TCS-TR-A-05-03

TCS Technical Report

VSOP (Valued-Sum-Of-Products) Calculator Based on Zero-Suppressed BDDs

by

Shin-ichi Minato

Division of Computer Science

Report Series A

May 17, 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

VSOP (Valued-Sum-Of-Products) Calculator Based on Zero-Suppressed BDDs

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

May 17, 2005

(Abstract) Recently, Binary Decision Diagrams (BDDs) are widely used for efficiently manipulating large-scale Boolean function data. BDDs are also applied for handling combinatorial item set data. Zero-suppressed BDDs (ZBDDs) are special type of BDDs which are suitable for implicitly handling large-scale combinatorial item set data. In this paper, we present *VSOP* program developed for calculating combinatorial item set data specified by symbolic expressions based on ZBDD techniques. Our program supports not only combinatorial set operations but also numerical arithmetic operations based on *Valued-Sum-Of-Products* algebra, such as addition, subtraction, multiplication, division, numerical comparison, etc. We discuss the data structures and algorithms in our program, and show some typical applications. VSOP calculator will be useful for solving many problems in Computer Science.

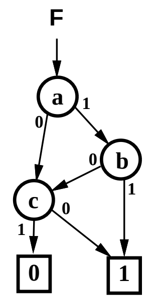
1 Introduction

Manipulation of Boolean functions is a fundamental techniques for handling various problems in computer science. Binary Decision Diagrams(BDDs)[4] are efficient graph-based representation of Boolean functions, intensively studied in 1990's, and now widely used in digital system design and many other areas. Zero-suppressed BDDs (ZBDD)[10, 15] are a special type of BDDs for efficient manipulation of combinatorial item set data. ZBDD-based method have been applied for many algorithmic problems such as minimizing sum-of-products forms[14] , database analysis[16], and many kinds of graph optimization problems[6].

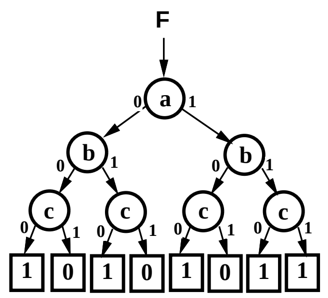
In this paper, we present *VSOP* calculator developed for calculating combinatorial item sets specified by symbolic expressions. Based on ZBDD techniques, VSOP can efficiently handle large-scale sum-of-products expressions with a number of item symbols. Our program supports not only Boolean set operations but also numerical arithmetic operations based on *Valued-Sum-Of-Products* algebra, such as addition, subtraction, multiplication, division, numerical comparison, etc.

The author has a past result of developing an arithmetic Boolean expression manipulator "BEM-II"[9] based on (ordinary) BDDs, and the program was utilized for many works[17, 18, 8]. VSOP deal with the arithmetic and numerical operations as well as BEM-II, and extends the data model from Boolean functions to combinatorial sets. The interface of VSOP is very flexible and customizable for solving many kinds of combinatorial problems, and it will facilitate research and development for knowledge processing.

This paper is organized as follows: First, we briefly review BDDs and ZBDDs in Section 2. We then describe the representation method of Valued-Sum-Of-Products forms based on ZBDDs in Section 3. We present algorithms of arithmetic operations in Section 4, and several display formats of VSOP are shown in Section 5. Finally we show

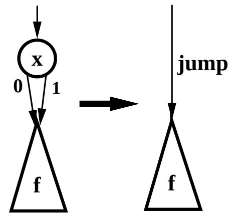


(a) BDD.

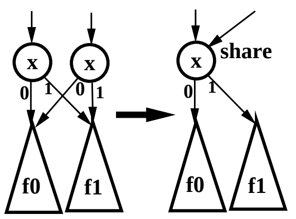


(b) Binary Decision Tree.

Figure 1: Reduced and non-reduced BDDs for $F = (a \wedge b) \vee \overline{c}$.



(a) Node deletion.



(b) Node sharing.

Figure 2: Conventional BDD reduction rules.

some typical applications of VSOP calculator followed by concluding remarks.

2 BDDs and ZBDDs

BDD is a directed graph representation of the Boolean function, as illustrated in Fig. 1(a). It is derived by reducing 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 research on BDDs are based on the above reduction

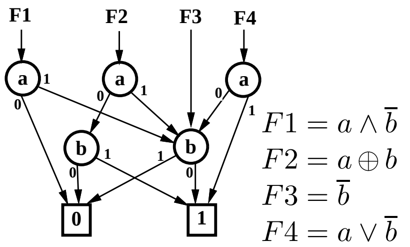


Figure 3: Shared BDD.

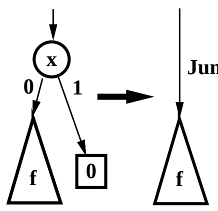


Figure 4: ZBDD reduction rule.

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 including 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.)

BDDs are originally developed for handling Boolean function data, however, they can also be used for implicit representation of combinatorial sets. Here we call "combinatorial item set" 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, Boolean item sets in the database, and combinatorial sets of edges or nodes in the graph data model.

A combinatorial item set can be mapped into Boolean space of n input variables. If we choose any one combination of items, a Boolean function determines whether the combination is included in the combinatorial item set. Such Boolean functions are called *characteristic functions*. The set operations such as union, intersection, and difference can

$S(abc)$:

abc	S
000	0
100	1
010	1
110	0
001	0
101	0
011	0
111	0

$S(abcd)$:

abcd	S
0000	0
1000	1
0100	1
1100	0
0010	0
1010	0
0110	0
1110	0
0001	0
1001	0
0101	0
1101	0
0011	0
1011	0
0111	0
1111	0

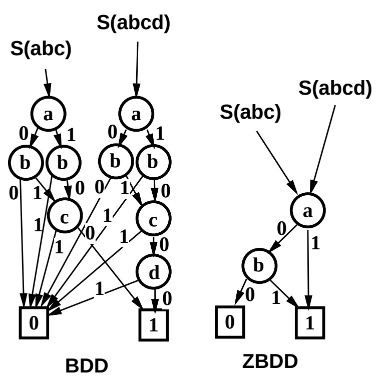


Figure 5: Effect of ZBDDs.

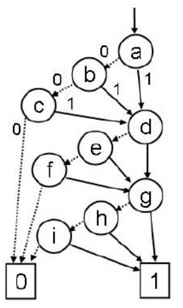


Figure 6: ZBDD for $(a+b+c)(d+e+f)(g+h+i)$

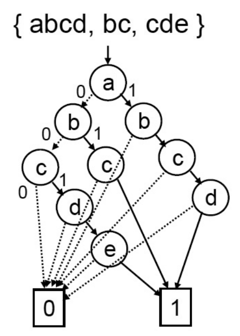


Figure 7: Explicit representation with ZBDD.

be performed by logic operations on characteristic functions.

By using BDDs for characteristic functions, we can manipulate combinatorial item set 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)[10, 15] is a special type of BDDs for efficient manipulation of combinatorial item set. 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 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.

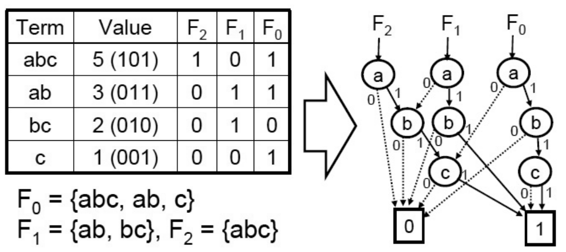
Here we summarise the features of ZBDDs.

- 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. 5.)
- 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. Namely, (the order of) ZBDD size never exceeds the explicit representation. An example is shown in Fig. 7. If more 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.

Figure 8 shows the most of primitive operations of ZBDDs. In these operations, $\emptyset$, $\mathbf{1}$, *P.top* are executed in a constant time, and the others are almost linear to the size of graph. We can describe

"$\emptyset$"	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.

Figure 8: Primitive ZBDD operations



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

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

Figure 9: ZBDD vector for ($5abc+3ab+2bc+c$).

various processing on combinatorial item sets by composing of these primitive operations.

3 VSOP Expressions Using ZBDDs

In this paper, we call VSOP (Valued-Sum-Of-Products) for a combinatorial item set (or a sum-of-products form) such that each product term has a value. This value can also be considered as a coefficient or a weight for each term. So far, we deal with only integer values. We define the value as zero for a product term not included in the VSOP.

For example, the formula ($5abc + 3ab + 2bc + c$) represents a VSOP with four terms abc, ab, bc, and c, each of which is valued as $5, 3, 2$, and 1, respectively. This meas that a pattern abc appears five times in a same database. Another meaning is that five times cost is needed to obtain a pattern abc in a certain process.

Not only enumerating combinations but also assigning such values (coefficients or weights) for each product term, we can represent a simple but fundamental knowledge data, which can be used for various problems in computer science. That is a motivation for us to develop a program to efficiently calculate VSOP expressions based on ZBDD techniques.

In the VSOP algebra, the addition follows the ordinary rule: $1 + 1 = 2$ and $x + x = 2x$. However, multiplication rule is not conventional: $2 \times 2 = 4$, $x \times y = xy$, but $x \times x = x$, because we only handle combinatorial item sets, not considering higher degree of item symbols. Notice that the same algebra is also used in calculating expressions of probabilistic variables.

Here we discuss the way to compactly represent VSOP data by using ZBDDs. Since ZBDDs are representation of combinatorial item sets, a simple ZBDD distinguishes only existence of each product term in the set. Thus we need some extended data structure to represent numerical numbers using ZBDDs. Two methods are known on this issue, the one is using BMDs (Binary Moment Diagrams)[5] handling not only 0- and 1-terminal nodes but also numerical valued terminal nodes. The other method is using vector of ordinary ZBDDs to represent binary coding of numerical values[12]. In our program, we use the latter method. We decompose the integer 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, F_0 represents a set of terms whose values are odd numbers, F_1 represents a set of terms whose values have '1' at the second digit of binary coding, and listing such ZBDDs until F_{m-1}, we can implicitly represent a VSOP data.

When dealing with integer values in binary coding, we have to consider the expression of negative numbers. There are two well-known methods, one of which is using 2's complement representation, and the other is using the absolute value with sign; however, both method have drawbacks. When using 2's complement, it yields many non-zero digits for small negative numbers (typically, -1 is "all one”), and the ZBDD reduction rule is not effective to those non-zero bits. On the other hand, when using absolute value, the operation of addition become complicated since we have to classify the product terms to choose addition or subtraction depending on magnitude of values.

To solve the above problems, we adopted another binary coding[12] based on (-2), namely, each bit represents $1, -2, 4, -8, 16, \ldots$. For example, -12 can be decomposed into $(-2)^5 + (-2)^4 + (-2)^2 = -2 \cdot 2^4 + 2^4 + 2^2$. In this encoding we can also uniquely represent the integer numbers. Using binary coding with (-2), the higher digits become zero both for positive and negative numbers, and the ZBDD reduction rule works effectively to eliminate the meaningless nodes of higher digits.

In our implementation, we define the special item symbols to combine the ZBDD vector into a single ZBDD. By using 20 special item symbols with higher variable order (near to the root node), up to 2^{20} (about one million) digits of ZBDD vector can be combined into one ZBDD. This means that practically unlimited long digital numbers are representable in our program.

4 Algorithms for Arithmetic Operations

VSOP expressions are manipulated by arithmetic operations, such as addition, subtraction, multiplication, and division. We first generate ZBDDs for trivial VSOP expressions which are single item symbols or integer constants, and then apply those arithmetic operations to construct more complicated VSOP expressions. In this section, we present efficient algorithms for the arithmetic operations of VSOPs based on ZBDDs.

(Multiplication by an item) Multiplication of a VSOP F and an item symbol v can be done by simply attach v to all product terms in F. This is easily written by the basic operations *Onset, Offset*, and *Change* of ZBDDs. Computation time is linear to the number of nodes which are ordered lower than v in the ZBDD.

```
procedure $(F + G)$
{ $C \leftarrow (F \cap G)$ ;
  $S \leftarrow (F \cup G) - C$ ;
  if $(C = 0)$ return $S$ ;
  else return $S - (-2 \cdot C)$;
}

procedure $(F - G)$
{ $B \leftarrow (\overline{F} \cap G)$ ;
  $D \leftarrow (F \cap \overline{G})$ ;
  if $(B = 0)$ return $D$ ;
  else return $D + (-2 \cdot B)$;
}
```

Figure 10: Algorithm for addition and subtraction.

```
procedure$(F \times G)$
{ if $(F.top < G.top)$ return $(G \times F)$ ;
   if $(G = 0)$ return 0 ;
   if $(G = 1)$ return $F$ ;
   $H \leftarrow$ cache(“$F \times G$”) ;
   if ($H$ exists) return $H$ ;
   $v \leftarrow F.top$ ; /* the highest item in $F$ */
   $(F_0, F_1) \leftarrow$ factors of $F$ by $v$ ;
   $(G_0, G_1) \leftarrow$ factors of $G$ by $v$ ;
   $H \leftarrow ((F_1 \times G_1) + (F_1 \times G_0) + (F_0 \times G_1)) \times v$
       $+ (F_0 \times G_0)$ ;
   cache(“$F \times G$”) $\leftarrow H$ ;
   return $H$ ;
}
```

Figure 11: Algorithm for multiplication.

(Multiplication by a constants)
Multiplication of F by an integer constant c means that each value of term is multiplied c times. If c is exactly exponential number of (-2), this operation is just shifting each digits of ZBDD vectors, so computation time is linear to the number of digits, not depending on the number of ZBDD nodes. For general integer c, we decompose c into a bit-vector $c_0, c_1, \ldots, c_m$ and compute $F \times (-2)^i c_i$ for each i. After that we calculate total of them by using addition operation, described as follows.

(Addition and Subtraction) Addition of the two VSOPs $F + G$ is defined as generating a new VSOP expression such that each value of product term is sum of values of the same item combinations in F and G. For example, When $F = ab + 2bc - 3c$ and $G = 3ac - 2bc + c$, then $(F + G) = ab + 3ac - 2c$.

Figure 10 shows the algorithms for addition and subtraction based on ZBDD operations. If $(F \cap G)$ is empty, that means there are no common combinations at any digit, in such case we do not need any carry up, so the addition $(F+G)$ can be completed by just merging them $(F \cup G)$. On the other hand, if $(F \cap G)$ contains some common combinations, it represents the set of carries of respective digits. We then make twice of the set of carries and call addition operation again to sum up the carries. By repeating this process, common combinations are eventually exhausted and the procedure is completed.

Since we use the binary coding based on (-2), the one-bit shift corresponds to not twice, but (-2) times, so the carry up formula becomes $S - (-2 \cdot C)$. Namely, we call a subtraction from the addition procedure. Similarly, a borrow of subtraction calls an addition operation. We can implement the both operations with a dual structure.

(Multiplication of two VSOPs) Here we define the multiplication (or product) of two VSOPs $(F \times G)$ as generating all possible concatenations of two product terms in respective F and G.

Using the multiplication by items or constants, we can compose the multiplication algorithm for the general VSOPs, as shown in Fig. 11. This algorithm is based on the divide-and-conquer idea. Suppose v is the highest-ordered item, F and G are then factored into two parts:
$F = v \cdot F_1 \cup F_0, \quad G = v \cdot G_1 \cup G_0$.
Under this factorization, the product $(F \times G)$ can be written as:
$((F_1 \times G_1) + (F_1 \times G_0) + (F_0 \times G_1)) \times v + (F_0 \times G_0)$.
Each sub-product term can be computed recursively. The expressions are eventually broken down into trivial ones and the result is obtained. In the worst case, this algorithm would require exponential number of recursive calls for the number of items; however, we can accelerate them by using a hash-based cache memory which stores the results of recent operations. By referring to the cache before each recursive call, we can avoid duplicate executions for equivalent sub-formulas. Consequently, the execution time depends on ZBDD size, not on the number of terms.

```
procedure($F/G$)
{     if ($G$ =constant:$c$) return ($F/c$) ;
      if ($F$ =constant:$c$) return 0 ;
      $Q \leftarrow$ cache("$F/G$") ;  if ($Q$ exists) return $Q$ ;
      $v \leftarrow G.top$ ; /* the highest variable in $G$ */
      $(F_0, F_1) \leftarrow$ factors of $F$ by $v$ ;
      $(G_0, G_1) \leftarrow$ factors of $G$ by $v$ ;
      $Q \leftarrow F_1/G_1$ ;
      if ($G_0 \neq 0$ and $Q \neq 0$ )
          $Q_0 \leftarrow F_0/G_0$ ;
          $Q \leftarrow$ (choose value from $Q$ or $Q_0$
              absolutely smaller one) ;
      cache("$F/G$") $\leftarrow Q$ ;
      return $Q$ ;
}
```

Figure 12: Algorithm for division.

(Division by an item) Division of a VSOP by an item, the quotient (F/v) and the remainder $(F\%v)$ are defined as classification of the product terms into the two subset, including or excluding v in the item combinations. These operations are exactly same as *Onset* and *Offset* operations of ZBDDs.

(Division by a constant) Division of a VSOP by a constant, (F/c) and $(F\%c)$, are simply defined as integer division (quotient and reminder) for each value of product terms in F. For example, computing $(F/30)$ can delete all product terms whose values are less than 30. Oppositely, $(F\%30)$ extracts such product terms valued less than 30. We can implement this numerical operation by using arithmetic shift and addition/subtraction operations.

(Division of VSOPs) In the VSOP algebra, we have the non-linear multiplication rule $v \times v = v$, and this rule leads that the result of arithmetic division (F/G) is not decided uniquely in general. Thus, we must define our division rule to make a unique result.

In the model of "Boolean" sum-of-products expressions without integer values, *Weak-division method*[3] has been known for long

time and widely used in VLSI logic optimization problems. This division method is based on the following rule:
If the divisor G consists of multiple product terms T_i, the quotient $Q(= F/G)$ is defined as the sum of product terms included in every $Q_i = F/T_i$ for all i.

Now we propose here the new division method, named *Valued weak division*, which is natural extension of (boolean) weak division. This new method is the same as conventional one until calculating Q_i. After that, we do not extract common product terms, but calculating values absolutely minimum in all Q_i. For example, assume that $F = 2ab+4ac+ad-2bc+3bd$ and $G = a + b$, then
$Q_1 = (F/a) = 2b + 4c + d$,
$Q_2 = (F/b) = 2a - 2c + 3d$
and we obtain $Q = -2c + d$.

If the given F and G have only boolean values in every terms, our division method gives completely same results as conventional weak division, so it is a natural extension of conventional method.

Figure 12 shows the algorithm of this division methods using ZBDDs. This is an extension of *ZBDD-based fast weak division method*[11] to the VSOP model. As well as the multiplication algorithm, we can accelerate the execution by using a hash-based cache memory to avoid duplicate executions for equivalent sub-formulas, and the computation time depends on ZBDD size, not on the number of terms.

The remainder of division ($F\%G$) can be obtained by computing $F - (F/G) \times G$.

(Comparison) VSOP program supports the operators (`==` `!=` `>` `>=` `<` `<=`) to compare the numerical values of the two VSOPs. Each of those operators extracts all the product terms included at least in the left or the right expressions and satisfying the arithmetic relation of the operator. For example, suppose $F = 3ab + 2bc - c$ and $G = 2ab - 2b + 3c$, and then we can get $(F > G) = ab+bc+b$. On the same case, (F `!=` 0) becomes $ab + bc + c$, and this is regarded as the regularization of all non-zero values to 1 (Boolean). Those comparison operations can be executed in almost same computation time as addition/subtraction operations.

(Other operations) We also implemented the *If-Then-Else* operator F ? G : H, which extracts the product terms from G such that the item combinations included in F, and also extracts the terms from H for the item combinations not included in F. Using this operations with arithmetic comparisons, we can specify various nonlinear functions. For instance, $(F > G)$? F : G generates a VSOP choosing the terms with larger values between F and G.

In addition, we implemented *Restrict* and *Permit* operations proposed in [19], which are basically same as *SupSet* and *Subset* operations in [6]. F.Restrict(G) extracts the product terms from F such that the item combination is a superset of at least one item combination in G. On the other hand, F.Permit(G) extracts the product terms from F such that the item combination is a subset of at least one item combination in G. The computation time is almost linear to ZBDD size. These two operations are useful for solving constraint satisfaction problems[19] by describing restrictive or permissive conditions with VSOP expressions.

5 Display Formats for Computation Results

VSOP program provides several helpful display formats to show the calculation results to the user. We explain typical formats as follows.

(Sum-of-products form with coefficients) The most basic method is to enumerate all product terms with their values. For example, the formula $3abc + 2bc - c$ shows all product terms with coefficients. The order of product terms is a lexicographical manner of item combinations. This format is easy to see if the number of terms is not so many. In our program, one VSOP data may have millions

```
 a b : c d
     |   00    01    11    10
  00 |    0     0     0     3
  01 |    1    25     2     0
  11 |    0     0    -4    -2
  10 |    1     0    -1     0
```

Figure 13: Integer Karnaugh map.

of terms, and in such cases, we cannot finish the output in a practical time.

(Integer Karnaugh map) As shown in Fig. 13, using a matrix indexing item combinations, and display the integer value on each element. We call this an *Integer Karnaugh map.* It is useful to understand the behavior of the VSOP data, but they are practical only for fewer than five or six items.

(Sorting by values) In some cases, it is useful to make sorting of the product terms in terms of their values. For example, the expression $2ab + 3ac + 2b - bc + 3$ can be listed as follows.

3:	$ac + 1$
2:	$ab + b$
-1:	bc

(Bit-wise listing) We can list the respective digits of the internal ZBDD vector representation. It is used for observing the relationship of VSOP data and the internal data structures.

(Statistical information) To know the total number of product terms in a VSOP expression corresponds to compute the number of solutions for a combinatorial problem. Although the number may become an exponential to the number of items, we can quickly count it only in a linear time to the ZBDD size. In addition, we can get other statistical information such as the density of the solutions, and the number of ZBDD nodes.

(Any satisfiable solutions) Sometimes we do not have to display all the solutions, just needed to see any one solution (or a counter example). If a ZBDD for the VSOP data has been constructed, it is easy (in a time linear to number of items) to show any one product term, even if the VSOP data is too complicated to display all at once.

When each item has a cost to use in a combination, we can also find the minimum (or maximum) cost combinations in the VSOP data. This operation can be executed in a linear time to the ZBDD size.

Our program can also display the maximum (or minimum value) in the VSOP data. In addition, the set of items used in the given VSOP data can be listed.

```
***** VSOP calculator <v0.95> *****
vsop> symbol a b c d e
vsop> F = (a + 2 b)(c + d)
vsop> print F
  a c + a d + 2 b c + 2 b d
vsop> print /rmap F
  a b : c d
      |   00    01    11    10
   00 |    0     0     0     0
   01 |    0     2     0     2
   11 |    0     0     0     0
   10 |    0     1     0     1
vsop> G = (2 a - d)(c - e)
vsop> print G
  2 a c - 2 a e - c d + d e
vsop> H = F * G
vsop> print H
  4 a b c d - 4 a b c e + 4 a b c - 4 a b
   d e + a c d e - 2 a c e + 2 a c - a d
   e + 2 b c d e - 4 b c d + 2 b d e
vsop> print /count H
    11
vsop> print /size H
    24 (35)
vsop> quit
```

Figure 14: Example of execution

6 Applications

Based on ZBDD techniques, we developed an arithmetic calculator to handle large-scale sum-of-products expressions with a number of item symbols. Here we briefly present the specification of VSOP and some typical applications.

6.1 VSOP Calculator

This program, called VSOP, has a C-shell-like interface, both for interactive keyboard inputs and batch style execution from a script file. The program is written in C, C++, and yacc, executable on 32bit Linux PCs.

In VSOP scripts, we can use two kind of symbols, *item symbols* and *program variables.* Item symbols, denoted by strings starting with a lower-case letter, represent the items used in the set of combinations. Program variables, starting with an upper-case letter, are used to identify the memory to which a computation result to be stored temporarily. We can describe multi-level expressions by using these two type of symbols. Calculation results are displayed in expressions of including item symbols only, not using program variables. VSOP allows up to 65,510 different item symbols to be used, and no limit for program variables, as long as the ZBDD nodes are handled in the main memory.

VSOP calculator supports not only set operations but also numerical arithmetic operations based on *Valued-Sum-Of-Products* algebra, as presented in the previous sections. The program parses the script only from left to right. Neither branches nor loop controls are supported. However, using another script processor such as C-shell or Perl, we can generate a straight VSOP script file by unrolling the complicated control structures, and feed it to the VSOP calculator by pipelined manner.

Our program need a few seconds to calculate VSOP expressions which are the size of human-readable or writable. More than ten millions of ZBDD nodes can be handled according to main memory capacity. Our ZBDD package uses about 30 byte memory per node. Calculation results are displayed in various formats as shown previously. Figure 14 shows a simple execution example.

6.2 Basic Performances

To evaluate our method, we constructed VSOP expressions of large number of product terms with large values. In this experiments, we used a Pentium-4 PC (800MHz, 512MB, SuSE Linux 9). We can deal with up to about 10,000,000 ZBDD nodes in this machine.

Table 1: Generating VSOPs for $\Pi_{k=1}^{n}(x_k + k)$

n	#Terms	Max.value	#Nodes	Time(s)
4	16	24	16	0.002
8	256	40,320	199	0.007
12	4,096	479,001,600	1,866	0.108
16	65,536	20,922,789,888,000	9,383	0.689
20	1,048,576	(2.43×10^{18})	76,705	14.399
24	16,777,216	(6.20×10^{23})	530,308	276.993

We first generated ZBDDs for large constant numbers. 100 !, which becomes as much as a 160 digits of decimal number, can be represented with only 121 nodes of ZBDD, in 0.2 second to compute it. Next we tried calculating $\Pi_{k=1}^{n}(x_k + k)$. As shown in Table 1, within a feasible time and space, we can generate ZBDDs for extremely large-scale expressions, some of which consist of millions of terms.

6.3 Database Analysis

Here we consider the following example of transaction database. In this data, one line corresponds to one record, and the numbers represents IDs of items included in the record.

```
1 3 9 13 23 25 34 36 38 40 52 54 59 63 67
2 3 9 14 23 26 34 36 39 40 52 55 59 63 67
. . .
```

In this database, the similar item combinations (sub patterns) appear many times in multiple records. To count the frequency (number of appearances) of the patterns is an important and fundamental problem in data mining techniques[2], which is regarded as the basis of knowledge processing. Using VSOP calculator, we can efficiently construct the pattern histogram and applying various analysis operations to the histogram data. At first, we transform the above database file into the following VSOP script.

```
P = 0
P = P + (1+x1)(1+x3)(1+x9)(1+x13)(1+x23)
 (1+x25)(1+x34)(1+x36)(1+x38)(1+x40)
 (1+x52)(1+x54)(1+x59)(1+x63)(1+x67)
P = P + (1+x2)(1+x3)(1+x9)(1+x14)(1+x23)
 (1+x26)(1+x34)(1+x36)(1+x39)(1+x40)
 (1+x52)(1+x55)(1+x59)(1+x63)(1+x67)
P = P + ...
```

Each line represents the set of all sub-patterns contained in one record. After execution of this script for all records, the variable `P` holds the histogram for all sub-patterns included in the database.

Once the histogram is generated, various queries can be applied as a sequence of VSOP operations. For example,

```
  print /count (P/30)
```

displays the number of product terms with the values more than 29, which mean the number of frequent patterns included in more than 29 records in the database. For another example,

```
  print /count (P/(x1 x2))
```

shows the number of patterns including both x1 and x2.

Here we will show an experimental result for one of the data mining benchmark “mushroom”[7] to compute the pattern-histogram. This dataset consists of 8,124 records each of which is a combination chosen from 119 items. VSOP calculator required about 4 minutes to generate the pattern-histogram of this dataset. 513,762 nodes of ZBDD is generated for the histogram, and totally 5,574,930,438 patterns are stored in the histogram. Such ZBDD-based database analysis method is presented in [16] for more detail.

6.4 Solving Constraint Satisfaction Problems

Okuno et al.[19] presented the way to solve various constraint satisfaction problems (CSP) using BDDs or ZBDDs. In this paper, they consider N-queens problems and magic square problems as examples of CSPs. Those problems can be described by arithmetic Boolean expressions handling logic variables and numerical numbers.

Previously, there is an arithmetic Boolean expression manipulator ”BEM-II”[9] based on (ordinary) BDDs, and the program was utilized for many works[17, 18, 8]. However, there have not been a good arithmetic calculator based on ZBDDs, so the research of ZBDD applications for CSPs have not been active as well as ordinary BDDs. Our VSOP calculator will extend the data model from Boolean functions to combinatorial sets.

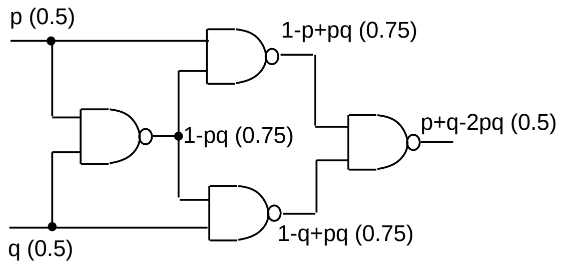


Figure 15: Probabilistic symbolic logic simulation.

For example, to describe constraints for a magic square, we can write the number for each square $A, B, C \ldots$ as:

```
 A = a1 + 2 a2 + 3 a3 + 4 a4 + ...
 B = b1 + 2 b2 + 3 b3 + 4 b4 + ...
 C = ...
```

We then compute the following formula.

```
 S = A (B != 0) + B (A != 0)
```

This result becomes as:

```
2 a1 b1 + 3 a1 b2 + 4 a1 b3 + 5 a1 b4 +
3 a2 b1 + 4 a2 b2 + 5 a2 b3 + 6 a2 b4 +
4 a3 b1 + 5 a3 b2 + 6 a3 b3 + 7 a3 b4 ...
```

We can see this expression enumerates the sum of two numbers at A and B for all possible combinations. Next, the formula

```
 S = S (C != 0) + C (S != 0)
```

produces the total number of A, B, and C for all possible combinations, and it is stored in S. After that, the formula

```
 C = (S == 15 (S != 0))
```

generates the constraint C such that the total S equals to 15. In similar manner, we can generate VSOP data representing the constraints of all horizontal, vertical, and diagonal lines.

In this way, we can describe various CSPs by using VSOP scripts, and easily try solving it by VSOP calculator.

6.5 Probabilistic Symbolic Simulation

VSOP calculator is based on the arithmetic operation rules as $x + x = 2x$, $x \times x = x$, and $x \times y = xy$. These rules are the same as the probabilistic calculation that the variables x and y represent probabilities. If the two events occur independently, the logical AND becomes arithmetic products of two variables, but if the two event are based on a same probabilistic variable, the logical AND does not become x^2 but just x. Consequently, VSOP calculator can be used for probabilistic analysis of systems in various areas.

One good application is computing signal probability in logic circuits. As illustrated in Fig. 15, on each primary input of the circuit, we assign a variable representing the probability that the signal is '1'. Then, the probability at primary outputs and internal nets can be expressed exactly in VSOP expressions using those probabilistic variables. On each logic gate with input A, B and output Y, we can compute $Y = A \times B$ for AND gate, $Y = A + B - (A \times B)$ for OR gate, and $Y = 1 - A$ for NOT gate.

This technique is applicable for various kinds of statistic analysis, such as probabilistic fault simulation, estimating power consumption, and timing hazard analysis.

7 Conclusion

We have presented a method of computing combinatorial item sets with numerical values. This method consists of an efficient data structure, manipulation algorithms, and helpful display formats. VSOP calculator, implemented based on the above techniques, is customizable for various applications. We expect it to be utilized as a helpful tool in solving many problems in computer science. In future, we will release our program as open software to facilitate research and development for various area.

Acknowledgment

The author thanks Prof. Arimura and Prof. Zeugmann of Hokkaido Univ. for their 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] Akers, S. B., Binary decision diagrams, IEEE Trans. Comput., C-27, 6 (1978), 509–516.

[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. K. Brayton, R. Rudell, A. Sangiovnni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," IEEE Trans. on CAD, vol. CAD-6, pp. 1062–1081, Nov. 1987.

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

[5] R. E. Bryant and Y.-A. Chen, "Verification of arithmetic functions with binary moment diagrams," Proc. of 32nd ACM/IEEE Design Automation Conference (DAC'95), session 32.1, June. 1995.

[6] O. Coudert, "Solving graph optimization problems with ZBDDs", In Proc. of IEEE The European Design and Test Conference (ED&TC'97), pp. 244-248, Mar. 1997.

[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] Y. Hayashi and J. Matsuki, "Determination of Optimal System Configuration in Japanese Secondary Power Systems," IEEE Trans. on Power Systems, VOL. 18, NO. 1, pp. 394–399, Feb. 2003

[9] S. Minato: "BEM-II: An Arithmetic Boolean Expression Manipulator Using BDDs", IEICE Trans. Fundamentals, Vol. E76-A, No. 10, pp. 1721-1729, Oct. 1993.

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

[11] S. Minato: "Calculation of Unate Cube Set Algebra Using Zero-Suppressed BDDs", In Proc. of 31st ACM/IEEE Design Automation Conference (DAC'94), pp. 420-424, Jun. 1994.

[12] S. Minato: "Implicit Manipulation of Polynomials Using Zero-Suppressed BDDs", In Proc. of IEEE The European Design and Test Conference (ED&TC'95), pp. 449-454, Mar. 1995.

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

[14] S. Minato: "Fast Factorization Method for Implicit Cube Set Representation", IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems, VOL. 15, No. 4, pp. 377-384, Apr. 1996.

[15] 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.

[16] 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.

[17] T. Miyazaki, Boolean-based formulation for data path synthesis, IEEE Asia-Pasific Conference on Circuits and Systems 'APCCAS'92), pp. 201–205, Dec. 1992.

[18] H.G. Okuno, "Reducing Combinatorial Explosions in Solving Search-Type Combinatorial Problems with Binary Decision Diagram," Trans of Information Processing Soc. Japan (IPSJ), (in Japanese), vol. 35, no. 5, pp 739-753, May 1994.

[19] H.G. Okuno, S. Minato, and H. Isozaki, On the properties of combination set operations, Information Processing Letters, Vol. 66, pp. 195–199, May 1998.