(max)}/Time_{(all)}$ shows the ratio of computation time between ZBDD-growth-max and the original ZBDD-growth algorithm. We observe that the computation time is almost the same (within a factor of two) for 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 that depends on the ZBDD size, which is almost the same cost as manipulating ZBDD vectors of itemset-histograms.

In many conventional methods, maximal pattern mining is less time consuming than generating all patterns because there are many fewer maximal patterns than all patterns. However, the complexity of ZBDD-growth does not directly depend on the number of patterns. We observe that ZBDD size is not significantly different between the maximal and all-pattern cases, so the computation time is also not significantly different.

Table 7. Results of maximal pattern mining

Data name: Min. freq. α	#Maximal freq. patterns	(output) |ZBDD|	ZBDD-growth-max Time(s)	$Time_{(max)}$ $/Time_{(all)}$
mushroom: 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
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
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-WebView1: 1,000	29	30	34.9	1.25
200	264	289	41.2	1.32
50	3,546	3,064	71.2	1.45
34	15,877	16,854	173.1	1.43
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

5 Postprocessing for Generated Frequent Patterns

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

(Subpattern matching for the frequent patterns): From the frequent-pattern results F, we can efficiently filter a subset S, such that each pattern in S contains a given sub-pattern P.

```
$S = F$
forall $v \in P$ do:
$S = S$.onset($v$).change($v$)
return $S$
```

Conversely, we can extract a subset of patterns not satisfying the given conditions. This 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 methods, it is too time consuming to generate all frequent patterns before filtering. Therefore, many researchers consider direct methods of constraint pattern mining without generating all patterns. However, using the ZBDD-based method, a large number of patterns can be stored and indexed compactly in main memory. In many cases, therefore, it is possible to generate all frequent patterns and then process them using algebraic ZBDD operations.

(Extracting Long/Short Patterns): Sometimes we are interested in the long/short patterns, comprising a large/small number of items. Using ZBDDs, all combinations of less than k out of n items are efficiently represented in polynomial size, bounded by $\mathrm{O}(k \cdot n)$. This ZBDD represents a length constraint on patterns. We then apply an intersection (or difference) operation to the frequent patterns that meet the length constraint of the ZBDD. In this way, we can easily extract a set of long/short frequent patterns.

(Comparison between Two Sets of Frequent Patterns): Our ZBDD manipulation environment can efficiently store more than one set of results of frequent pattern mining. Therefore, we can compare two sets of frequent patterns generated under different conditions. For example, if a database gradually changes over time, the itemset-histograms and frequent patterns do not stay the same. Our ZBDD-based method can store and index a number of snapshots 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 within a monolithic multi-rooted graph, requiring much less memory than that required to store each ZBDD separately.

(Calculating Statistical Data): After generating a ZBDD for a set of patterns, we can quickly count the number of patterns by using a primitive ZBDD operation S.count. The computation time is linearly bounded by the ZBDD size, not depending on the pattern count. 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 a recent paper [20], 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 $f(X, Y) = g(h(X), Y)$, with X and Y having no common items, then we call it a simple disjoint decomposition. The decomposition method can be applied to the result of our ZBDD-growth algorithm, and we can extract other aspects of hidden structures from the complex itemset data. This will be a powerful tool for database analysis.

6 Related Works

A ZBDD can be regarded as a compressed representation of a *trie* [9] for a set of patterns, by sharing subgraphs of the tree structure. From this viewpoint, it can be compared with the existing state-of-the-art condensed representations, such as closed sets [22], free sets [4], and non-derivable itemsets [7].

Recently, Mielikäinen et al. [15] reported a fast method of answering itemset support query for frequent itemsets using a condensed representation. Their data structure is based on a trie with a frequency number on each node. Using this data structure, they represent a histogram of the frequent patterns occurring more than α times. In this way, counting the occurrence number for a given pattern can extremely be accelerated. In addition, they also proposed some techniques not to store all frequent patterns in the trie, by only storing a

Table 8. Comparison with a condensed representation [15]

Data name (min. freq.)	#freq.	condensed rep. [15] (in KB)		ZBDD (in KB)	
	patterns	all-freq-hist	closed-freq-hist	all-freq-set	all-freq-hist
mushroom (α: 500)	1,442,503	56,348	420	160	1,513
pumsb (α: 35,000)	1,897,479	74,120	10,416	250	16,293
BMS-WebView-1 (α: 35)	1,177,607	46,000	3,308	1,898	10,014

part of patterns (e.g. closed patterns) to save the memory requirement without information loss.

Here we show an experiment to compare our ZBDD-based method with Mielikäinen's results. In the Table 8, the first column shows the data name with the minimum frequency α. The second column shows the total amount of all frequent patterns for α. In the columns of condensed representations, "all-freq-hist" shows the memory requirement in KByte to represent a histogram of all frequent patterns in the trie, and "closed-freq-hist" shows the results only storing closed patterns in the trie. In the columns of our ZBDD-based representation, "all-freq-set" shows the ZBDD size in KByte to represent a set of all frequent patterns (without counting frequency number for each patterns), and "all-freq-hist" shows the size of a ZBDD vector to represent the histogram of frequent patterns. Here we assume that one ZBDD node consumes 40 Byte in average.

In general, ZBDD-based "all-freq-set" is much more compact than using a trie, since the equivalent subgraphs are shared in a ZBDD. Notice that a simple ZBDD represents just a set of frequent patterns, but not representing a histogram. To store the occurrence numbers exactly, we have to use ZBDD vectors, shown as "all-freq-hist" in the table. This is still more compact than trie-based "all-freq-hist".

The condensed representation "closed-freq-hist" would be more powerful than using ZBDDs in terms of memory reduction. They use a domain-specific property of frequent itemsets, i.e. monotonous relation. However, if we consider more various inductive queries after generating frequent patterns, it would not work well because such a beautiful property of itemsets may be broken. ZBDDs can be used more robustly as they are based on more general data compression principles. The results may depend on what kind of operations are performed after generating patterns. Analysis of those data efficiencies will be an interesting future work.

7 Conclusion

In this paper, we have presented a new ZBDD-based frequent pattern-mining algorithm. Our method generates a ZBDD for a set of frequent patterns from the ZBDD vector for the itemset-histogram of a given transaction database. Our experimental results show that our ZBDD-growth algorithm is comparably fast as existing state-of-the-art algorithms such as FP-growth. Especially for the cases where the ZBDD nodes are well shared, an exponential speed-up is observed, compared with existing algorithms based on explicit table/tree representation.

On the other hand, for the cases where ZBDD nodes are not well shared, or the number of patterns is very small, the ZBDD-growth method is not effective and the overhead factors dominate. However, we do not have to use the ZBDD-growth algorithm in all cases. We may use existing methods for cases where they are more effective than ZBDD-growth. In addition, we could develop a hybrid program that uses an FP-tree or a simple array for the internal data structure but with the output constructed as a ZBDD.

The ZBDD-based method will be useful as a fundamental technique for database analysis and knowledge indexing, and will be utilized for various applications in inductive data analysis.

Acknowledgments. This research was partially supported by Grant-in-Aid for Specially Promoted Research on "Semi- Structured Data Mining," 17002008, Ministry of Education, Culture, Sports, Science and Technology of Japan.

References

1. Agrawal, R., Imielinski, T., Swami, A.N.: Mining Association Rules between Sets of Items in Large Databases. In: Buneman, P., Jajodia, S. (eds.) Proc. of the 1993 ACM SIGMOD International Conference on Management (Data of SIGMOD Record), vol. 22(2), pp. 207–216. ACM Press, New York (1993)
2. Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A.I.: Fast Discovery of Association Rules. In: Advances in Knowledge Discovery and Data Mining, pp. 307–328. MIT Press, Cambridge (1996)
3. Boulicaut, J.-F.: Proc. 2nd International Workshop on Knowledge Discovery in Inductive Databases (KDID'03), Cavtat-Dubrovnik (2003)
4. Boulicaut, J.-F., Bykowski, A., Rigotti, C.: Free-sets: A Condensed Representation of Boolean Data for the Approximation of Frequency Queries. Journal of Data Mining and Knowledge Discovery (DMKD) 7(1), 5–22 (2003)
5. Bryant, R.E.: Graph-based Algorithms for Boolean Function Manipulation. IEEE Trans. Computers 35(8), 677–691 (1986)
6. Burdick, D., Calimlim, M., Gehrke, J.: MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases. In: Proc. ICDE 2001, pp. 443–452 (2001)
7. Calders, T., Goethals, B.: Mining All Non-derivable Frequent Itemsets. In: Elomaa, T., Mannila, H., Toivonen, H. (eds.) PKDD 2002. LNCS (LNAI), vol. 2431, pp. 74–85. Springer, Heidelberg (2002)
8. Coudert, O., Madre, J.C., Fraisse, H.: A New Viewpoint on Two-level Logic Minimization. In: Proc. of 30th ACM/IEEE Design Automation Conference, pp. 625–630 (1993)
9. Fredkin, E.: Trie Memory. CACM 3(9), 490–499 (1960)
10. Goethals, B.: Survey on Frequent Pattern Mining, Manuscript (2003), http://www.cs.helsinki.fi/u/goethals/publications/survey.ps
11. Goethals, B., Javeed Zaki, M. (eds.): Frequent Itemset Mining Dataset Repository, Frequent Itemset Mining Implementations (FIMI'03) (2003), http://fimi.cs.helsinki.fi/data/

12. Han, J., Pei, J., Yin, Y., Mao, R.: Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach. Data Mining and Knowledge Discovery 8(1), 53–87 (2004)
13. Jiang, L., Inaba, M., Imai, H.: A BDD-based Method for Mining Association Rules. In: Proceedings of 55th National Convention of IPSJ, IPSJ, September 1997, vol. 3, pp. 397–398 (1997)
14. Mannila, H., Toivonen, H.: Multiple Uses of Frequent Sets and Condensed Representations. In: Proc. KDD, pp. 189–194 (1996)
15. Mielikäinen, T., Panov, P., Dzeroski, S.: Itemset Support Queries using Frequent Itemsets and Their Condensed Representations. In: Todorovski, L., Lavrač, N., Jantke, K.P. (eds.) DS 2006. LNCS (LNAI), vol. 4265, pp. 161–172. Springer, Heidelberg (2006)
16. Minato, S.: Zero-suppressed BDDs for Set Manipulation in Combinatorial Problems. In: Proc. 30th ACM/IEEE Design Automation Conf (DAC-93), pp. 272–277 (1993)
17. Minato, S.: Binary Decision Diagrams and Applications for VLSI CAD. Kluwer Academic Publishers, Dordrecht (1996)
18. Minato, S.: Zero-suppressed BDDs and Their Applications. International Journal on Software Tools for Technology Transfer (STTT) 3(2), 156–170 (2001)
19. Minato, S., Arimura, H.: Efficient Combinatorial Itemset 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 (2005)
20. Minato, S.: 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, November 2005, pp. 3–11. IEEE Computer Society Press, Los Alamitos (2005)
21. Okuno, H., Minato, S., Isozaki, H.: On the Properties of Combination Set Operations. Information Processing Letters 66, 195–199 (1998)
22. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Efficient Mining of Association Rules Using Closed Itemset Lattices. Journal of Information Systems 24(1), 25–46 (1999)
23. Baeza-Yates, R., Ribiero-Neto, B.: Modern Information Retrieval. Addison Wesley, Reading (1999)
24. Zaki, M.J.: Scalable Algorithms for Association Mining. IEEE Trans. Knowl. Data Eng. 12(2), 372–390 (2000)