# Dynamic Variable Ordering for Ordered Binary Decision Diagrams

Richard Rudell
Synopsys, Inc.
700 E. Middlefield Road
Mountain View, California 94043

## Abstract

*The Ordered Binary Decision Diagram (*OBDD*) has proven useful in many applications as an efficient data structure for representing and manipulating Boolean functions. A serious drawback of* OBDD*'s is the need for application-specific heuristic algorithms to order the variables before processing. Further, for many problem instances in logic synthesis, the heuristic ordering algorithms which have been proposed are insufficient to allow* OBDD *operations to complete within a limited amount of memory. In this paper, I propose a solution to these problems based on having the* OBDD *package itself determine and maintain the variable order. This is done by periodically applying a minimization algorithm to reorder the variables of the* OBDD *to reduce its size. A new* OBDD *minimization algorithm, called the sifting algorithm, is proposed and appears especially effective in reducing the size of the* OBDD*. Experiments with dynamic variable ordering on the problem of forming the* OBDD*'s for the primary outputs of a combinational circuit show that many computations complete using dynamic variable ordering when the same computation fails otherwise.*

## 1 Introduction

Boolean function manipulation is an important component of many logic synthesis algorithms including logic optimization and logic verification of combinational and sequential circuits. The Ordered Binary Decision Diagram (OBDD) has proven useful in these applications as an efficient data structure for the representation and manipulation of Boolean functions. However, a serious drawback of OBDD's is the need to order the variables.

When an order is found which keeps the OBDD size manageable (e.g., less than 100,000 nodes), OBDD-based techniques perform very well. However, it is usually necessary to devise heuristics to order the variables for each OBDD application. Besides the burden this places on the programmer trying to apply OBDD's in a particular setting, there is the problem that in many instances the heuristic algorithms which have been proposed are unable to find a variable order which keeps the OBDD's small. This implies that many OBDD applications give-up (*space-out*) before an operation can be completed.

Many OBDD applications in logic synthesis begin by forming the OBDD for each primary output in a combinational circuit (in terms of the primary inputs to the circuit). For many problem instances, choosing a random order for the variables leads to OBDD's which are too large. For example, it is not possible to form the OBDD's for 23 of 35 large circuits from the IWLS'91 benchmark set when using a random variable order. Heuristic ordering algorithms, such as the *depth-first heuristic* algorithm and its variations ([10, 5, 11]), are a significant improvement over random variable ordering. However, it is still not possible to form the OBDD's for 11 of 35 large circuits from the IWLS'91 benchmark set when using this heuristic order. It has remained an open problem whether this is a limitation of the variable ordering algorithms or simply the inherent exponential worst case complexity of the OBDD representation.

This paper describes a general paradigm to improve the robustness of any OBDD package by using automatic variable re-ordering. First I review the necessary details of an OBDD package and then describe the dynamic variable ordering strategy. Two OBDD minimization algorithms are presented, including a new algorithm called the sifting algorithm. Experimental results are given for these algorithms which demonstrate the utility of dynamic variable ordering when applied to the problem of forming OBDD's for the primary outputs in a combinational logic network. The last section provides directions for future improvements of this technique.

## 2 OBDD Implementation Review

I assume that the reader is familiar with Ordered Binary Decision Diagrams as introduced by Bryant [3]. In the paper by Brace, Rudell and Bryant [2], details for an efficient implementation of a OBDD package were outlined. I review here some details necessary for the remainder of the paper.

A multi-rooted (shared) directed acyclic graph (DAG) is used to represent a set of Boolean functions. Each node in the DAG represents a Boolean function $F$ and has an associated variable $x_i$ and pointers to two other nodes (functions) in the DAG. The node $F$ is written as the tuple $(x_i, G, H)$ where $x_i$ is called the *top variable* of the function $F$, $G$ is the positive cofactor of $F$ with respect to $x_i$ ($G = F_{x_i}$), and $H$ is the negative cofactor of $F$ with respect to $x_i$ ($H = F_{\bar{x}_i}$). The node $F$ thus represents the function $F = x_iG + \bar{x}_iH$. $G$ is also known as the THEN node and $H$ is also known as the ELSE node. The sink nodes represent the constant functions **0** and **1**.

Ordered BDD's have a total order imposed on the variables; i.e., an order is assigned to each variable, and the variables must appear in ascending order along every path in the OBDD. Because the BDD is ordered, the DAG can be levelized with all nodes with a particular top variable at a given level. Level $i$ refers to all nodes with a top variable $x_i$.

A global hash table, called the *unique table*, allows a node of the DAG, $(x_i, G, H)$, to be found in constant time. A hash function is computed on the tuple $(x_i, G, H)$ which provides an index into an array of bins which store the first DAG node for that hash value. All of the nodes with the same hash value (i.e., collisions) are stored in a linked list. Each node of the DAG occupies 4 words: the variable index plus other flags, a pointer to the THEN node, a pointer to the ELSE node, and a pointer to the next node on the collision chain for the unique table.

A global cache, called the *computed table*, is used as a memory function for the recursive algorithms which operate on the DAG. This table, implemented as a hash-based cache, stores the results of recursive operations such as ITE, but overwrites an entry when a collision occurs, rather than using a link-chain to resolve collisions. Each hash-based cache entry occupies 4 words: 3 words which form the key for the operation (e.g., ITE(F,G,H) uses F, G, and H as the key) and a single word which is the operation result. A ratio of one cache entry is maintained for every four unique table entries so that the total memory usage of the package, including all overhead, is approximately 24 bytes per DAG node on a 32-bit machine.

The OBDD package uses garbage collection to recycle memory. A reference count is maintained for each node in the DAG, and a count of the number of dead (i.e., unreferenced) nodes in the DAG is maintained as nodes are created and freed. Dead nodes cannot be freed immediately because an entry from the computed table may point to the node; these references are not included in the reference count because the computed table entries are never deleted. During a recursive operation such as ITE, a *find or add* operation is performed to either find a node in the DAG or to create a new node if the given node does not exist. If a new node is created causing the unique table to become too full, then either a garbage collection is performed if there are enough dead nodes in the DAG to make it worthwhile, or the unique table array is increased in size by a factor of two.

The key drawback to this package, which this paper addresses, is that the ordering of the variables is specified in advance by the user. No assistance is offered to help order the variables, and the order cannot be subsequently changed.

## 3 Dynamic Variable Ordering

In this paper, I propose a general paradigm for maintaining variable orders in an OBDD. The idea is to have the OBDD package determine and maintain the variable order of the OBDD. This variable order is changed automatically by the OBDD package, transparently to the user, as operations are performed. Because the variable order within the OBDD is no longer static, this technique is referred to as **dynamic variable ordering**.

Dynamic variable ordering differs from the typical use of OBDD's where the variables are ordered once when the OBDD is created and the order is maintained throughout all subsequent processing. It also differs slightly from other proposed variable ordering schemes in that the re-ordering is not performed at the explicit request of the user. Instead, the package determines appropriate points at which to stop processing, choose a new order, and then resume processing.

When using dynamic variable ordering, a total order is defined for all variables before and after each package operation; however, the order is periodically adjusted by the OBDD package, as a consequence of an operation, to find a better order. Logically, the variable order changes in-between package operations. Thus, we maintain all advantages provided by the ordered BDD data structure, such as canonicity and efficient recursive algorithms.

A well-designed interface to an OBDD package hides all details of the OBDD data structure. The programmer simply creates the variables and then uses package operations such as AND, OR, and NOT to form new functions. This allows dynamic variable ordering to be applied transparently to the user of the OBDD package.

There are two goals we hope to achieve with dynamic variable ordering. The first goal is to allow OBDD operation sequences which fail when using a fixed heuristic variable order to succeed when a new order is chosen mid-stream. The second goal is to reduce the need for the heuristic ordering algorithms – i.e., problems which complete with a heuristic variable order should also complete when starting from a random variable order. If we can deliver on these goals, the advantage of dynamic variable ordering to the user of the OBDD package is clear.

One implementation of dynamic variable ordering is as follows. At each garbage collection within the OBDD package, which is triggered based on the growth of the number of nodes in the DAG, a variable-reordering algorithm is applied to the OBDD to reduce the OBDD size. Any variable ordering algorithm can be applied at this step, but the algorithms which are used must be efficient because they will be applied repeatedly as OBDD processing proceeds. Of course, the algorithms must also be effective in finding a better variable order for the OBDD.

Dynamic variable ordering motivates the exploration of algorithms for OBDD minimization; i.e., reducing the size of all functions simultaneously represented by a multi-rooted OBDD by changing the variable order. Two algorithms for OBDD minimization are considered in the next section.

## 4 Variable Reordering Algorithms

Many people, including Brace [1], Fujita et al. [6], and Ishiura et al. [7] have made the observation that swapping the order of two adjacent variables in an

OBDD affects only the DAG nodes at the two levels; all other nodes remain unchanged. In this section, I describe how to implement this operation so that its complexity is proportional to the number of nodes at the particular level of the DAG and independent of the size of the entire DAG. This efficient adjacent variable swap forms the core for many OBDD minimization algorithms. I then describe the *window permutation algorithm* and the *sifting algorithm* for minimizing the size of the OBDD.

## 4.1 Efficient Variable Swap

There are two problems with making the variable swap of $x_i$ and $x_{i+1}$ have local complexity. The first is that we need to find all nodes at level $i$ without walking the entire DAG starting from the roots. The second is that each node in the DAG must represent the same function before and after the variable swap to avoid patching any references to that node.

A memory-efficient scheme to find all nodes at level $i$ replaces the single unique table with an array of hash tables, one per level of the DAG. The variable index is used to locate the hash table which stores all nodes for that level of the DAG. The hash table has an array of bins which store the first node for each hash value for this level. Hence, all nodes at a given level can be visited by walking the collision chain which starts at each hash table array position.

A node $F$ at level $i$ can be pointed to by other nodes above it in the DAG, and by functions which have already been returned to the user. To reduce memory, back-pointers are not maintained. Hence, there is no way to reach all references to node $F$ without walking the entire DAG. Therefore, to perform a local variable swap, it is necessary to maintain an identical logical function at each node. This is done by overwriting the node representing $F$ with the new node which results from the modification to the variable order. This is done as follows.

Let $F = (x_i, F_1, F_0)$ be a node at level $i$. Let $F_{11}$ be the cofactor of $F_1$ with respect to $x_{i+1}$. Computing this cofactor is trivial: the result is either the THEN node pointed to by $F_1$ (if $x_{i+1}$ is the top variable of $F_1$) or $F_1$ (otherwise). Similarly, let $F_{10}$ be the negative cofactor of $F_1$, and let $F_{01}$, $F_{00}$ be the two cofactors of $F_0$. Node $F$ is overwritten with the tuple $(x_{i+1}, (x_i, F_{11}, F_{01}), (x_i, F_{10}, F_{00}))$. Expansion of this formula shows that it preserves the function of node $F$ and inspection ensures that the new variable order (i.e., $x_{i+1}$ is above $x_i$) is established for all paths through $F$.

The new nodes required at level $i$ (i.e., $(x_i, F_{11}, F_{01})$ and $(x_i, F_{10}, F_{00})$) may be degenerate nodes (e.g., in the case that $F_{11} = F_{01}$), or may already exist in the DAG as required to implement other functions. When $F$ is re-expressed as a result of the variable swap, the DAG's rooted at $F_1$ and $F_0$ can be freed. Note, however, that the nodes $F_{00}$, $F_{01}$, $F_{10}$, $F_{11}$ all have references after the variable swap, so that only the root nodes $F_1$ and $F_0$ can be freed as a result of the swap. To be specific, node $F_1$ can be freed if the only reference to $F_1$ previously came from node $F$.

We can make use of this observation to perform incremental garbage collection during the variable swap. Before OBDD minimization is applied, a garbage collection is performed and the computed table is cleared. Thereafter, nodes at level $i+1$ can be deleted incrementally if they have no other reference beside the reference from level $i$.

Attributed edges have been proposed by, among others, Karplus [8], Madre and Billon [9], and Minato et al. [11]. The edges in the OBDD are tagged to indicate a modification of the referenced function. This reduces the size of the DAG by allowing a single node to represent several different functions. The most popular and useful attribute is the *negate-output edge*, although other attributes, such as *negate-input edge* [11] and *negate-else edge* [4] have been proposed. The inclusion of attributed edges is usually transparent to the algorithms which operate on the OBDD. In particular, the complexity of an adjacent variable swap is unaffected by the inclusion of negate-output edges. However, inclusion of either negate-input or negate-else edges appears to destroy the local complexity of a variable swap by requiring pointers to a modified node to be changed. (More details can be found in [12].) One impact of dynamic variable ordering, then, is that these last two edge attributes cannot be used.

## 4.2 Window Permutation Algorithm

Fujita et al. [6] and Ishiura et al. [7] presented similar heuristic algorithms for minimizing the size of a OBDD using adjacent variable exchange. I refer to this algorithm as the *window permutation algorithm*.

The window permutation algorithm proceeds by choosing a level $i$ in the DAG and exhaustively searching all $k!$ permutations of the $k$ adjacent variables starting at level $i$. This is done using $k! - 1$ pairwise exchanges followed by up to $k(k-1)/2$ pairwise exchanges to restore the best permutation seen. This is then repeated starting from each level until no improvement in the DAG size is seen. Figure 1 shows the variable permutations which are explored when applying a window of size $k = 3$ starting at variable $x_2$. Six permutations are explored with 5 adjacent variable swaps, and then 3 additional variable swaps (worst case) are used to restore the best permutation.

| | |
|---|---|
| $x_1, x_2, x_3, x_4, x_5, x_6, x_7$ | initial |
| $x_1, x_3, x_2, x_4, x_5, x_6, x_7$ | swap $(x_2, x_3)$ |
| $x_1, x_3, x_4, x_2, x_5, x_6, x_7$ | swap $(x_2, x_4)$ |
| $x_1, x_4, x_3, x_2, x_5, x_6, x_7$ | swap $(x_3, x_4)$ |
| $x_1, x_4, x_2, x_3, x_5, x_6, x_7$ | swap $(x_3, x_2)$ |
| $x_1, x_2, x_4, x_3, x_5, x_6, x_7$ | swap $(x_4, x_2)$ |
| $x_1, x_2, x_3, x_4, x_5, x_6, x_7$ | swap $(x_4, x_3)$ |
| $x_1, x_3, x_2, x_4, x_5, x_6, x_7$ | swap $(x_2, x_3)$ |
| $x_1, x_3, x_4, x_2, x_5, x_6, x_7$ | swap $(x_2, x_4)$ |

Figure 1: Window Permutation Example.

Marking can be used to record when a variable exchange at a given level may be profitable. A level is marked after the permutation at the level is known optimal. This mark is reset when a new permutation is determined for any of the preceding $k-1$ levels; when all levels in the DAG are marked, the window permutation algorithm cannot further improve the DAG size).

Because the swap of two adjacent variables is efficient, the window permutation algorithm remains practical for values of $k$ as large as 4 or 5. However, results presented in Section 5.2 and Section 5.3 indicate that the window permutation algorithm is limited in its ability to find good variable orders.

### 4.3 Sifting Algorithm

I propose a new OBDD minimization algorithm in this paper which I call the *sifting algorithm.* This algorithm is based on finding the optimum position for a variable, assuming all other variables remain fixed. If there are $n$ variables in the DAG (excluding the constant level which is always at the bottom), then there are $n$ potential positions for a variable, including its current position. Among these $n$ positions, the subgoal employed by the sifting algorithm is to find the spot which minimizes the size of the DAG.

Ideally, we could find the best position for a variable assuming all other variables remain fixed with a low-complexity analysis of the OBDD. However, this does not appear possible. Therefore, the optimum position for a variable is determined by brute-force enumeration as follows. The variable is exchanged with its successor variable until the variable becomes the next to last variable in the DAG; i.e., the variable is sifted down to the bottom of the DAG. Then the variable is exchanged with its predecessor variable until the variable becomes the top variable in the DAG; i.e., the variable is sifted up to the top of the DAG. The best DAG size seen during this search is remembered and the position of the variable is restored by moving the variable from the top position down to its optimum position. Figure 2 shows the variable permutations which are explored when applying the sifting algorithm to variable $x_4$. The 7 positions for variable $x_4$ are explored using 9 adjacent swaps, and the optimum position is restored with an additional 6 swaps (worst-case).

| | |
|---|---|
| $x_1, x_2, x_3, x_4, x_5, x_6, x_7$ | initial |
| $x_1, x_2, x_3, x_5, x_4, x_6, x_7$ | swap $(x_4, x_5)$ |
| $x_1, x_2, x_3, x_5, x_6, x_4, x_7$ | swap $(x_4, x_6)$ |
| $x_1, x_2, x_3, x_5, x_6, x_7, x_4$ | swap $(x_4, x_7)$ |
| $x_1, x_2, x_3, x_5, x_6, x_4, x_7$ | swap $(x_7, x_4)$ |
| $x_1, x_2, x_3, x_5, x_4, x_6, x_7$ | swap $(x_6, x_4)$ |
| $x_1, x_2, x_3, x_4, x_5, x_6, x_7$ | swap $(x_5, x_4)$ |
| $x_1, x_2, x_4, x_3, x_5, x_6, x_7$ | swap $(x_3, x_4)$ |
| $x_1, x_4, x_2, x_3, x_5, x_6, x_7$ | swap $(x_2, x_4)$ |
| $x_4, x_1, x_2, x_3, x_5, x_6, x_7$ | swap $(x_1, x_4)$ |
| $x_1, x_4, x_2, x_3, x_5, x_6, x_7$ | swap $(x_4, x_1)$ |
| $x_1, x_2, x_4, x_3, x_5, x_6, x_7$ | swap $(x_4, x_2)$ |
| $x_1, x_2, x_3, x_4, x_5, x_6, x_7$ | swap $(x_4, x_3)$ |
| $x_1, x_2, x_3, x_5, x_4, x_6, x_7$ | swap $(x_4, x_5)$ |
| $x_1, x_2, x_3, x_5, x_6, x_4, x_7$ | swap $(x_4, x_6)$ |
| $x_1, x_2, x_3, x_5, x_6, x_7, x_4$ | swap $(x_4, x_7)$ |

Figure 2: Sifting Algorithm Example.

The sifting algorithm proceeds as follows. The variables are sorted into decreasing size based on the number of nodes at each level of the DAG. Then each variable is moved to its locally optimum position assuming that all other variables remain fixed. Each variable is moved only once in this process, although the algorithm could be iterated to convergence.

The sift algorithm has the advantage that a variable can move a long distance in the ordering. Note that the DAG-size can increase significantly after the first few variable swaps, and then eventually reduce below the starting point. This allows a type of up-hill move to be taken – the acceptance of the entire sequence of pairwise swaps is based on the best position seen regardless of any increase in the intermediate DAG size. A limitation of the window permutation algorithm appears to be that several moves can be required to move a variable a long distance and these moves can be blocked by an intermediate up-hill move.

The sift algorithm requires $O(n^2)$ swaps of adjacent levels in the DAG, and each of these variable swaps has complexity proportional to the width of the DAG. To control the worst-case complexity, the search in a particular direction is terminated if the DAG size grows to twice its original size.

## 5 Experimental Results

The IWLS'91 benchmark set includes a directory of 76 combinational circuits (`cmlexamples`) and 40 sequential circuits (`smlexamples`). This includes the ISCAS'85 and ISCAS'89 benchmarks. Because most of these circuits are trivially small, I focus here on the 35 largest multiple-level examples. All DAG-sizes are given in thousands of nodes, and the run-times are measured on a Sun Microsystems SparcStation-10 Model 41.

Due to space limitations, only summary results are presented in this paper. Complete tables of results appear in [12].

### 5.1 Random Orders vs. Heuristic Orders

The first experiment was to form the OBDD's for all primary outputs. The same variable order was used for all of the primary outputs. The variable order was first determined with a single random trial, and then using the depth-first heuristic ordering algorithm.

When the maximum DAG size was set to 100,000 nodes (2.4 mB memory), it was not possible to form the OBDD's for 23 of the 35 circuits when using a random variable order, and it was not possible to form the OBDD's for 11 of the 35 circuits when using the depth-first heuristic ordering algorithm. The 11 circuits which failed when using a depth-first ordering were *c2670, c3540, c6288, c7552, i10, mm9a, mm9b, mm30a, s9234.1, s15850.1, and s38417.*

When the maximum DAG size was increased to 1,000,000 nodes (24 mB memory), the random order failed for 13 of the 35 circuits and the heuristic order failed for 7 of the 35 circuits. The 4 additional circuits which completed when given more memory were *c3540, i10, s9234.1, and s15850.1.*

### 5.2 OBDD Minimization Comparison

The next experiment compares the window permutation algorithm against the sift algorithm for OBDD minimization. The OBDD's for 24 of the 35 examples can be formed using the heuristic ordering algorithm and a 100,000 node limit. These 24 circuits were minimized after the OBDD's had been formed with the window permutation algorithm for $k = 2, 3, 4, 5$ and the sifting algorithm. The relative DAG-size and CPU ratios for each minimization algorithm is given in Table 1.

The sift algorithm results in OBDD's which are 45%[1] smaller than the heuristic order, while the window permutation algorithm with $k = 4$ produces OBDD's which are only 30% smaller. The sift algorithm produces OBDD's which are 20% smaller than the window permutation algorithm ($k = 4$) at the cost of an additional 40% in run-time.

## 5.3 Dynamic Variable Ordering

The next experiment compares the window permutation algorithm and the sifting algorithm in the context of dynamic variable ordering. For this experiment, I focused on the 11 examples which cannot complete with the heuristic order. Dynamic variable ordering was performed by applying the corresponding OBDD minimization algorithms at each garbage collection, and the measurement criteria was to see if the examples could complete. The results are given in Table 2 where FAIL refers to the number of primary outputs which failed to have their OBDD formed.

Dynamic variable ordering using the window permutation algorithm ($k$=4) was able to complete 3 of the 11 failed examples (*mm9a, mm9b, and s15850.1*). Dynamic variable ordering using the sifting algorithm was able to complete 9 of the 11 failed examples. Only *c6288* and *s38417* still fail with the 100,000 node limit. Even for these two failed examples, fewer outputs failed when the sifting algorithm was used.

## 5.4 Performance Impact

The last experiment compares the run-time for the 35 largest circuits without dynamic variable ordering, with dynamic variable starting from the heuristic variable order, and with dynamic variable ordering starting from a randomly generated variable order. The results are summarized in Table 3. The sift algorithm was used when performing dynamic variable ordering.

For the 24 examples which complete both with and without dynamic variable ordering, the run-time was increased an average of 6.8 when using dynamic variable ordering starting from the heuristic order. The average OBDD size for these 24 examples was reduced by a factor of 1.8 (45%) when dynamic variable ordering was used.

Table 3 also compares dynamic variable ordering starting from the heuristic order (DVO/heuristic) to dynamic variable ordering starting from a randomly generated variable order (DVO/random). The sifting algorithm was still able to complete for all but 3 examples (*c6288, mm30a, and s38417*). Interestingly, the DAG sizes starting from the random order were only slightly larger than the DAG sizes when starting from the heuristic order while the run-time increased by almost a factor of 2.

[1] Averages for the benchmark set are computed as the arithmetic mean of the ratio computed for each example.

## 5.5 Summary

The sift algorithm is superior to the window permutation algorithm, both as a static OBDD minimization algorithm and in the application of dynamic variable ordering. The sift algorithm consistently produces smaller OBDD's than the window permutation algorithm, although it has run-times which are longer.

The application of the sift algorithm in conjunction with dynamic variable ordering allows 9 of the 11 combinational circuits which could not complete using a static variable order to complete.

When a random variable order was used as the starting point rather than the heuristic variable order, OBDD processing was still able to complete for 32 of the 35 largest examples, including 8 of the 11 difficult examples. While some improvement still exists when starting from the heuristic order, for most examples a random order does not affect the ability of the OBDD processing to complete.

These results indicate that the goals of completing more computations and reducing the dependence on the need for heuristic ordering algorithms have been achieved for this application.

## 6 Conclusions

This paper proposes a modification to an OBDD package whereby the OBDD package owns and maintains the order of the variables. At each garbage collection, an algorithm is applied on the OBDD to reorder the variables so as to reduce the number of nodes in the OBDD. Two OBDD minimization algorithms were tried: the window permutation algorithm and the sifting algorithm. Little benefit was seen from the application of the window permutation algorithm for $k = 4$. Dynamic variable ordering using the sifting algorithm was able to complete several OBDD operations which were not able to complete without dynamic variable ordering, and the resulting OBDD tended to be significantly smaller. In almost all cases, dynamic variable ordering was able to complete the OBDD operation sequences even when starting from a random variable order rather than a heuristically determined variable order. The drawback of dynamic variable ordering is that the runtime for the OBDD operations increases significantly.

## 7 Future Directions

One direction for this work is to investigate dynamic variable ordering when applied to other OBDD problems. For example, the fixed-point algorithm found in sequential verification has the problem that determining a good variable order *a priori* for both the transition relation OBDD and the state space OBDD is difficult. It would be interesting to see if dynamic variable ordering could improve the efficiency and application of these algorithms.

The utility of dynamically determining the variable order for the OBDD has been demonstrated; however, the run-time impact is very large. One idea to improve the sifting algorithm would be to devise a more efficient algorithm to determine the optimum position for a variable (assuming all other variables remain fixed).

Another idea would be to explore exact bounding techniques that determine when a search in a particular direction can be terminated. Also, exploring completely different algorithms for OBDD minimization is a possibility.

Finally, it would be interesting to determine bounds on the growth of the OBDD as a single variable is moved $\pm k$ positions.

## Acknowledgements

Discussions with Karl Brace in 1989 led to the observation that only local operations were needed to exchange two variables in the OBDD and that all nodes at a given level of the OBDD could be reached efficiently at no cost. I would like to thank Jerry Burch and David Long for several discussions on the manual techniques they use to find good variable orders. Their suggestions motivated the sifting algorithm.

## References

[1] K. Brace. Personal Communication, June 1989.

[2] K. Brace, R. Bryant, and R. Rudell. Efficient Implementation of a BDD Package. In *Proceedings 27th Design Automation Conference*, June 1990.

[3] R. E. Bryant. Graph-based Algorithms for Boolean Function Manipulation. IEEE *Trans. Comp.*, C-35(8):677–691, August 1986.

[4] J. Burch. Personal Communication, June 1992.

[5] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams. In *Proceedings International Conference on Computer-Aided Design*, pages 2–5, November 1988.

[6] M. Fujita, Y. Matsunaga, and T. Kakuda. On Variable Ordering of Binary Decision Diagrams for the Application of Multi-level Logic Synthesis. In *Proceedings European Design Automation Conference*, pages 50–54, March 1991.

[7] N. Ishiura, H. Sawada, and S. Yajima. Minimization of Binary Decision Diagrams Based on Exchanges of Variables. In *Proceedings International Conference on Computer-Aided Design*, pages 472–475, November 1991.

[8] K. Karplus. Representing Boolean Functions with If-Then-Else DAGs. Computer Engineering UCSC-CRL-88-28, UC Santa Cruz, December 1988.

[9] J.-C. Madre and J.-P. Billon. Proving Circuit Correctness using Formal Comparison Between Expected and Extracted Behavior. In *Proceedings 25th Design Automation Conference*, pages 205–210, June 1988.

[10] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment. In *Proceedings International Conference on Computer-Aided Design*, pages 6–9, November 1988.

[11] S. Minato, N. Ishiura, and S. Yajima. Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation. In *Proceedings 27th Design Automation Conference*, June 1990.

[12] R. Rudell. Dynamic Variable Ordering for Ordered Binary Decision Diagrams. Technical Report, Synopsys, Inc., 700 E. Middlefield Road, Mountain View, CA 94043, January 1993.

| Algorithm | size | cpu |
|---|---|---|
| No minimization | 1.00 | 1.00 |
| Window, $k$=2 | 0.81 | 1.19 |
| Window, $k$=3 | 0.72 | 1.49 |
| Window, $k$=4 | 0.70 | 2.83 |
| Window, $k$=5 | 0.67 | 9.19 |
| Sift | 0.55 | 3.84 |

Table 1: Minimization Comparison.

| | No Dynamic Ordering | | Window Algorithm k=4 | | Sift Algorithm | |
|---|---|---|---|---|---|---|
| Example | SIZE | FAIL | SIZE | FAIL | SIZE | FAIL |
| C2670 | >100 | 2 | >100 | 2 | 6.6 | 0 |
| C3540 | >100 | 2 | >100 | 7 | 27.2 | 0 |
| C6288 | >100 | 22 | >100 | 22 | >100 | 21 |
| C7552 | >100 | 4 | >100 | 4 | 8.2 | 0 |
| i10 | >100 | 7 | >100 | 6 | 41.2 | 0 |
| mm9a | >100 | 7 | 4.4 | 0 | 2.0 | 0 |
| mm9b | >100 | 7 | 5.2 | 0 | 2.5 | 0 |
| mm30a | >100 | 60 | >100 | 60 | 17.6 | 0 |
| s9234.1 | >100 | 7 | >100 | 6 | 4.5 | 0 |
| s15850.1 | >100 | 8 | 54.0 | 0 | 17.5 | 0 |
| s38417 | >100 | 532 | >100 | 414 | >100 | 203 |

Table 2: Results for 11 Hard Examples.

| Algorithm | size | cpu |
|---|---|---|
| No minimization | 1.00 | 1.00 |
| DVO/Heuristic | 0.56 | 6.80 |
| DVO/Random | 0.58 | 11.80 |

Table 3: Performance Results.