HOKKAIDO UNIVERSITY

Title	Minimum-Width Method of Variable Ordering for Binary Decision Diagrams
Author(s)	Minato, Shin-ichi
Description	PAPER: Special Section on the 4th Karuizawa Workshop on Circuits and Systems
Citation	IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences, E75(A3), 392-399
Issue Date	1992-03-20
Doc URL	https://hdl.handle.net/2115/47490
Rights	copyright©1992 IEICE
Type	journal article
File Information	62_IEICE75_392.pdf

Hokkaido University Collection of Scholarly and Academic Papers : HUSCAP

IEICE TRANS. FUNDAMENTALS, VOL. E75-A, NO. 3 MARCH 1992

PAPER Special Section on the 4th Karuizawa Workshop on Circuits and Systems

Minimum-Width Method of Variable Ordering for Binary Decision Diagrams

Shin-ichi MINATO†, *Member*

SUMMARY *Binary Decision Diagrams* (*BDDs*) and *Shared Binary Decision Diagrams* (*SBDDs*), which are improved BDDs, are useful for implementing VLSI logic design systems. Recently, these representations, which are graph representations of Boolean functions, have become popular because of their efficiency in terms of time and space. The forms of the BDD vary with the order of the input variables though they represent the same function. The size of the graphs greatly depends on the order. The variable ordering algorithm is one of the most important issues in the application of BDDs. In this paper, we consider methods which reduce the graph size by reordering input variables on a given BDD with a certain variable order. We propose the *Minimum-Width Method* which gives a considerably good order in a practicable time and space. In the method, the order is determined by *width of BDDs* as a cost function. In addition, we show the effect of combining our method with the local search method, and also describe the improvement using the threshold. Experimental results show that our method can reduce the size of BDDs remarkably for most examples. The method needs no additional information, such as the topological information of the circuit. The results can be a measure for evaluation of other ordering methods.

key words: *binary decision diagrams, boolean function, logic synthesis, variable ordering*

1. Introduction

Techniques of efficient Boolean function representation and manipulation are very important in implementation of VLSI logic design systems such as logic synthesis, verification, and test generation. *Binary Decision Diagrams* (*BDDs*), which are graph representations of Boolean functions were presented by Akers[1] and Bryant[2]. *Shared Binary Decision Diagrams* (*SBDDs*)[3],[5] are improved BDDs. These representations have recently become popular because of their efficiency in terms of memory requirement and manipulation time.

In using BDDs, we have to define the order of the input variables of the Boolean functions to be represented. The forms of the BDD vary with the ordering for the same function. The size of the graphs greatly depends on the order. The variable ordering algorithm is one of the most important issues in the application of BDDs.

There are some works on the ordering. References (11) and (12) are the methods of finding the optimal order. However, it is still difficult to find the best order for larger scale functions. As another approach, Refs. (4), (13) and (3) show the heuristic methods using the topological information of logic circuits. Reference (6) uses testability measure for the heuristics, which reflect not only topological but logical information of the circuit. These heuristic methods can be applied to the practical benchmark circuits and can compute a good order in many cases. Their weak point is that the results depend on the structure of the circuits, and none of them are effective for universal circuits[6]. Reference (7) shows another approach that improves the order for the given BDD by repeating the exchange of the variables. It can give further better results than the initial BDDs, but sometimes it is trapped in local optimum.

In this paper, we propose a new method that finds an appropriate order for the given BDD by reordering the variables. Our method features that it evaluates the *width of the BDD*, which we defined. It can compute a good order independently of the circuit structure. Our experiments show that our method is much effective for many practical functions. In a reasonable amount of time, it gives good results independently of the initial variable orders.

2. Binary Decision Diagrams

We begin with an explanation of BDDs which are the ground of this paper.

2.1 Binary Decision Diagrams

A Binary Decision Diagram (BDD) is a directed graph representation of a Boolean function, as shown in Fig. 1(a). This graph 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, which produces the two subfunctions from a Boolean function by assigning 0 or 1 to an input variable.

The following reduction process gives a *Reduced Ordered BDD* (*ROBDD*) which represents a Boolean function uniquely. (see Ref. (2) for details).

1. Fixing the order of the input variables.

Manuscript received July 13, 1991.
Manuscript revised October 5, 1991.

† The author is with NTT LSI Laboratories, Atsugi-shi, 243-01 Japan.

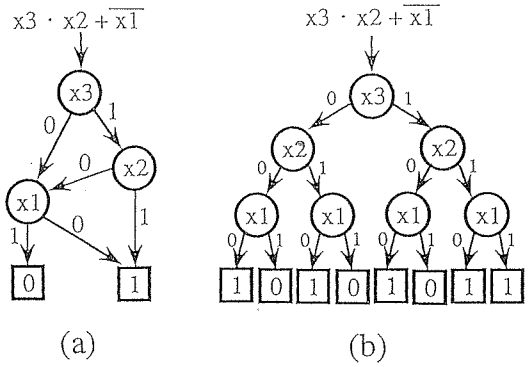

Fig. 1 (a) Binary Decision Diagrams (BDDs).
(b) Binary Decision Tree.

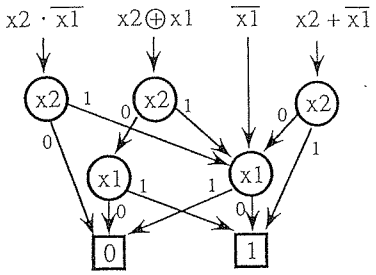

Fig. 2 Shared BDDs (SBDDs).

2. Eliminating all the redundant nodes whose two sub-graphs are identical.
3. Sharing all the equivalent sub-graphs.

ROBDDs have the following properties.

- Equivalence between two functions can be checked by an isomorphism check of those graphs.
- For many functions, the graph size are feasible for handling on a computer[8].
- Logic operation can be carried out within a time almost in proportional to the size of the graphs.

Resent research[9] indicating that ROBDDs enable the efficient execution of a new algorithm, which used to be considered impracticable in terms of the memory requirement and execution time, has contributed to the current popularity of BDD application.

2.2 Shared Binary Decision Diagrams

A set of ROBDDs representing multiple functions can be united into a single graph which consists of the ROBDDs sharing their sub-graphs with each other. The efficiency of manipulation can be improved by uniting all the ROBDDs into a single graph, as in Fig. 2, We call such graphs SBDDs (Shared BDDs)[3],[5].

With an SBDD, two isomorphic BDDs never coexist because they are completely shared. This property offers the following advantages.

- We can compactly represent many functions together.
- Equivalence between two functions can be checked by only comparing the pointer to the graph.

Generally, the graphs whose variable orders are not fixed or non-reduced graphs may be referred as BDDs. In the following section, for the sake of simplification, BDDs mean ROBDDs (and also SBDDs).

3. Variable Ordering on BDDs

As mentioned above, BDDs have some good properties; however, we have to fix the order of the input variables for the function to be represented in BDDs. Forms of BDDs may vary according to the order though they represent the same function. In many cases, the size greatly depends on the order. Since the size of the graphs decides not only memory requirement but execution time for the graph manipulation, it is very important to find a good variable order for the application of BDDs.

Fluctuations in the BDD size for various variable orders depend on the function to be handled. There are very sensitive examples whose BDD size vary extremely (exponentially to the number of inputs) by only reversing the variable order. On the other hand, the BDDs for symmetric functions never vary for any variable order. There are also an example that the multiplier functions[10] cannot be represented in a polynomial-sized BDD in any variable order.

For many functions which are often manipulated in practical LSI designs, there are good variable orders which make BDDs much smaller then using random orderings[3]. However, it is not easy to find an appropriate order automatically for any given function. If we can compute such order in a reasonable amount of time, BDDs become more practicable and can be used instead of *truth tables* or *cube sets*.

3.1 General Properties on the Variable Ordering

Empirically, the following properties are observed on the variable order for the reduction of BDDs[3].

- The group of the inputs with local computability should be near in the order. Namely, we should keep inputs near that are closely related with each other and that have some independence from the other inputs. For example, in the case of the n-bit adder functions, each pair of inputs with same figure should be next in the order.
- The inputs which greatly affect the output functions should be located at higher positions (near positions to the root) in the order. When the control inputs and the data inputs can be distinguished, the control inputs should be located higher than the data inputs.

If we could find a variable order which satisfies those two properties, the BDD size would not increase. However, the two properties are generally mixed ambiguously, so it is difficult to find always a good

order only by considering the two properties.

3. 2 The Approach to the Variable Ordering

At first, the variable ordering methods are classified as follows:

- To seek the best order, which gives the minimum size of BDD.
- To seek a quasi-minimum solution in a reasonable time.

Concerning the method to find the best order, there is an algorithm[11] of $O(n^2 3^n)$ time, where n is the number of inputs, that is based on the *dynamic programming*. However, it is still difficult to find the best order in a practical time for functions with many inputs, although this algorithm has been improved to the point where the best order can be found for some functions with 17 inputs[12]. In this paper, we discuss our method of finding an appropriate order for the larger scale functions in a practical time and space.

There are two approaches to seek a quasi-minimum solution:

- To find an appropriate order before generating a BDD by using logic circuit information which is the source of the Boolean function to be represented.
- To reduce the size of the graph by reordering the input variables on a given BDD in a certain variable order.

Some methods using the topological information of the circuit include ones where the circuit is traversed in a heuristic manner[13],[4], and another in which a weight is assigned to each net[3]. Reference (6) uses testability measure for the heuristics. This approach is one of the most effective ways at present and gives a good order in many cases. Nevertheless, depending on the structure of the circuits, this approach may not be effective. None of them are effective for universal circuits[6].

On the other hand, one of the disadvantages of the latter approach, which reorders on a given BDD, is that it cannot be executed if we cannot make an initial BDD of a reasonable size. In most of the application based on BDDs at present, especially in logic verification, the problems can be solved if the initial BDDs are generated. However, there are some applications, such as logic synthesis, that require many logic operations after generating of initial BDDs[9]. In such cases, variable reordering is an important technique because the efficiency of the programs is sensitive to the size o BDDs. It is useful especially when the heuristics with circuit information is not available or ineffective. We discuss this reordering approach later in this paper.

Next, there are two reordering algorithms to reduce BDDs:

- *Local search*: Repeating the swap of the variables if it improves the size of BDDs.
- *Greedy method*: Fixing the positions of the variables one by one, based on a certain cost function.

As a *local search* algorithm, there is a method which swaps the pairs of variables at the next position with each other. Some good results have been reported[7]. There is the interesting idea of swapping variables at random[12]. In any case, the local search greatly depends on the initial order. If the initial order is far from the best, many swaps are needed. This takes a long time, and there is the higher risk of being trapped in a bad local minimum solution. But at least, this algorithm never gives a result worse than the initial order, therefore, it is adequate for the last optimization in combination with other ordering methods.

The *greedy method* seeks a solution with a global view. Its good points include that it is robust to the variation of the initial order. An algorithm for this method has been proposed[14]. In greedy methods, the most important and difficult point is to find an effective cost function. In this paper, we propose a new ordering method using the width of BDDs, as a cost function.

4. Minimum-Width Method

We describe our ordering method in this section. In the following, n denotes the number of the input variables. Each variable is identified by an index number, as $x_1, x_2, \cdots, x_n$, where the variable with a larger index is located at a higher position (near to the root) in the BDDs.

4. 1 The Cost Function for the Greedy Method

Our method is based on the greedy method. At first, we choose one variable from among the all, and fix it at the highest position (x_n). Next, another variable is chosen from among the rest, and fixed at the second highest position (x_{n-1}). In this manner, all the variables are chosen one by one, and they are fixed from the highest to the lowest. This algorithm has no back tracking.

When choosing $x_k (1 \leqq k \leqq n)$, the variables with higher indexes than k have already been fixed, and the form of the higher part of the graph never varies. Namely, the choice of x_k affects only the part of the graph lower than x_k. The aim on each step is to choose x_k which minimizes the lower part of the graph.

However, it is dfficult to know how the lower part of the graph will be minimized, because the positions of the lower variables has not been fixed yet. To avoid back tracking, we decide to choose x_k by evaluating with a certain cost function.

It is desired that the cost function should give a good estimation of the minimum size of the graph for each choice of x_k. Furthermore, the cost function should be computable within a feasible time. We

propose the use of the *width of BDDs* as the cost function to satisfy those requirements.

4.2 The Width of BDDs

We define the width of BDDs here.

Definition: The *width of BDDs at height k*, denoted as $Width_k$, is the number of edges crossing the section of the graph between x_k and x_{k+1}, where the edges pointing to the same node are counted as one. The width between x_1 and the bottom of graph is denoted as $width_0$. □

An example is shown in Fig. 3. $width_k$ resembles to the number of the nodes with x_k; however, they are generally different. $width_k$ may be larger than the number of the x_k nodes because the width can count the edges which skips the node with x_k.

We present the following theorem on the width of BDDs.

Theorem: The $width_k$ is constant for any permutation among $\{x_1, x_2, \cdots, x_k\}$ and any permutation among $\{x_{k+1}, x_{k+2}, \cdots, x_n\}$.

Proof: Each edge crossing the section of the graph between x_{k+1} and x_k represents a sub-function obtained by assigning a vector of Boolean values $\{0, 1\}^{n-k}$ to the input variables along the path from the root node to that edge. BDDs have the property that the edges representing the same sub-function point to the same node. Therefore, $width_k$ means the number of all the sub-functions obtained by assigning all the patterns of the vector $\{0, 1\}^{n-k}$ into the input variables $\{x_{k+1}, x_{k+2}, \cdots, x_n\}$.

In assigning *all the patterns* of the vector into the variables, the order of the patterns does not affect the result. The permutation of the variables can be regarded as the change of the order of the patterns. That is to say, the $width_k$ is constant for any permutation among $\{x_{k+1}, x_{k+2}, \cdots, x_n\}$.

All the sub-functions, which are obtained by assigning all the patterns of the vector, are uniquely represented in BDDs with the same variable order of $\{x_1, x_2, \cdots, x_k\}$. For any permutation among these variables, the number of the sub-functions is constant because the sub-functions are still represented uniquely. Therefore, $width_k$ never varies for any permutation among $\{x_1, x_2, \cdots, x_k\}$. □

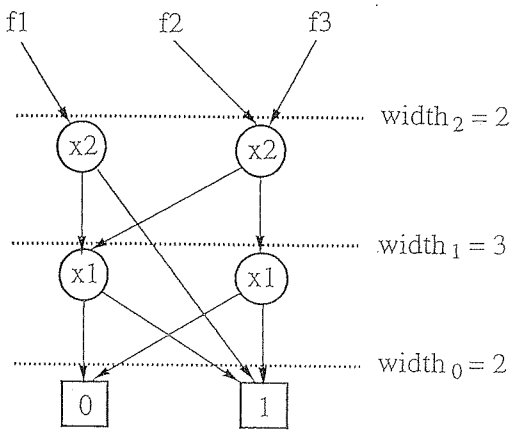

Fig. 3 The width of BDDs.

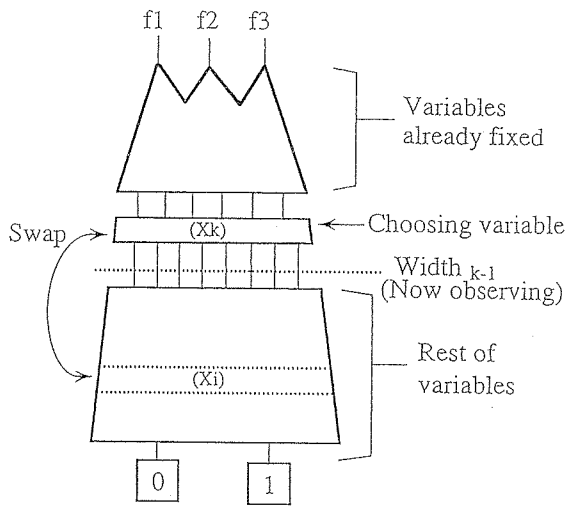

Fig. 4 Minimum-width method.

4.3 Minimum-Width Method

Our ordering method uses the width of the BDD as a cost function to estimate the complexity of the graph, and the variables are chosen one by one from the highest to the lowest by observing the cost function. Namely, we choose x_k which gives minimum $width_{k-1}$ among the rest of the variables, as shown in Fig. 4. We call this algorithm *the Minimum-Width Method.* If there are two candidates with the same $width_{k-1}$, the variable at the higher position in the initial order is chosen before the other.

The grounds for using the width of BDDs as a cost function are as follows.

- On choosing x_k; $width_{k-1}$ is independent from the order of the rest of the variables $x_1, x_2, \cdots, x_{k-1}$, as in the above theorem. Therefore, it is robust for the variation of the initial order.
- It is clear that we should avoid to choose x_k that makes $width_{k-1}$ large because $width_{k-1}$ is a lower bound of the number of the nodes in the part of the graph lower than x_k.
- It is not difficult to compute $width_k$.

For these reasons, it is effective to use the width of BDDs as the cost function.

The time complexity of our method is $O(n^2G)$, where G is the average size of the graph since the size varies during the ordering. This complexity is considerably less than the conventional algorithms which seeks the best order.

5. Implementation of Minimum-Width Method

In this section, we present the techniques of im-

plementing the Minimum-Width Method efficiently.

5. 1 Swapping of Variables

Basically, there are two methods of modifying BDDs by permuting the variable order. They are

- deleting the initial BDDs, and regenerating new BDDs in a new order,
- and applying the logic operations of swapping variables on the initial BDDs.

When all the variables are reordered at one time, the former method is more efficient. In this paper, we use the latter method because the Minimum-Width Method repeats partial permutation of the variable order.

Any permutation is carried out by combining the swap of a pair of variables. Now, we consider the realization of swapping of two variables using logic operations.

We describe here the sub-functions obtained by assigning a value of (0, 1) into the two variables x_i and x_j for a Boolean function f.

$x_i=0 \quad x_j=0$: f_{00}

$x_i=0 \quad x_j=1$: f_{01}

$x_i=1 \quad x_j=0$: f_{10}

$x_i=1 \quad x_j=1$: f_{11}

In the above case, we can swap x_i and x_j by swapping only f_{01} and f_{10}. We do not have to move f_{00} and f_{11}. The swapping can be completed by reconstructing these swapped sub-functions. Namely, the swap operation is represented as:

$$f_{\text{swap}}=\bar{x}_i\ \bar{x}_j f_{00}+\bar{x}_i x_j f_{10}+x_i\bar{x}_j f_{01}+x_i x_j f_{11}.$$

This operation requires no traverse on the part of the graph lower than x_i and x_j. The operation time is in proportion to the number of the nodes at positions higher then x_i and x_j. Therefore, the higher variable can be swapped more quickly.

5. 2 Computing of the Width of BDDs

The $width_k$ can be computed by counting the number of elements in the set S_k, where S_k consists of all the sub-functions obtained by assigning any combination of value (0, 1) to the input variables at positions higher than x_k. In our implementation, each sub-function is identified by a 1-word index. In counting procedure, we enter the indexes which have already counted into a hash table to check equivalent sub-functions. The $width_k$ can be computed in a time that is proportional to the number of the nodes at the positions higher than k.

In the Minimum-Width Method, we compute the $width_k$ for each k from $k=n-1$ to $k=0$. In this case, we can compute $width_{k-1}$ using the result of counting $width_k$. When we store the set of sub-function S_k, which has been generated in computing $width_k$, the next set S_{k-1} can be obtained by only assigning $x_k=0$ and $x_k=1$ for S_k.

In addition, if we assign $x_i=0$ and $x_i=1$ $(1\leqq i\leqq k)$ for S_k, the next set S_{k-1} represents the sub-functions when we choose x_i as the variable at the k-th highest position in the order. Therefore, we do not have to repeat swapping between x_k and x_i for all $(1\leqq i\leqq k)$ in choosing x_k.

5. 3 Management of Attributed Edges

In our BDD manipulator, we have implemented *attributed edges*[3] in order to reduce the storage and the time. They are the edges attached some sorts of attributes; each attribute represents a certain operations. Three attributes are proposed:

- *Output inverters*, to complement an output function.
- *Input inverters*, to complement an input variable.
- *Variable Shifters*, to shift the index numbers of the input variables.

Here, we present how the attributed edges are managed in our ordering method.

In using the attributed edges, the property is kept that a BDD represents a Boolean function uniquely, although the size of graphs and execution time can be different. If we apply the Minimum-Width Method to the BDDs with the attributed edges, the same order should be computed as to the original BDDs, because the Minimum-Width Method is based on logic operations. The order are computed for the reduction of the original BDDs; however, empirically the results are also effective for BDDs with the attributed edges.

The ordering method can be improved by considering the affect of the attributed edges. In using attributed edges, there are the edges pointing to the same node but whose attributes are different. Since these edges represent different sub-functions, they were counted as different ones in the computing of the width of BDDs in original method. Regarding such edges as identity to be counted one, we can obtain the order for the BDDs with attributed edges. In our implementations, only the effect of *output inverters* are considered.

5. 4 Preventing Temporary Increases in Graph Size

Our method repeats the swap of two variables to approach an appropriate order. During the ordering, the size of graph increases temporarily. Some increase may be unavoidable since we may not be able to reduce BDDs monotonously; however, the increase cannot be overlooked if it becomes too large.

For example, when x_i is chosen for the k-th highest variable, we simply swap x_i and x_k. Repeating

this procedure, the rest of the variables may be shuffled at random and the graph size increases temporarily. In the worst case, this random shuffling causes memory overflow in the ordering halfway through, in spite of the fact that the final result is a smaller graph. We had better not to change the order of the rest of the variables during the ordering process.

There are two devices. The first one is to choose the variable at a higher position in the initial order when there are two candidates with the same $width_{k-1}$, as written in the preceding section. This does not modify the part of the graph lower than the chosen variable, therefore, the risk of the memory overflow is lessened.

The other device involves the manner of the swapping. In our first implementation, we fixed the variable one by one from the highest to the lowest as x_n, x_{n-1}, $\cdots$, x_1. In this way, the variables of the former occupants were forced to move to a lower position. This is one of the reasons that the rest of the variables are shuffled. To avoid this, we move the chosen variables to x_{2n}, x_{2n-1}, $\cdots$, x_{n+1}, which are new variables higher than x_n, so that the rest of the variables do not have to move. This device enables us to avoid the meaningless shuffles of the variables. After moving all the variables, the indexes have to be revised by subtracting n. The revision can be executed easily if the *variable shifters*[3] are implemented.

6. Experiments

We implemented our ordering method as shown above, and conducted some experiments for an evaluation. We used a SPARC Station 2 (SunOS 4.1.1, 32 M Byte). The program is described in C andC++. The memory requirement of BDDs is about 21 Bytes a node.

6. 1 Experimental Results

In our experiments, we generated initial BDDs for given logic circuits in a certain order of the variables and applied our ordering method to the initial BDDs. We use the three kinds of the attributed edges. Our ordering method works on the BDDs with the three attributed edges, but it considers only the effect of *output inverters*. There are no serious differences of the performance empirically.

The results for some examples are summarized in Table 1. In this table, *dec8* is the output function of an 8-bit data selector, and the function *enc8* is an 8-bit priority encoder. *adder8* is an 8+8 bit adder, and *mult6* is a 6×6 bit multiplier. The other items were chosen from the benchmark circuits in DAC'86[15]. These circuits generally have multiple outputs. Our program can handle multiple output functions using Shared BDDs.

In this experiment, the initial order is important.

Table 1 Experimental results.

func.	in.	out.	#node: ave.(min.-max.) init.		after		time (sec)
dec8	12	2	75.2	(19-204)	17.8	(16-20)	0.09
enc8	9	4	25.2	(23-28)	19.9	(19-22)	0.05
adder8	17	9	543.1	(337-938)	40.0	(40-40)	0.53
mult6	6	6	2803.5	(2382-3209)	2145.9	(2123-2296)	6.63
5xpl	7	10	63.9	(58-72)	36.0	(36-36)	0.20
9sym	9	1	23.0	(23-23)	23.0	(23-23)	0.25
alupla	25	5	8101.7	(4178-12952)	1055.0	(856-1178)	33.21
vg2	25	8	861.2	(562-1688)	84.2	(81-87)	1.80

Table 2 Experiments on large scale examples.

func.	in.	out.	#node init.	#node after	time (sec)
c432	36	7	23290	1383	177.5
c499	41	32	29702	21962	1311.8
c880	60	26	19100	18336	721.1
c1908	33	25	11083	6590	239.1
c3540	50	22	214941	33975	7493.9
c5315	178	123	27958	2066	14548.3

We generated 10 initial BDDs in random orders, and applied our ordering method in each case. The table shows the maximum, minimum, and average number of the nodes before and after ordering. *CPU time* is the average time of ordering for 10 cases (including the time of generating initial BDDs).

The results show that our method can reduce the size of BDDs remarkably for the most examples, except for *9sym*. (This is natural because *9sym* is a symmetric function.) Note that for the various initial order, the fluctuation of the size after ordering is comparably smaller than the initial BDDs. This shows that our Minimum-Width Method gives a good solution independently of the initial order. This solution can act as the guiding values for the BDDs for which the best order has not been found, and it is useful in the evaluation of other ordering methods. In terms of speed, it takes a reasonable time for a BDD with scores of inputs and thousands of nodes.

Next, the similar experiments were conducted for the larger examples. The functions were chosen from the benchmark circuits in ISCAS'85[16]. On these benchmark circuits, there are several reports of the heuristic ordering method using the circuit information[4],[13],[3],[6]. We cannot exactly compare these heuristics because the manners of the experiments are different. (Someone counts the maximum size of BDDs in the operations, others count the size of the Shared BDD of all output functions or all functions which includes internal nets.) However, Butler[6] concludes that none of them are effective for universal circuits.

In our experiments, the initial orders are given by *the Dynamic Weight Assign Method*[3], which is one of the heuristic methods using the circuit information. These benchmark circuits are too complex to generate

initial BDDs in a random order.

The results are shown in Table 2. Our method is also effective for large scale functions in terms of graph reduction, thought it takes longer time (but much faster than the methods which seeks best order). The sizes of the BDDs after reordering are almost equal to the heuristic methods[4],[13],[3],[6] which use circuit information, and our method may be more generally effective for all the circuits. Remarkably, we find that *c5315* can be represented in only about 2000 nodes, which is far less than the results by any oter method (as about 18000 nodes[17]). Our results are useful to evaluate other heuristic methods of variable ordering.

The weak points of our method include that it takes longer time than the heuristic methods using the circuit information and that it requires a certain initial BDDs. However, we can say that it is effective to the applications which have many logic operations after generating of BDDs.

6. 2 Comparison and Combination with the Local Search Method

We did another experiment to compare the properties of the Minimum-Width Method and the local search method, which was presented in the preceding section. We implemented a local search method of variable ordering which swaps the pairs of variables on the next position with each other if the swap reduces the size of the graph.

For the three examples, we applied both ordering methods to the BDDs with various initial orders, that include the best and worst orders in our knowledge, and average size for 10 random orders, as shown in Table 3. It shows that the two ordering methods have complementary properties. The local search never gives worse results than the initial order, but the effect greatly depends on the initial order. On the other hand, the Minimum-Width Method does not guarantee obtaining a better result than the initial order; however, the result is always close to the best solution.

These properties lead us to conclude that it is effective to apply the Minimum-Width Method at first because it seeks a good order with a global view, and then to apply the local search for the least optimization. The results of our experiments with such combination, which are summarized in Table 3, show that the combination is more effective than applying either of the two methods.

Table 3 Comparison with the local search.

func.	init.#node	Min-Width	local search	combination
dec8 (best)	8	12	8	10
(worst)	382	17	38	11
(random)	75.2	17.8	19.7	10.1
adder8 (best)	40	40	40	40
(worst)	1139	40	178	40
(random)	543.1	40.0	182.9	40.0
alupla (best)	830	969	830	830
(worst)	12968	979	2835	830
(random)	8101.7	1055.0	2311.5	998.0

6. 3 An Improvement with Threshold

The Minimum-Width Method may give an order worse than the initial BDD, when the initial variable order is already nearly the best. This property is not good for the practical use, and in the following, we consider how to prevent this increase in graph size.

In our method, the width of the BDD shows a lower bound of the number of the nodes in the part of the graph which has not been fixed yet. It is clearly unfavorable to choose variables which make the lower bound large because the chance of reducing the graph size may be lost. The important point is that we should not choose variables which make the lower bound large but that it is not important to actually seek the minimum lower bound itself. If we repeat the swap of variables based on the small differences in the widths, there is no great effect and the order may be shuffled thoughtlessly.

We made a further improvement in which a swap of the variables is canceled if the difference of the widths is less than a threshold value. Precisely speaking, we do not swap the variables when the reduction ratio of the minimum width to the width of the former occupant variable is not greater than the threshold value.

When the initial order is nearly the best, some swaps are canceled since the fluctuation of the width is smaller than the threshold, and consequently the initial order is protected. If the inital order is far from the best order, the fluctuation of the width become greater than the threshold and the swap is executed as described previously.

To evaluate this improvement, we experimented with three threshold values as 5%, 7%, and 10%. The initial orders are almost good ones computed by the Dynamic Weight Assign Method. The results are shown in Table 4, they indicate that this improvement is effective to a degree. Too small a threshold cannot give a remarkable effect, but one that is too large cancels the swaps necessary to reduce the graph. The appropriate value of the threshold is about 5% or 10%, and may depend on the kind of the function.

Table 4 Improvement with threshold.

func.	#node				
	init.	0%	5%	7%	10%
c432	23290	1383	1383	1353	1301
c499	29702	21962	22562	22434	34074
c880	19100	18336	19254	8930	8930
c1908	11083	6590	5246	5246	5202

7. Conclusion

We have shown a method of variable ordering for the reduction of BDD size. Our method finds an appropriate order using no additional information, such as the topological information of the circuit. It reduces the size of the graph by reordering the input variables on a given BDD with a certain variable order.

This method has the good property of giving a quasi-minimum solution independently of the initial order in a reasonable time. The results can be measures for the evaluation of other ordering methods. The method can reduce some large scale BDDs in a practical time. We showed the effect of combining our method with the local search method, and also described the improvement using the threshold.

The weak point of our method is that it cannot be executed if we cannot generate a feasible size of BDD in the initial order. When we apply the method to some very large scale BDDs, initial BDDs cannot be generated because of memory overflow. Future work remains to develop a method which can compute a good order concurrently with generating BDDs.

Acknowledgement

The author would like to express his appreciation to Mr. Adachi and Mr. Endo of NTT LSI laboratories for their encouragement. The author also thanks the members of Professor Yajima's research laboratory of Kyoto University for fruitful discussions.

References

(1) Akers S. B.: "Binary Decision Diagrams", IEEE Trans. Comput., pp. 509-516(1978).
(2) Bryant R. E.: "Graph-Based Algorithms for Boolean Function Manipulation", IEEE Trans. Comput., pp. 677-691(1986).
(3) Minato S., Ishiura N. and Yajima S.: "Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation", ACM/IEEE Proc. 27th DAC, pp. 52-57(1990).
(4) Malik S., Wang A. R., Brayton R. K. and S.-Vincentelli A.: "Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment", Proc. ICCAD'88, pp. 6-9(1988).
(5) Brace K. S., Rudell R. L. and Bryant R. E.: "Efficient Implementation of a BDD Package", ACM/IEEE Proc. 27th DAC, pp. 40-45(1990).
(6) Butler K. M., Ross D. E., Kapur R. and Mercer M. R.: "Heuristics to Compute Variable Orderings for Efficient Manipulation of Ordered Binary Decision Diagrams", ACM/IEEE Proc. 28th DAC, pp. 417-420(1991).
(7) Fujita M., Matsunaga Y. and Kakuda T.: "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).
(8) Yajima S. and Ishiura N.: "A Class of Logic Functions Expressible by a Polynomial-Size Binary Decision Diagrams", Proc. Synthesis and Simulation Meeting and Int. Interchange (SASIMI'90) (1990).
(9) Matsunaga Y. and Fujita M.: "Multi-level Logic Optimization Using Binary Decision Deagrams", Proc. ICCAD'89, pp. 556-559(1989).
(10) Bryant R. E.: "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication", IEEE Trans. on Computers, **40**, 2(1991).
(11) Friedman S. J. and Spowit K. J.: "Finding the Optimal Variable Ordering for Binary Decision Diagrams", ACM/IEEE Proc. 24th DAC, pp. 348-356(1987).
(12) Ishiura N., Sawada H. and Yajima S.: "Minimization of Binary Decision Diagrams Based on Exchanges of Variables", Proc. ICCAD'91, pp. 472-475(1991).
(13) Fujita M., Fujisawa H. and Kawato N.: "Evaluations and Improvements of a Boolean Comparison Method Based on Binary Decision Diagrams", Proc. ICCAD'88, pp. 2-5 (1988).
(14) Aborhey S.: "Binary decision graph reduction", IEE Proc., **136**, Pt.E, 4(1989).
(15) de Geus A. J.: "Logic Synthesis and Optimization Benchmarks for the 1986 DAC", ACM/IEEE Proc., 23rd DAC, p. 78(1986).
(16) Brglez F. and Fujiwara H.: "A Neutral Netlist of 10 Combinational Circuits, Special Session on ATPG and Fault Simulation", IEEE Proc., ISCAS'85(1985).
(17) Fujita M., Matsunaga Y. and Kakuda T.: "Multi-level Logic Minimization with Binary Decision Diagrams for Large Circuits", Record of 38th IPS Japan National Convention, 4N-5, pp. 6.9-6.10(1990).

Shin-ichi Minato was born in Ishikawa, Japan, on August 30, 1965. He received the B. E. and M. E. degrees in Information Science from Kyoto University, Japan in 1988 and 1990, respecvively. Since joining NTT LSI Laboratories, Kanagawa, Japan in 1990, he has been working on the research of logic design systems. His current interest is in the representation and manipulation of Boolean functions, and logic synthesis systems.