TCS-TR-A-**05-09**

TCS Technical Report

Finding All Simple Disjoint Decompositions in Frequent Itemset Data

by

Shin-ichi Minato

Division of Computer Science

Report Series A

September 27, 2005

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

Finding All Simple Disjoint Decompositions in Frequent Itemset Data

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

September 27, 2005

(Abstract) In this paper, we propose a method of finding simple disjoint decompositions in frequent itemset data. The techniques for decomposing Boolean functions have been studied for long time in the area of logic circuit design, and recently, there is a very efficient algorithm to find all possible simple disjoint decompositions for a given Boolean functions based on BDDs (Binary Decision Diagrams). We consider the data model called "sets of combinations" instead of Boolean functions, and present a similar efficient algorithm for finding all possible simple disjoint decompositions for a given set of combinations. Our method will be useful for extracting interesting hidden structures from the frequent itemset data on a transaction database. We show some experimental results for conventional benchmark data.

1 Introduction

Manipulation of large-scale combinatorial data is one of the fundamental technique for data mining process. In particular, frequent item set analysis is important in many tasks that try to find interesting patterns from web documents and databases, such as association rules, correlations, sequences, episodes, classifiers, and clusters. Since the introduction by Agrawal et al.[1], the frequent item set 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[6, 8, 22].

After generating frequent itemset data, we sometimes faced with the problem that the frequent itemsets are too large and complicated to retrieve useful information. So, it is an important technique for extracting some hidden structures from the frequent itemsets to make the data more understandable. Closed/maximal itemset mining[23, 20, 21] is one of the useful method in this approach.

In this paper, we propose a new method of finding "simple disjoint decompositions" in the frequent itemset data. Our method extracts another aspect of hidden structures from complicated itemset data, and will be useful for database analysis.

Our method is based on the Boolean function decomposition technique, which is a fundamental theory of logic circuit design. Simple disjoint decomposition is a basic and useful concept in this theory. This decomposition gives a single-output sub-block function whose input variable set is disjoint from the other part. It is a special case of decompositions and not always possible for all Boolean functions. If we find a such decomposition for a given function, it must be a good choice for optimal design, and we may proceed to the local optimization of each sub-block. There are so many studies on the method of finding simple disjoint decompositions, and currently, the method[4][10][11] based on the recursive algorithm using BDDs (Binary Decision Diagrams) is remarkably fast and powerful to find all possible simple disjoint decompositions for a given Boolean functions.

In this paper, we focus on the data model called "sets of combinations", instead of Boolean functions. A set of combinations consists of the ele-

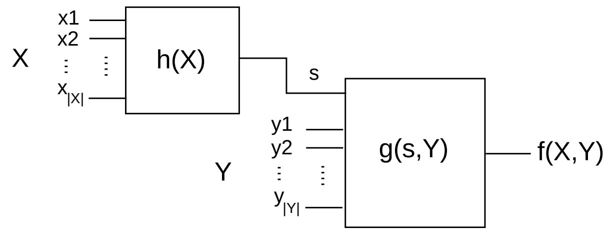


Figure 1: Simple disjoint decomposition on Boolean function.

ments each of which is a combination of multiple items. This data model often appears in many kind of combinatorial problems, and of course, it can be used for representing frequent itemset data. A set of combinations can be mapped into Boolean space of n input variables, and efficiently manipulated by using BDDs. In addition, we can enjoy more efficient manipulation using "Zero-suppressed BDDs" (ZBDD)[12], which are special type of BDDs optimized for handling sets of combinations.

As a major contribution of this paper, we present that we can define the operation of simple disjoint decomposition for sets of combinations, as well as for Boolean functions. We then show that all possible simple disjoint decompositions on sets of combinations can be extracted in a method based on ZBDDs, as well as the conventional BDD-based functional decomposition method. Our new ZBDD-based decomposition algorithm for sets of combinations is not completely same as the BDD-based one for Boolean functions, but they have similar recursive structures and complexities. We also show the experimental results of finding all simple disjoint decompositions in the frequent itemsets extracted from a conventional benchmark dataset.

The paper is organized as follows: First, we review the decomposition methods for Boolean functions in Section 2. In Section 3, we define the decomposition for sets of combinations and present the algorithm for finding simple disjoint decompositions. We then show the experimental results in section 4, followed by conclusion.

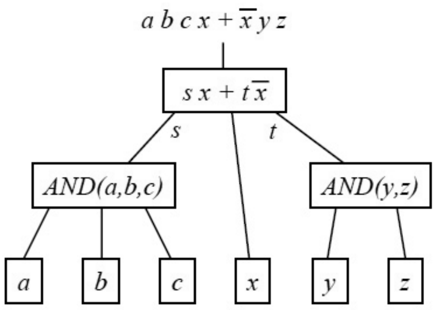


Figure 2: Tree structure of simple disjoint decompositions.

2 Boolean Function Decomposition

At first, we review the methods of Boolean function decomposition. If the function f can be represented as $f(X,Y) = g(h(X),Y)$, then f can be realized by the network shown in Fig. 1. We call it *simple disjoint decomposition.* It is called "simple" because h is a single-output function and "disjoint" because X and Y have no common variables. We call a trivial decomposition if X consists of only one variable. A non-trivial simple disjoint decomposition does not always exist in a given function, but if exists, it is considerably effective for logic optimization.

A function may have more than one simple disjoint decompositions. They can be nesting. For example, the function $abcx + \overline{x}yz$ has five decompositions as $X = \{a,b\}$, $\{b,c\}$, $\{a,c\}$, $\{a,b,c\}$, and$\{y,z\}$.

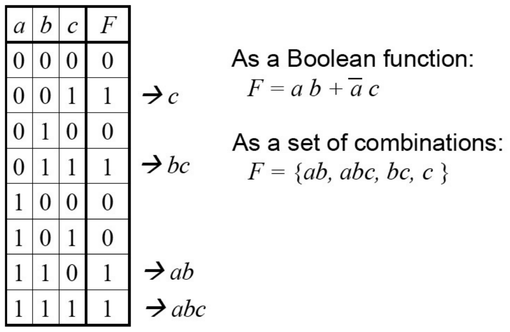
Multiple input logic operations (AND, OR, EXOR) may produce a number of associative sub-decompositions. In such cases, we handle those decompositions as one group, and only use the full-merged form to represent the group. On the above example, we only use $\{a,b,c\}$ to represent the group including three associative sub-decompositions $\{a,b\}$, $\{b,c\}$, $\{a,c\}$. After merging such associative ones, two simple disjoint decompositions never overlap each other. The structure of simple disjoint decompositions can be expressed by a tree graph as shown in Fig. 2. Since each input variable appears only once as a leaf of tree, the number of branching nodes never exceeds the number of input variables. The problem of finding simple disjoint decompositions is to construct such a tree structure for a given Boolean function.

There are many studies on the methods of finding simple disjoint decompositions. At first, a classical method with a *decomposition chart* is presented[3][17]. In recent years, more efficient way using a BDD-based implicit decomposition chart is discussed[9][18][19]. One proposed another approach[13] to extract all simple disjoint decompositions based on factoring of sum-of-products expressions.

In the long history of studies on Boolean function decomposition, the BDD-based recursive algorithm, which is proposed by Bertacco et al. on 1997 and improved later by Matsunaga[10][11], is now overwhelmingly effective to extract all simple disjoint decompositions for a given Boolean functions. This algorithms is based on the following two properties:

- If we consider NPN-equivalence and associativity, the tree structure of simple disjoint decompositions is canonical for a given Boolean functions.
- Basically, all simple disjoint decompositions for f can also be found in the two cofactor functions $f_{(x=0)}$ and $f_{(x=1)}$.

Using the two properties, the algorithm expands a given Boolean function to the two cofactor functions and call itself recursively. The final results of tree structure is obtained by checking common part of the results for the two cofactor functions. Since the algorithm has a cache mechanism to avoid duplicated traversals for the same BDD nodes, we can efficiently execute the procedure in a time bounded by the BDD size (roughly square for BDD nodes). Actually, we need only a few seconds to extract all possible simple disjoint decompositions for a benchmark function with ten thousands of BDD nodes and dozens of input variables. This decomposition method is effectively used for VLSI logic synthesis and technology mapping[16].



a	b	c	F	
0	0	0	0	
0	0	1	1	$\rightarrow c$
0	1	0	0	
0	1	1	1	$\rightarrow bc$
1	0	0	0	
1	0	1	0	
1	1	0	1	$\rightarrow ab$
1	1	1	1	$\rightarrow abc$

As a Boolean function:
$F = a\,b + \overline{a}\,c$

As a set of combinations:
$F = \{ab, abc, bc, c\}$

Figure 3: Correspondence of Boolean functions and Sets of combinations.

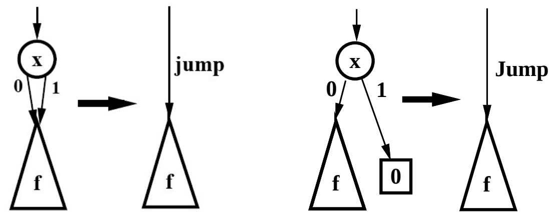
3 Decomposition for Sets of Combinations

In this section, we discuss simple disjoint decomposition for sets of combinations, and show a method of finding those decompositions.

3.1 Sets of combinations and Boolean functions

A set of combinations consists of the elements each of which is a combination of a number of items. There are 2^n combinations chosen from n items, so we have 2^{2^n} variations of sets of combinations. For example, for a domain of five items a, b, c, d, and e, we can show examples of sets of combinations as: $\{ab, e\}$, $\{abc, cde, bd, acde, e\}$, $\{1, cd\}$, $\emptyset$. Here "1" denotes a combination of null items, and "$\emptyset$" means an empty set. Sets of combinations are one of the basic data structure for handling combinatorial problems. They often appear in real-life problems, such as combinations of switching devices, sets of faults, paths in the networks, etc., and of course, it can be used for representing frequent itemset data.

A set of combinations can be mapped into Boolean space of n input variables. For example, Fig. 3 shows a truth table of Boolean function $(ab + \overline{a}c)$, but also represents a set of combinations $\{ab, abc, bc, c\}$. Such Boolean functions are called *characteristic functions* for the sets of combinations. Using BDD manipulation for characteristic functions, we can implicitly represent and



(a) Ordinary BDD. (b) ZBDD.

Figure 4: Different reduction rules for BDD and ZBDDs.

manipulate large-scale sets of combinations. In addition, we can enjoy more efficient manipulation using "Zero-suppressed BDDs" (ZBDD)[12], which are special type of BDDs optimized for handling sets of combinations.

ZBDDs are based on the reduction rule different from one used in ordinary BDDs. As illustrated in Fig. 4(a), the ordinary reduction rule deletes the nodes whose two edges point to the same node. However, in ZBDDs, we do not delete such nodes but delete another type of nodes whose 1-edge directly points to the 0-terminal node, as shown in Fig. 4(b).

In ZBDDs, a 0-edge points to the subset (cofactor) of combinations not including the decision variable x, and a 1-edges points to the subset (cofactor) of combinations including x. If the 1-edge directly points to the 0-terminal node, it means that the item x never appear in the set of combinations. Zero-suppressed reduction rule automatically deletes such a node with respect to the irrelevant item x, and thus ZBDDs are more compactly represent sets of combinations than using ordinary BDDs.

The detailed techniques of ZBDD manipulation are described in the articles[12][14]. A typical ZBDD package supports cofactoring operations to traverse 0-edge or 1-edge, and binary operations between two sets of combinations, such as union, intersection, and difference. the computation time for each operation is almost linear to the number of ZBDD nodes related to the operation.

3.2 Simple disjoint decompositions on sets of combinations

In this paper, we propose the definition of "simple disjoint decomposition on sets of combinations", as a similar concept as one for Boolean functions. As shown in Fig. 1 again, if a given sets of combination 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. Here we explain the meaning of substitution operation on the set of combinations. At first, we consider the set of combination $g(s,Y)$. Let us extract all combinations including s (a cofactor of g w.r.t s), and then replace s with any one combination in $h(X)$. The result of substitution $g(h(X),Y)$ is the set of all possible combinations obtained by the replacements. Notice that the combinations irrelevant to s are left as is. If the substitution result exactly equals to $f(X,Y)$, it is a decomposition on f.

For instance, we consider the two sets of combinations:

$g(c,d,e) = \{cd, cde, de, e\}$ and $h(a,b) = \{a, ab\}$.

When we substitute $h(a,b)$ for c, then

$g(h(a,b),d,e) = \{ad, abd, ade, abde, de, e\}$.

Namely, $f(a,b,d,e) = \{ad, abd, ade, abde, de, e\}$ has a simple disjoint decomposition as $g(h(a,b),d,e)$.

A set of combinations has one-to-one mapping to a Boolean function, however, the simple disjoint decompositions for the two data models do not have such direct relation. Although simple disjoint decomposition on sets of combinations is a similar concept as one for Boolean functions, we have to develop another decomposition algorithm considering particular properties on sets of combinations.

Similarly to the case for Boolean functions, a set of combinations may have more than one simple disjoint decompositions. For example, the set of combinations $\{abcx, abcy, abcz, xy, xz\}$ has five decompositions as $X = \{a,b\}$, $\{b,c\}$, $\{a,c\}$, $\{a,b,c\}$, and $\{y,z\}$. They may have a nested structure, and can be represented by a tree graph as shown in Fig. 5.

Contrary to the case for Boolean functions, sets of combinations do not have symmetric NOT operation (uses difference operation instead of NOT),

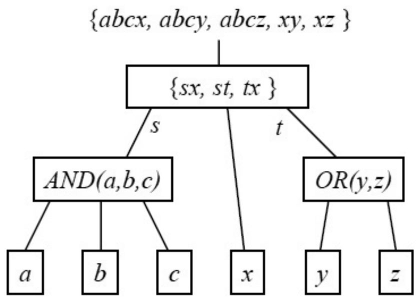


Figure 5: Tree structure of decompositions on sets of combinations.

there are no equivalences on input/output polarity. Only input permutation should be considered. In Matsunaga's decomposition algorithm[11] on Boolean functions uses only OR and NOT instead of AND operation to keep canonical forms, however, in the case for sets of combinations, AND (product) operation and OR (union) operation do not have dual relation, so we should treat them individually. We do not have to consider XOR operation on sets of combinations. Finally we found two kind of binary operations: AND (product) and OR (union), which may produce associative sub-decompositions on sets of combinations. As well as for Boolean functions, we handle those sub-decompositions as one group, and only use the full-merged form to represent the group.

We need one more special consideration when the set of combinations includes "1" (the element of null combination). For example, let us consider $f(a, b, c) = \{ab, c, 1\}$. If we choose $h = \{ab\}$, we can find simple disjoint decomposition as $g(h, c) = \{h, c, 1\}$. However, we may choose $h = \{ab, 1\}$ and then $g(h, c) = \{h, c\}$ or $g(h, c) = \{h, c, 1\}$ are possible. Consequently, the decomposition structure may not be unique when the set of combinations includes "1". In such cases, we need additional rule to keep the decomposition trees canonical. Basically, we may put a restriction that $h(X)$ does not include "1", namely, only the parent set $g(s, Y)$ can have "1". This rule is working well except for AND (product) operation. Let us consider $f = \{abc, ac\}$. It can be decomposed as $\{ab, b\} \times \{c\}$, $\{a\} \times \{bc, c\}$, $\{ac\} \times \{b, 1\}$, and $\{a\} \times \{b, 1\} \times \{c\}$. This example shows that AND (product) operation has the associative property no matter how "1" is included or not, so we must decompose by a set with "1" if it is possible. Fortunately, we can see that if f is AND-decomposable by h with "1", then f is not decomposable by h without "1". On the other hand, if f is AND-decomposable by h without "1", then f is not decomposable by h with "1". This means that we can keep the decomposition tree unique.

3.3 ZBDD-based algorithm for finding simple disjoint decompositions

In the following discussion, we use "$\cdot$" for AND (product) operator, and "+" for OR (union) operator.

If a given sets of combinations $F(X, Y)$ contains a simple disjoint decomposition with $P(X)$, it can be written as:

$$F(X, Y) = P(X) \cdot Q(Y) + R(Y).$$

Here we choose an item v used in F, and compute the two cofactors F_0, F_1. (Namely, $F = v \cdot F_1 + F_0$.) Since v must be included in either of X or Y, the following two cases are considered:

- In case of $v \in X$: (Let $X' = X - v$.)
  $F(X, Y) = P(X) \cdot Q(Y) + R(Y)$
  $= \{v \cdot P_1(X') + P_0(X')\} \cdot Q(Y) + R(Y)$
  $= v \cdot \{P_1(X') \cdot Q(Y)\} + \{P_0(X') \cdot Q(Y) + R(Y)\}$
  Thus,
  $F_1(X', Y) = P_1(X') \cdot Q(Y)$,
  $F_0(X', Y) = P_0(X') \cdot Q(Y) + R(Y)$.

- In case of $v \in Y$: (Let $Y' = Y - v$.)
  $F(X, Y) = P(X) \cdot Q(Y) + R(Y)$
  $= P(X) \cdot \{v \cdot Q_1(X') + Q_0(X')\}$
  $+\{v \cdot R_1(X') + R_0(X')\}$
  $= v \cdot \{P(X) \cdot Q_1(Y') + R_1(Y')\}$
  $+\{P(X) \cdot Q_0(Y') + R_0(Y')\}$
  Thus,
  $F_1(X, Y') = P(X) \cdot Q_1(Y') + R_1(Y')$,
  $F_0(X, Y') = P(X) \cdot Q_0(Y') + R_0(Y')$.

In any case, if a given sets of combination F has simple disjoint decompositions, they can be found by checking the common set of decompositions on the cofactors F_0 and F_1. Similarly to BDD-based

```
Decomp($F$)
{
  if ($F = 0$) return 0 ;
  if ($F = 1$) return 1 ;
  $H \leftarrow$ Cache($F$) ;
  if ($H$ exists) return $H$ ;
  $v \leftarrow$ F.top ; /* Top item in $F$ */
  $(F_0, F_1) \leftarrow$ (Cofactors of $F$ w.r.t $v$) ;
  $H_0 \leftarrow$Decomp($F_0$) ;
  $H_1 \leftarrow$Decomp($F_1$) ;
  $H \leftarrow$Merge($v, H_0, H_1$) ;
    /* Make $H$ from common part of $H_0$, $H_1$ */
  Cache($F$) $\leftarrow H$ ;
  return $H$ ;
}
```

Figure 6: Sketch of the algorithm.

recursive method of Boolean function decomposition, there are some cases that a part of factors, such as $P_0(X')$ and $Q_0(Y')$, may be reduced to a constant "1" or "$\emptyset$", so we need more detailed classification in actual implementation of the algorithm.

The above discussion shows that we can generate a tree structure of representing all possible simple disjoint decompositions on a set of combinations using a ZBDD-based recursive algorithm, which is a similar manner to the BDD-based method[4][10][11] for Boolean function decomposition. The sketch of algorithm is shown in Fig.6. The computation time can be roughly square to the ZBDD size as well as the previous method, by using a cache mechanism to avoid duplicated traversals for the same ZBDD nodes.

4 Experimental Results

We implemented a program for finding all simple disjoint decompositions on sets of combinations. The program is based on our own ZBDD package, and additional 1,600 lines of C++ code for the decomposition algorithm. We used a Pentium-4 PC, 800MHz, 512MB of main memory, with SuSE Linux 9. We can manipulate up to 10,000,000 nodes of ZBDDs in this PC.

For evaluating the performance, we conducted an experiment for finding all simple disjoint decompositions in th frequent itemset data extracted from two example of benchmark database[7]. The specification of the two databases are shown in Table 1. In this table, the column "#Items" shows the number of items used in the database, "#Records" is the number of records, "Total literals" means the total number of each record size. (Record size is the number of items appearing in the record.) "Literals/Records" shows average number of items appearing in a record.

In our experiment, at first we generate a ZBDD representing the histogram of all patterns seen in the database, and then we extract a set of frequent patterns that appear more than or equal to α times in the database. We conducted this frequent itemset mining procedure for various threshold α. In this process, we need about 300 second for "mushroom", and 6 second for the first 1,000 lines of "T10I4D100K", respectively. The detailed techniques in the ZBDD-based frequent itemset mining method are described in a recent articles[15].

The experimental results are summarized in Table 2. In the table, the column "Min-freq.(α)" shows the minimum frequency parameter α. "#Items" means the number of items related to the frequent itemset. "#Patterns" is the number of patterns in the frequent itemsets. "ZBDD nodes" shows the number of ZBDD nodes representing the frequent itemset data, which is the input of the decomposition algorithm. "Time(sec)" shows the CPU time for generating the decomposition tree representing all simple disjoint decompositions. (not including the time for generating initial ZBDDs.) "#Decomp." is the number of non-trivial decompositions extracted by our method. Notice that all sub-decompositions caused by associative operations are counted only once for one group, so, actual number of decompositions may exist more.

As shown in the table, our decomposition method is very powerful to deal with a huge number of frequent patterns. We succeeded in finding simple disjoint decompositions in the frequent itemset data in a feasible time. It is another interesting point that the decomposition results are different depending on threshold α.

In Fig. 7, we show an example of decomposition result for "mushroom" with threshold $\alpha = 5,000$.

Table 1: Spec. of databases

Data name	#Items	#Records	Total Literals	Literals/Records
mushroom	119	8,124	186,852	23.0
T10I4D100K (first 1000 lines)	795	1,000	10,098	10.1

Table 2: Experimental result

Data name	Min-freq.(α)	#Items	#Patterns	ZBDD nodes	Time(sec)	#Decomp.
mushroom	5,000	7	42	11	(<0.1)	5
	2,000	35	6,624	286	(<0.1)	4
	1,000	54	123,278	1,417	0.1	4
	500	67	1,442,504	4,011	0.2	5
	200	83	18,094,822	12,340	0.5	6
	100	90	66,076,586	23,068	0.8	7
	50	98	198,169,866	36,652	2.0	7
	20	113	781,458,546	53,776	4.0	8
	10	115	1,662,769,668	61,240	4.9	9
	5	117	2,844,545,896	62,389	11.7	10
	2	119	5,043,140,094	51,217	7.1	10
	1	119	5,574,930,438	40,557	5.0	10
T10I4D100K	70	1	2	1	(<0.1)	1
(first 1000 lines)	60	4	5	4	(<0.1)	1
	50	12	13	12	(<0.1)	1
	30	74	75	74	(<0.1)	1
	20	171	173	172	0.1	2
	10	378	506	430	0.8	22
	5	585	3,891	1,322	1.5	42
	2	745	30,893	9,903	13.3	13
	1	795	24,467,220	70,847	1,698.0	8

List of all frequent patterns:

```
x39 x86 x85 x34, x39 x86 x85, x39 x86 x34, x39 x86, x39 x85 x34, x39 x85, x39 x34, x39,
x90 x86 x85 x36 x34, x90 x86 x85 x36, x90 x86 x85 x34, x90 x86 x85, x90 x86 x36 x34,
x90 x86 x36, x90 x86 x34, x90 x86, x90 x85 x36 x34, x90 x85 x36, x90 x85 x34, x90 x85,
x90 x36 x34, x90 x36, x90 x34, x90, x86 x85 x36 x34, x86 x85 x36, x86 x85 x34, x86 x85,
x86 x36 x34, x86 x36, x86 x34, x86, x85 x59, x85 x36 x34, x85 x36, x85 x34, x85, x59,
x36 x34, x36, x34, 1
```

Decomposition result:

```
AND(OR(AND(OR(x39 AND(!x90 !x36)) !x86 !x34) x59) !x85)
```

Figure 7: Decomposition result for “mushroom” with $\alpha = 5,000$.

The upper description lists all frequent patterns of itemsets, and the lower is the decomposition result. In this format, "`AND( )`" and "`OR( )`" show the associative decomposition groups, and "`[ ]`" means another general decomposition. "`!`" is a special character to mean "+1". For example, "`AND(!x !y)`" indicates $(x+1)(y+1)$. Even for this small example, we can see that it is hard for human beings to find the hidden structures from the plain list. By using our decomposition method, the data becomes remarkably understandable.

In Fig. 8 and Fig. 9, we show the decomposition results for "mushroom" and "T10I4D100K" with various threshold α. We can observe interesting structures hidden in the frequent itemset data. Notice that the frequent itemset data handled here are too large to manipulate explicit manner. ZBDD-based implicit manipulation is a key technique

5 Conclusion

In this paper, we proposed the definition of simple disjoint decomposition for sets of combinations, and presented an efficient ZBDD-based method for finding all possible simple disjoint decompositions on a set of combinations. The experimental results shows that our method is very powerful and useful for extracting hidden interesting structures from the frequent itemset data.

Now we have just finished implementation of decomposition algorithms, and starting experiments for data mining applications. The concept of simple disjoint decomposition will be a meaningful operation in database processing, and we hope that our result has an impact to the data mining community.

Acknowledgment

The author thanks Prof. Arimura of Hokkaido Univ. for his technical comments. This study is partly supported by Grant-in-Aid Scientific Research on Priority Area "Informatics", 2004 (Area #006) and Scientific Research (B), 2005, 17300041.

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] R. L. Ashenhurst, "The decomposition of switching functions," Proc. of an International Symposium on the Theory of Switching, pp. 74–116, 1957.

[4] V. Bertacco and M. Damiani, "The disjunctive decomposition of logic functions," In Proc. of International Conference on Computer-Aided Design Design (ICCAD-97), pp. 78–82, Nov.1997.

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

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

[7] B. Goethals, M. Javeed Zaki (Eds.), Frequent Itemset Mining Dataset Repository, Frequent Itemset Mining Implementations (FIMI'03), 2003. `http://fimi.cs.helsinki.fi/data/`

[8] 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.

[9] Y.-T. Lai, M. Pedram and S. B. K. Vrudhula, "BDD based decomposition of logic functions with application to FPGA synthesis," In Proc. of 30th ACM/IEEE Design Automation Conf. (DAC-93), pp. 642–647, 1993.

[10] Y. Matsunaga, "An exact and efficient algorithm for disjunctive decomposition," In Proc. of the Workshop on Synthesis And System Integration of Mixed Technologies (SASIMI-98), pp. 44–50, 1998.

[11] Y. Matsunaga, "An efficient algorithm finding simple disjoint decompositions using BDDs," IEICE Trans. on fund., Vol. E-85-A, No. 12, pp. 2715–2724, 2002.

[12] Minato, S., 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 and G. De Micheli, "Finding All Simple Disjunctive Decompositions Using Irredundant Sum-of-Products Forms," In Proc. of ACM/IEEE International Conference on Computer-Aided Design (ICCAD-98), pp. 111–117, Nov. 1998.

[14] Minato, S., 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", IEEE/IEICE/IPSJ International Workshop on Challenges in Web Information Retrieval and Integration (WIRI-2005), pp. 3–10, Apr., 2005.

[16] A. Mishchenko, X. Wang, and T. Kam, "A New Enhanced Constructive Decomposition and Mapping Algorithm," In Proc. of 40th ACM/IEEE Design Automation Conf. (DAC-2003), pp. 143–148, 2003.

[17] J. Roth and R. Karp, "Minimization Over Boolean Graphs," IBM Journal, pp. 227-238, Apr. 1962.

[18] T. Sasao, "FPGA Design by Generalized Functional Decomposition," Logic Synthesis and Optimization, Kluwer Academic Publishers, pp. 233–258, 1993.

[19] H. Sawada and T. Suyama and A. Nagoya, "Logic Synthesis for Look-Up Table based FPGAs Using Functional Decomposition and Boolean Resubstitution," IEICE Trans. Information and Systems, Vol. E80-D, No. 10, pp. 1017-1023, Oct. 1997.

[20] T. Uno, T. Asai, Y. Uchida, H. Arimura, "LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets," In Proc. IEEE ICDM'03 Workshop FIMI'03, 2003. (Available as CEUR Workshop Proc. series, Vol. 90, http://ceur-ws.org/vol-90)

[21] T. Uno, T. Asai, Y. Uchida and H. Arimura, "An Efficient Algorithm for Enumerating Closed Patterns in Transaction Databases," In Proc. of the 8th International Conference on Discovery Science 2004 (DS-2004), 2004.

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

[23] M. J.Zaki, C. Hsiao, "CHARM: An Efficient Algorithm for Closed Itemset Mining," 2nd SIAM International Conference on Data Mining (SDM'02), pp. 457-473, 2002.

$\alpha = 2,000$:

```
AND(![x102 x58 OR(x66 x61) x29 x116 x56 x6 x11 x110 x94 x53 x28 x24 x10 x114 x39 x2 x93
x90 AND(!x86 !x34) x76 x67 x63 x59 x52 x38 x36 x23 x13 x9 x3 x1] !x85)
```

$\alpha = 1,000$:

```
AND(![x48 x102 x58 AND(!x101 !x95) x66 x61 x29 x17 OR(x69 x77) x117 x116 x56 x6 x111 x11
x44 x110 x43 x94 x53 x37 x28 x24 x16 x10 x41 x15 x114 x99 x39 x14 x2 x107 x98 x93 x90 x86
x76 x67 x63 x59 x54 x52 x38 x36 x34 x23 x13 x9 x3 x1] !x85)
```

$\alpha = 500$:

```
AND(![OR(x32 x7 x31) x119 x48 x102 x91 x58 x80 x101 x95 x66 x61 x29 x17 OR(x78 x68) x69
x77 x45 OR(x60 x64) x117 x116 x56 x6 x111 x11 x44 x110 x43 x42 x94 x53 x37 x28 x24 x16
x10 x41 x15 x114 x99 x55 x39 x14 x2 x107 x98 x93 x90 x86 x76 x67 x63 x59 x54 x52 x38 x36
x34 x23 x13 x9 x3 x1] !x85)
```

$\alpha = 200$:

```
AND(![x35 OR(x32 x31) x7 x119 x48 x112 x102 x91 x58 OR(x80 x71 x79 x70) x101 x95 x66 x61
x29 x17 x46 OR(x78 x68) x69 x77 x45 x60 x117 x65 x116 x56 x6 x111 x64 x11 x44 x110 x43
x42 x109 x94 x53 x37 x28 x24 x16 x10 x115 x41 OR(x27 x26) x15 x4 x114 x108 x99 x55 x39
x14 x2 x113 x107 x98 x93 x90 x86 x76 x67 x63 x59 x54 x52 x40 x38 x36 x34 x25 x23 x13 x9
x3 x1] !x85)
```

Figure 8: Decomposition results for “mushroom” with various α.

$\alpha = 60$:

```
!OR(x354 x177 x217 x368)
```

$\alpha = 50$:

```
!OR(x438 x460 x829 x684 x354 x177 x217 x529 x766 x283 x120 x368)
```

$\alpha = 20$:

```
!OR(x893 x634 x661 x826 x510 x886 x38 x440 x75 x438 x998 x752 x812 x57 x694 x692 x982
x413 x8 x349 x912 x906 x162 x275 x55 x470 x598 x793 x989 x33 x995 x509 x797 x69 x516
x593 x236 x93 x140 x112 x48 x348 x239 x32 x21 x132 x10 x72 x871 x832 x580 x168 x154
x573 x472 x31 x362 x494 x489 x196 x919 x744 x28 x600 x460 x829 x605 x477 AND(!x346
!x217) x684 x354 x296 x12 x617 x522 x885 x676 x145 x956 x758 x935 x780 x638 x631 x487
x541 x116 x789 x788 x526 x403 x918 x844 x722 x419 x310 x73 x151 x874 x480 x204 x70 x43
x944 x888 x614 x523 x803 x778 x579 x411 x147 x921 x78 x27 x862 x392 x960 x795 x623 x571
x177 x175 x161 x878 x653 x276 x183 x914 x720 x675 x597 x280 x279 x192 x71 x390 x970
x947 x809 x782 x682 x658 x529 x350 x798 x569 x966 x883 x766 x738 x381 x283 x229 x937
x895 x449 x854 x674 x581 x401 x205 x120 x39 x825 x775 x561 x538 x368 x274)
```

$\alpha = 10$:

```
OR(x207 x820 x732 x769 x550 x428 x450 x258 x173 x922 x893 x949 x405 x351 x215 x634 [x661
x394 x510 AND(!x515 !x33) x346 x780 x487 AND(!x888 !x561) x217 x720 x71 x766 x283] x308
x948 x815 x838 x707 x826 x804 x309 x887 x318 x860 x68 x241 x129 x843 x429 x886 [x819 x75
x438 AND(!x598 !x782) x460 x829 x684 x789 x70 x529 x937 x368] x468 x686 x265 x784 x252 x38
x440 x486 x108 x322 x361 [x357 AND(!x752 !x58) AND(!x158 !x617) x583 x354 x480 x27] x563
x170 x867 x710 x595 x899 x998 x991 x852 x160 x923 x812 x800 x57 x567 x694 x692 x984 x982
x94 x413 x197 x8 x349 x963 x665 x546 x406 x373 x117 AND(!x912 !x348) x906 x711 AND(!x534
!x470 !x995) x378 x162 x45 x259 x275 x897 x95 x55 x37 x427 x793 x97 x989 x336 x527 x343
x983 x611 [x509 x862 x801 x461 x392 x569] x110 x577 x913 x797 x415 x69 x6 AND(!x516 !x744)
x423 x1 x122 x593 x90 x952 x236 x606 x93 x594 x387 x285 x140 x112 x521 x765 x639 x319 x126
x48 x500 x100 x239 x136 x54 x32 x21 x464 x172 x132 x10 x981 x72 x988 x871 x832 x580 x168
x154 x111 x651 x628 x573 x472 x329 x181 x31 x591 x362 x673 x641 x494 x489 x196 x919 x736
AND(!x517 !x883) x115 x5 x742 x28 AND(!OR(x709 x177 x970) !x310) x600 x746 x649 x355 x234
x884 x605 x477 x210 x841 x740 x548 x296 x12 x522 x885 x792 x790 x731 x676 x385 x145
AND(!x956 !x788) x763 x758 x242 x17 x935 x735 x638 x631 x471 x946 x805 x701 x541 x171 x395
x201 x198 x116 x975 x774 x526 x403 x326 x967 x918 x846 x844 x810 x722 x484 x469 x419 x118
x73 x890 x830 x504 x151 x874 x334 x204 x43 x944 x614 x523 x290 x266 x903 x842 x803 x778
x579 x572 x411 x147 x921 x78 x839 x130 x125 x960 x910 AND(OR(x795 AND(!x623 !x853)) !x571)
x490 x424 x175 x161 AND(!x878 !x538) x706 x653 x277 x276 x256 x193 x183 x932 x914 x675 x618
x597 x530 x496 x280 x279 x272 x192 x390 x227 x947 x809 x682 x658 x350 x214 x798 x620 x143
x104 x978 x966 x738 x708 x381 x352 x294 x229 x964 x895 x857 x449 x422 x950 x854 x733 x674 x35
x814 x704 x581 x401 x205 x120 AND(!x39 !x825) x834 x775 x687 x448 x274 x240 x164 x52 x25)
```

Figure 9: Decomposition results for “T10I4D100K” (first 1000 lines) with various α.