Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems

Shin-ichi Minato
NTT LSI Laboratories
3-1, Morinosato Wakamiya, Atsugi-shi
Kanagawa Pref., 243-01, JAPAN

Abstract

In this paper, we propose *Zero-Suppressed BDDs (0-Sup-BDDs)*, which are BDDs based on a new reduction rule. This data structure brings unique and compact representation of *sets* which appear in many combinatorial problems. Using 0-Sup-BDDs, we can manipulate such sets more simply and efficiently than using original BDDs. We show the properties of 0-Sup-BDDs, their manipulation algorithms, and good applications for LSI CAD systems.

1 Introduction

In the various phases of LSI design systems, we are often faced with combinatorial problems. Techniques for solving combinatorial problems efficiently are very important not only for developing LSI CAD systems but also for many other applications. The data structure is the key to solving combinatorial problems on computers.

Binary Decision Diagrams (BDDs), which are graph representations of Boolean functions, were introduced by Akers[1] and developed by Bryant[2]. Using BDDs, we can represent Boolean functions uniquely and compactly. At first, BDDs were applied to verification and test pattern generation for combinational circuits[3][4]. In these initial simple applications, BDDs were generated for the functions of the circuits and compared to find design errors or fault activities. These were straight ways to utilize the properties of BDDs.

Recently, there have been many works on BDD applications, but most of them do not use BDDs to simply represent Boolean functions. In many cases, BDDs are used to represent *sets*. One proposal is for multiple fault simulation by representing sets of fault combinations with BDDs[5]. Two others are verification of sequential machines using BDD representation for state sets[6], and computation of prime implicants using *Meta Products*[7], which represent cube sets using BDDs. There is also general method for solving binite covering problems using BDDs[8].

A set can be represented by a Boolean function, which is called *characteristic function*. Using a BDD for a characteristic function, we can enumerate all the elements of a set implicitly. This compact representation enables us to manipulate a huge number of elements efficiently as if using parallel computation. However, we found that this set representation does not completely match the properties of BDDs. In such cases, sometimes the size of BDDs grow large because the reduction rules are not very effective.

In this paper, we propose *Zero-Suppressed BDDs*, which are BDDs based on a new reduction rule. This data structure is more efficient and simpler than usual BDDs when we manipulate sets in combinatorial problems.

2 BDDs and Combination Sets

2.1 BDDs

A Binary Decision Diagram (BDD) is a directed graph representation of a Boolean function, as shown in Fig. 1(a). BDDs have the two terminal nodes, which we call *0-terminal node* and *1-terminal node*, and many decision nodes with the two edges, called *0-edge* and *1-edge*. A BDD is derived by reducing a binary tree graph, as shown in Fig. 1(b). The binary tree represents the recursive execution of *Shannon's expansion*.

The following reduction rules give a *Reduced Ordered BDD (ROBDD)*, which represents a Boolean function efficiently (see [2] for details.)

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

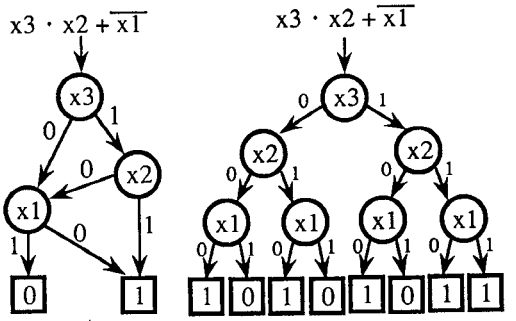
ROBDDs give canonical forms for Boolean functions when the variable order is fixed. Most of the works on BDDs are based on the above reduction rules. In the following sections, for the sake of simplification, BDDs (or original BDDs) refer to ROBDDs.

A set of BDDs representing multiple functions can be united into a graph which consists of BDDs sharing their sub-graphs with each other. The efficiency of manipulation can be improved by managing all the BDDs as a single graph, as in Fig. 3. We call such graphs *SBDDs (Shared BDDs)*[9]. We can furthermore reduce the operation time and memory requirement by using *attributed edges*[9], which represent certain logic operations such as inverting.

BDD packages implemented with the above techniques exhibit the following useful properties.

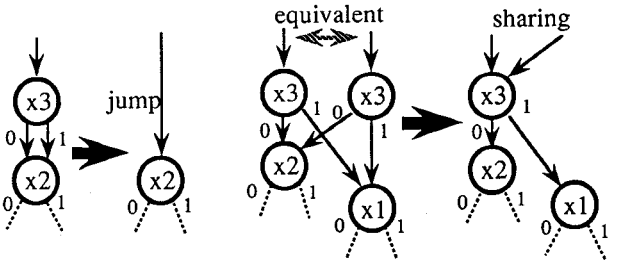
- They can generate BDDs for large-scale functions, some of which have never been represented by previous methods.

30th ACM/IEEE Design Automation Conference®



(a) BDDs (b) Binary Tree Graph

Fig. 1: Binary Decision Diagram (BDD)



(a) Node Elimination (b) Node Sharing

Fig. 2: Reduction Rules on BDDs

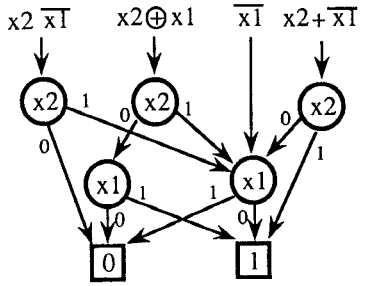


Fig. 3: Shared BDD (SBDD)

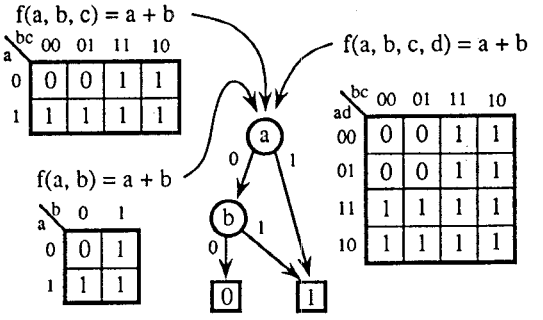


Fig. 4: Suppression of Irrelevant Variables

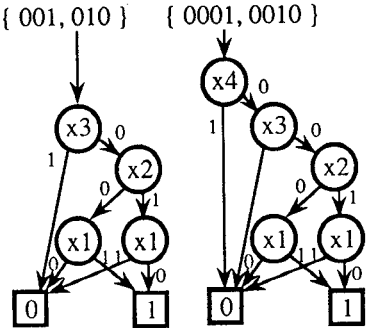


Fig. 5: BDDs for Combination Sets

- After generating BDDs, the equivalence of two functions can be checked in a constant time.
- Logic operations can be carried out within a time that is almost proportional to the size of graphs.

The size of BDDs largely depends on the order of the input variables. We can find a good order in many cases using heuristic methods[9][10].

The BDD package cannot exhibit its power if the graph size is too large to be held in the memory. It is important how the graph is shrunk by the reduction rules. One recent paper[11] shows that for general (or random) Boolean functions, node sharing makes a much more significant contribution to storage saving than node elimination. However, we consider that the node elimination is also important for practical functions. For example, as in Fig. 4, the form of a BDD does not depend on the number of input variables as long as the expressions of the functions are the same. Using BDDs, the irrelevant variables are suppressed automatically and we do not have to consider them. This property is significant because sometimes we manipulate a function which depends on only a few variables among hundreds. This suppression comes from the node elimination of BDDs.

2.2 Combination Sets

A combination on n objects can be represented by an n bit binary vector, $(x_n x_{n-1} \ldots x_2 x_1)$, where each bit, $x_k \in \{1, 0\}$, expresses whether the corresponding object is included in the combination or not. A set of combinations can be represented by a set of the n bit binary vectors. We call such sets *combination sets*. Combination sets can be regarded as subsets of the power set on n objects.

Combination sets can describe solutions to combinatorial problems. We can solve combinatorial problems by manipulating combination sets. The representation and manipulation of combination sets are important techniques for many applications.

We can represent a combination set with a Boolean function by using n input variables for each bit of the vector. The output value of the function expresses whether each combination specified by the input variables are included in the set or not. Such Boolean functions are called *characteristic functions*. The operations of sets, such as union, intersection and difference, can be executed by logic operations on characteristic functions.

Using BDDs for characteristic functions, we can manipulate combination sets efficiently. In such BDDs, the paths from the starting node to the 1-terminal node, which we call *1-paths*, represent possible combinations in the set. Because of the effect of node sharing, BDDs represent combination sets with a huge number of elements compactly. In many practical cases, the size of graphs becomes much less than the number of elements. BDDs can be generated and manipulated within a time that is almost proportional to the size of graphs, while previous set representations, such as arrays and sequential lists, require a time proportional to the number of elements.

Set manipulation using BDDs is very efficient, however, there is one inconvenience in that the form of BDDs depends on the number of input variables, as shown in Fig. 5. Therefore we have to fix the number of input variables before generating BDDs. This inconvenience comes from the difference in the model on default variables. In combination sets, default variables are regarded as zero when the characteristic function is true, since the irrelevant objects never appear in any combination. Unfortunately, such variables can not be suppressed in the BDD repre-

sentation. We have to generate many useless nodes for irrelevant objects when we manipulate very sparse combinations. Node elimination does not work well in reducing the graphs in such cases.

In the following section, we describe a method that solves the above problem by using BDDs based on new reduction rules.

3 Zero-Suppressed BDDs

We propose the following reduction rules for BDDs to represent combination sets efficiently.

1. Eliminate all the nodes whose 1-edge points to the 0-terminal node. Then connect the edge to the other subgraph directly, as shown in Fig. 6.
2. Share all equivalent sub-graphs the same as for original BDDs.

Notice that, contrary to the situation with original BDDs, we do not eliminate the nodes whose two edges point to the same node. This reduction rule is asymmetric for the two edges, as we do not eliminate the nodes whose 0-edge points to a terminal nodes.

We call BDDs based on the above rules *Zero-Suppressed BDDs (0-Sup-BDDs)*. If the number and the order of input variables are fixed, a 0-Sup-BDD represents a Boolean function uniquely. This property is clear because a binary tree as in Fig. 1(b) can be reconstructed from a 0-Sup-BDD by applying the reduction rule reversely.

Fig. 7 shows 0-Sup-BDDs representing the combination sets which are the same ones shown in Fig. 5. One feature of 0-Sup-BDDs is that the form does not depend on the number of input variables as long as the combination sets are the same. We do not have to fix the number of input variables before generating graphs. 0-Sup-BDDs automatically suppress the variables for objects which never appear in any combination. It is very efficient when we manipulate very sparse combinations.

For evaluating efficiency of 0-Sup-BDDs, we conducted a statistical experiment. We generated a set of one hundred combinations each of which selects k out of 100 objects randomly. We then compared the size of both the 0-Sup-BDDs and original BDDs representing these random combinations. The result for variation in k is shown in Fig. 8. This result shows that 0-Sup-BDDs are much more compact than original BDDs especially when k is small. Namely, 0-Sup-BDDs are remarkably effective for representing sets of sparse combinations. The effect weakens for large k; however, we can use complement combinations to make k small. For example, the combination selecting 90 out of 100 objects is equivalent to selecting the remaining 10 out of 100.

Another advantage of 0-Sup-BDDs is that the number of 1-paths in the graph is exactly equal to the number of elements in the combination set. In original BDDs, the node elimination breaks this property. Therefore, we believe that 0-Sup-BDDs are more suitable than original BDDs to represent combination sets.

On the other hand, it would be better to use original BDDs when representing ordinary Boolean functions, as shown in Fig. 4. The difference is in the models on default variables; that is, "fixed to zero" in combination sets, and "both OK" in Boolean functions. We can choose one of the two types of BDDs according to the feature of applications.

4 Manipulation of 0-Sup-BDDs

4.1 Basic Operations

In original BDDs, we first generate BDDs with only one input variable for each input, and then we construct more complex BDDs by applying basic logic operations, such as AND, OR and EXOR. 0-Sup-BDDs are also constructed from the trivial graphs by applying basic operations, but the kinds of operations are not the same as with original BDDs since 0-Sup-BDDs are adapted for combination set manipulation.

We arranged the line up of basic operations for 0-Sup-BDDs as follows:

Empty()	returns ϕ. (empty set)
Base()	returns $\{\mathbf{0}\}$.
Subset1(P, var)	returns the subset of P such as $var = 1$.
Subset0(P, var)	returns the subset of P such as $var = 0$.
Change(P, var)	returns P when var is inverted.
Union(P, Q)	returns $(P \cup Q)$
Intsec(P, Q)	returns $(P \cap Q)$
Diff(P, Q)	returns $(P - Q)$
Count(P)	returns $\|P\|$. (number of elements)

In Fig. 9, we show the examples of 0-Sup-BDDs generated with the above operations. Empty() returns the 0-terminal node, and Base() is the 1-terminal node. Any one combination can be generated with Base() operation followed by Change() operations for all the variables which appear in the combination. Using Intsec() operation, we can check whether a combination is contained in a set or not.

In 0-Sup-BDDs, we do not have NOT operation, which is an essential operation in original BDDs. This is reasonable since the complement set $\overline{P}$ cannot be computed if the universal set U is not defined. Using the difference operation, $\overline{P}$ can be computed as $U - P$.

4.2 Algorithms

We show here that the above operations for 0-Sup-BDDs can be executed recursively like original BDDs.

First, to describe the algorithms simply, we define the procedure Getnode(top, P_0, P_1), which generates (or copies) a node for a variable top and two subgraphs P_0, P_1. In the procedure, we use a hash table, called *uniq-table*, to keep each node unique. Node elimination and sharing are managed only by Getnode().

```
Getnode (top, P0, P1) {
    if (P1 == φ) return P0;  /* node elimination */
    P = search a node with (top, P0, P1) in uniq-table;
    if (P exist) return P;   /* node sharing */
    P = generate a node with (top, P0, P1);
    append P to the uniq-table;
    return P;
}
```

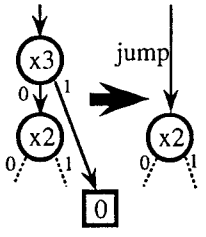


Fig. 6: New Reduction Rule on 0-sup-BDDs

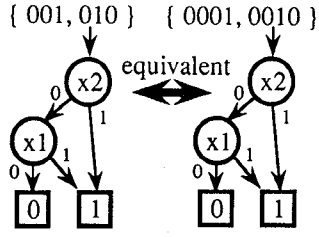
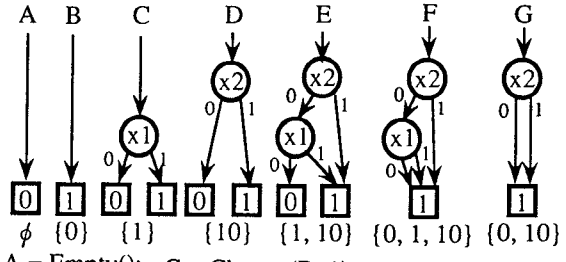


Fig. 7: 0-Sup-BDDs for Combination Sets



A = Empty(); C = Change(B, 1); E = Union(C, D);
B = Base(); D = Change(B, 2); F = Union(B, E);
G = Diff(F, C);

Fig. 9: Examples of 0-sup-BDD Manipulation

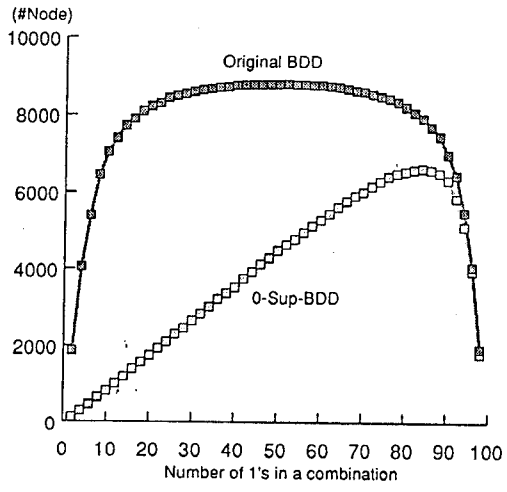


Fig. 8: Experimental Result

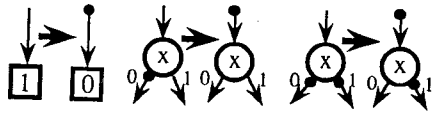


Fig. 10: Rules on 0-element Edges

Using Getnode(), the operations for 0-Sup-BDDs are described as follows. In the description, $P.top$ means a variable with the highest order, and P_0, P_1 are the two subgraphs.

```
Subset1 (P, var) {
  if (P.top < var) return ϕ;
  if (P.top == var) return P1;
  if (P.top > var)
    return Getnode(P.top, Subset1(P0, var), Subset1(P1, var));
}

Subset0(P, var) {
  if (P.top < var) return P;
  if (P.top == var) return P0;
  if (P.top > var)
    return Getnode(P.top, Subset0(P0, var), Subset0(P1, var));
}

Change (P, var) {
  if (P.top < var) return Getnode(var, ϕ, P);
  if (P.top == var) return Getnode(var, P1, P0);
  if (P.top > var)
    return Getnode(P.top, Change(P0, var), Change(P1, var));
}

Union (P, Q) {
  if (P == ϕ) return Q;
  if (Q == ϕ) return P;
  if (P == Q) return P;
  if (P.top > Q.top) return Getnode(P.top, Union(P0, Q), P1);
  if (P.top < Q.top) return Getnode(Q.top, Union(P, Q0), Q1);
  if (P.top == Q.top)
    return Getnode(P.top, Union(P0, Q0), Union(P1, Q1));
}
```

```
Intsec (P, Q) {
  if (P == ϕ) return ϕ;
  if (Q == ϕ) return ϕ;
  if (P == Q) return P;
  if (P.top > Q.top) return Intsec(P0, Q);
  if (P.top < Q.top) return Intsec(P, Q0);
  if (P.top == Q.top)
    return Getnode(P.top, Intsec(P0, Q0), Intsec(P1, Q1));
}

Diff (P, Q) {
  if (P == ϕ) return ϕ;
  if (Q == ϕ) return P;
  if (P == Q) return ϕ;
  if (P.top > Q.top) return Getnode(P.top, Diff(P0, Q), P1);
  if (P.top < Q.top) return Diff(P, Q0);
  if (P.top == Q.top)
    return Getnode(P.top, Diff(P0, Q0), Diff(P1, Q1));
}

Count (P) {
  if (P == ϕ) return 0;
  if (P == {0}) return 1;
  return Count(P0) + Count(P1);
}
```

These algorithms take an exponential time for the number of variables in the worst case; however, we can accelerate them by using a cache which memorizes results of recent operations in the same manner as it is used in original BDDs[2]. By referring the cache before every recursive call, we can avoid duplicate executions for equivalent sub-graphs. With this technique, we can execute these operations in a time that is almost proportional to the size of graphs.

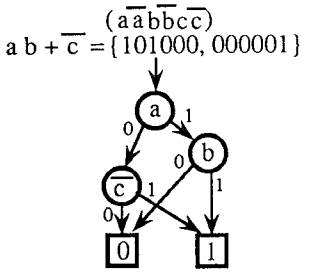


Fig. 11: Cube Set Representation Using a 0-sup-BDD

Table 1: Experimental Result for Irredundant Cube Set Representation

Name	In	Out	Net	#Cube	#Literal	#ZBDD	Time(s)	#BDD	Time
c432	36	7	203	84235	969028	14407	83.2	41869	111.8
c499	41	32	275	348219564	6462057445	195356	2400.1	(>1M)	-
c880	60	26	464	114299	1986014	18108	78.7	69446	119.3
c1908	33	25	938	56323472	1647240617	233775	385.6	(>1M)	-
c5315	178	123	2608	137336131	742606419	41662	886.4	(>1M)	-

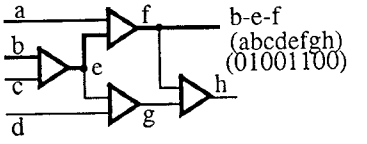


Fig. 12: Path Representation as a Combination

4.3 Using of Attributed Edges

In original BDDs, we can reduce the execution time and memory requirement by using *attributed edges*[9] to indicate certain logic operations such as inverting. Also 0-Sup-BDDs may have a kind of attributed edges, but the operation should be different from original BDDs.

Here we present an attributed edge for 0-Sup-BDDs. This attributed edge, named *0-element edge*, indicates that the pointing sub-graph has a 1-path which consists of 0-edges only. In other word, a 0-element edge means that the combination set includes the element of **0**. We use the notation $\hat{P}$ to express the 0-element edge pointing P.

As with other attributed edges, we have to place a couple of constraints on the location of 0-element edges to keep the uniqueness of the graphs. They are:

- Use the 0-terminal only, since $\{\mathbf{0}\}$ can be written as $\hat{\phi}$
- Do not use 0-element edges at the 0-edge on each node.

If necessary, 0-element edges can be carried over, as shown in Fig. 10. The constraint rules can be implemented in the Getnode().

0-element edges accelerate operations on 0-Sup-BDDs. For example, the result of Union(P, $\{\mathbf{0}\}$) depends only on whether P includes the element of **0** or not. In such a case, we can get the result in a constant time using 0-element edges, otherwise we have to repeat the expansion until P becomes a terminal node.

5 Applications

In this section, we show that 0-Sup-BDDs can be readily applied to practical problems. As mentioned in foregoing sections, 0-Sup-BDDs are suitable for manipulation of combination sets, especially when the combinations are sparse.

5.1 Cube Set Representation

In many logic design systems, cube sets (also called covers, PLAs, sum-of-products forms or two-level logics) are employed to represent Boolean functions. Cube sets are sets of cubes each of which is a combination of positive or negative literals for input variables.

Coudert and Madre proposed a representation of cube sets using BDDs named *Meta-products*[7]. Meta-products are BDD representations for characteristic functions of cube sets. In their method, two variables are used for each input and these decide the existence of the literal and its sign. They also presented further reduced graphs, named *Implicit Prime Sets (IPS)*[12], to represent prime cube sets efficiently. However, IPS can represent only *prime* cube sets and they do not give canonical expressions for *general* cube sets.

Using 0-Sup-BDDs, we can represent any cube set simply, efficiently and uniquely. As shown in Fig. 11, a cube set can be seen as a combination set by using two variables for the literals x_k and $\overline{x_k}$. Both x_k and $\overline{x_k}$ never appear in the same cube; that is, at least one of them should be 0. The 0's are conveniently suppressed in 0-Sup-BDDs. The cube set manipulation can be composed of basic operations on 0-Sup-BDDs. The number of cubes equals the number of 1-paths in the graph. The total number of literals can be counted in a time proportional to the size of the graph.

We conducted an experiment to evaluate this method. We generated prime and irredundant cube sets for the functions of large-scale combinational circuits using the *ISOP algorithm* presented at SASIMI'92[13]. We represented the results of the cube sets in 0-Sup-BDDs and in original BDDs to compare the size of graphs and the execution time. In Table 1, *In*, *Out* and *Net* show the size of the circuits. *#Cube* and *#Literal* show the cube sets obtained. *#ZBDD* is the size of 0-Sup-BDDs representing the cube sets, and *#BDD* is when using original BDDs.

The experimental results show that 0-Sup-BDDs enable us to generate very large prime-irredundant cube sets containing billions of literals. Noteworthy is c5315, where only 5.5 literals appear in a cube on average, while the function has 178 inputs (i. e., 356 literals). In such a case where the combinations are very sparse, 0-Sup-BDDs are dramatically effective. When we manipulate cube sets using intermediate variables to represent multi-level logics, the cube sets are more sparse, and 0-Sup-BDDs will be more effective.

5.2 For Combinatorial Problems

The methods of solving combinatorial problems are important in developing VLSI CAD systems. The solutions of a combinatorial problem can be represented in a combination set. There are many practical applications which require good representation of combination sets.

One example is representing sets of paths in networks for timing analysis. We can express a path as a combination of variables each of which represents a net, as shown in Fig. 12. A set of paths can be represented as a combination set. Since a path consists of a comparatively small number of nets in the network, the combinations are usually sparse and effectively represented using 0-Sup-BDDs.

Representation of fault sets in logic circuits is another good application. We can express any combination of faults using variables that represent faults, as presented in [5]. If we represent double or triple faults using this data structure, the combinations will be very sparse, therefore we expect that 0-Sup-BDDs would be effective. This representation method may be also utilized for test pattern compaction and fault diagnosis.

We expect that 0-Sup-BDDs will be useful not only in logic design systems but also in layout or high-level CAD systems, because in these systems there are many instances where the manipulation of sparse combinations is required.

Furthermore, 0-Sup-BDDs can be used to solve general combinatorial problems. Lin and Somenzi proposed a method of solving binite covering problems using BDDs[8]. Using this method as a basis, we can efficiently compute the minimum-cost solution by generating a BDD which represents the constraint function of the problem. This method can also be used if we use 0-Sup-BDDs instead of original BDDs.

0-Sup-BDDs are effective when the solutions are represented in sparse combinations. For example, in the *N-Queen problem*, there are only n queens on a chessboard matrix with n^2 points. If we use variables for each point, solutions are obviously expressed in a sparse combination. Similarly in the *traveling salesman problem*, there are $n(n-1)/2$ paths connecting n points with each other, and we take only n paths to travel to all the points.

Compact representation is the key to solving problems. We consider that many useful algorithms, such as the dynamic programming and divide-and-conquer method, could be efficiently composed of operations on 0-sup-BDDs.

6 Conclusion

We have proposed 0-Sup-BDDs, which are BDDs based on a new reduction rule, and presented their manipulation algorithms and applications. 0-Sup-BDDs can represent a combination set uniquely and more compactly than original BDDs. The effect of 0-Sup-BDDs is remarkable especially when manipulating sparse combinations. We expect that there are many applications which require the manipulation of such combination sets.

0-Sup-BDDs appear to display some interesting properties. So far, we have only evaluated them statistically. Our future work includes a theoretical analysis of 0-Sup-BDDs and developing good practical applications for them.

Acknowledgment

The author wish to acknowledge the interesting discussions with Olivier Coudert and Fabio Somenzi.

References

[1] S. B. Akers: "Binary Decision Diagrams", IEEE Trans. Comput., pp. 509-516, 1978

[2] R. E. Bryant: "Graph-Based Algorithms for Boolean Function Manipulation", IEEE Trans. Comput., pp. 677-691, 1986

[3] M. Fujita, H. Fujisawa and n. Kawato: "Evaluation and Improvement of Boolean Comparison Method Based on Binary Decision Diagrams", Proc. IEEE ICCAD'88, pp. 2-5, 1988

[4] R. K. Gaede, M. R. Mercer, K. M. Bulter and D. E. Ross: "CATAPULT: Concurrent Automatic Testing Allowing Parallelization and Using Limited Topology", Proc. ACM/IEEE 25th DAC, pp. 597-600, 1988

[5] N. Takahashi, n. Ishiura and S. Yajima: "Fault Simulation for Multiple Faults Using Shared BDD Representation of Fault Sets", Proc. IEEE ICCAD'90, pp. 550-553, 1991

[6] J. R. Burch, E. M. Clarke, K. L. McMillan and D. L. Dill: "Sequential Circuit Verification Using Symbolic Model Checking", Proc. ACM/IEEE 27th DAC, pp. 46-51, 1990

[7] O. Coudert and J. C. Madre: "Implicit and Incremental Computation of Primes and Essential Implicant Primes of Boolean Functions", Proc. ACM/IEEE 29th DAC, pp. 36-39, 1992

[8] Bill Lin and Fabio Somenzi: "Minimization of Symbolic Relations", Proc. IEEE ICCAD'90, pp. 88-91, (1990).

[9] S. Minato, N. Ishiura and S. Yajima: Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation, ACM/IEEE Proc. 27th DAC, pp. 52-57, 1990

[10] M. Fujita, Y. Matsunaga and T. Kakuda: On Variable Ordering of Binary Decision Diagrams for the Application of Multi-level Logic Synthesis, Proc. the European Conference on Design Automation, pp.50-54, 1991

[11] H.-T. Liaw and C.-S. Lin: "On the OBDD-Representation of General Boolean Functions" IEEE Trans. Comput., pp. 661-664, 1992

[12] O. Coudert and J. C. Madre: "A New Implicit Graph Based Prime and Essential Prime Computation Technique", Proc. International Symposium on Logic Synthesis and Microprocessor Architecture (ISKIT'92, Japan), pp. 125-131, 1992.

[13] S. Minato: 'Fast Generation of Irredundant Sum-of-Products Forms from Binary Decision Diagrams', Proceedings of the Synthesis and Simulation Meeting and International Interchange (SASIMI'92, Japan), 1992, pp. 64-73