Universität Freiburg
Institut für Informatik

Report 79, April 1996

Learning Heuristics for OBDD Minimization by Evolutionary Algorithms

Rolf Drechsler
Nicole Göckel
Bernd Becker

Learning Heuristics for OBDD Minimization by Evolutionary Algorithms

Rolf Drechsler Nicole Göckel Bernd Becker

Institute of Computer Science
Albert-Ludwigs-University
79110 Freiburg im Breisgau, Germany

April 4, 1996

Abstract

Ordered Binary Decision Diagrams (OBDDs) are the state-of-the-art data structure in CAD for ICs. OBDDs are very sensitive to the chosen variable ordering, i.e. the size may vary from linear to exponential.

In this paper we present an Evolutionary Algorithm (EA) that learns good heuristics for OBDD minimization starting from a given set of basic operations. The difference to other previous approaches to OBDD minimization is that the EA does not solve the problem directly. Rather, it developes strategies for solving the problem.

To demonstrate the efficiency of our approach experimental results are given. The newly developed heuristics are more efficient than other previously presented methods.

1 Introduction

Decision Diagrams (DDs) are often used in CAD systems for efficient representation and manipulation of Boolean functions (see e.g. [7]). The most popular data structure is the *Ordered Binary Decision Diagram* (OBDD) [6]. Recently, several approaches in logic synthesis have been presented that make use of OBDDs. Especially the good testability properties of circuits derived from OBDDs make them an interesting candidate for circuit design for testability [1, 2]. In [21] an OBDD-based method for low-power design has been proposed. Better results than using conventional synthesis tools could be obtained. Using OBDDs becomes also more and more popular in the area of *Field Programmable Gate Arrays* (FPGAs) [26]. There OBDDs have also been integrated in commercial FPGA tools [22]. It should be mentioned that in these applications a small gain (counted in the number of nodes) can tremendously simplify the problem to be solved by OBDDs.

One drawback of this data structure is that they are very sensitive to the variable ordering, i.e. the size may vary from linear to exponential. Finding the optimal variable ordering is an

NP-complete problem [5] and the best known algorithm has runtime $O(n^2 \cdot 3^n)$ [13, 19], where n denotes the number of variables.

This is the reason why in the last few years many authors presented heuristics for finding good variable orderings [24, 15, 25, 14, 16]. The most promising methods are based on *dynamic variable ordering* [17, 29, 27], since OBDDs for some Boolean functions could be constructed for which all other topology oriented methods failed.

Recently, new methods based on non deterministic algorithms have been proposed for OBDD minimization, e.g. GAs [11] and *Simulated Annealing* (SA) [28, 4]. The major drawback of these approaches is that in general they obtain good results with respect to quality of the solution, but the running times are often much larger than that of classical heuristics[1]. Due to the high complexity of the design process of CAD of ICs often "fast" heuristics are used. These heuristics are developed by the designer himself. But they also often fail for specific classes of circuits. Thus it would help a lot, if the heuristics could learn from previous examples, e.g. from benchmark examples.

Recently, a theoretical model for learning heuristics by *Genetic Algorithms* (GAs) has been presented [20, 9]. The new aspect of this model is that the GA is not directly applied to the problem. Instead the GA developes a good heuristic for the problem to be solved. First applications to multi-level synthesis and to 2-level AND/EXOR minimization have been presented. There the model has not fully been used, i.e. only a part of the features has been implemented.

In this paper we present an *Evolutionary Algorithm* (EA)[2] based approach to learn heuristics for OBDD minimization. Several heuristics have been proposed in the past few years (as described above). Our EA learns heuristics starting from some simple basic operations that are based on dynamic reordering. The learning environment is a set of benchmark examples, that is called the *training set*. We show by experiments that our EA finds out a heuristic that improves the results obtained by iterated sifting [29] by up to 60%. Furthermore the runtimes of the developed heuristics are low, since the costs of the heuristic are minimized during the learning process.

2 The Learning Model

In [9] a learning model has formally been introduced. In this section we briefly review the main notations and definitions to make the paper selfcontained.

It is assumed that the problem to be solved has the following property: There is defined a non empty set of optimization procedures that can be applied to a given (non-optimal) solution in order to further improve its quality. (These procedures are called *Basic Optimization Modules* (BOMs).) These BOMs are the basic modules that will be used. Each heuristic is a sequence of BOMs. The goal of the approach is to determine a good (or even optimal) sequence of BOMs such that the overall results obtained by the heuristic are improved.

In the following we assume that the reader is familiar with the basic concepts and notations of evolutionary approaches. For simplicity of the description we use multi-valued strings, i.e. strings that may assume various values from a fixed finite range. (For more details see [18, 8, 3].)

[1]This observation also holds for many other applications of CAD of ICs (see e.g. [23, 12, 20, 10].)

[2]In our application we make use of a modified GA, i.e. the GA works on multi-valued strings. Thus we call our implemented algorithm an EA.

The set of BOMs defines the set H of all possible heuristics that are applicable to the problem to be solved in the given environment. H may include problem specific heuristics but can also include some random operators.

To each BOM $h \in H$ we associate a *cost function*

$$cost : H \to \mathbf{R},$$

where $\mathbf{R}$ denotes the real valued numbers. *cost* estimates the resources that are needed for a heuristic. We measure the fitness *fit* of a string $s = (h_1, h_2, \ldots, h_l)$ of length l (representing a heuristic composed from l BOMs) by

$$fit(s) = c_c / fit_c(s) + c_q \cdot fit_q(s),$$

where

$$fit_c(s) = \sum_{i=0}^{l-1} cost(h_i)$$

is the cost fitness of string s and

$$fit_q(s) = \sum_{i=0}^{\#\ of\ examples} quality(example_i)$$

is the quality fitness of string s. c_c and c_q are problem specific constants.

The cost fitness measures the cost for the application of the string. If this cost is relatively high the resulting heuristic will take long time. If the heuristic has a low cost fitness it will terminate quickly.

The quality fitness measures the quality of the heuristic that is represented by the string s by summing up the results for a given *training set*.

The constants c_c and c_q are used to influence the primary goal of the heuristic: If c_c is set to 0 the GA will only optimize the heuristic with respect to the quality of the result, i.e. it will not care about the expenditure of the BOMs. If c_q is set to a small value the GA will determine a very fast heuristic, but the quality of the result will not be very good. Using these parameters the designer can influence the trade-off between runtime and quality and he can determine the primary goal of the GA: Should the heuristic focus on fast runtime or on good results?

For more details about the learning model see also [9].

3 Problem Domain

The essential definitions and properties of OBDDs [6] are briefly reviewed. Then the exchange of neighbouring variables that is the basic operation of our BOMs are given.

3.1 Binary Decision Diagrams

As well-known each Boolean function $f : \mathbf{B}^n \to \mathbf{B}$ can be represented by a *Binary Decision Diagram* (BDD) [6], i.e. a directed acyclic graph where a Shannon decomposition is carried out in each node.

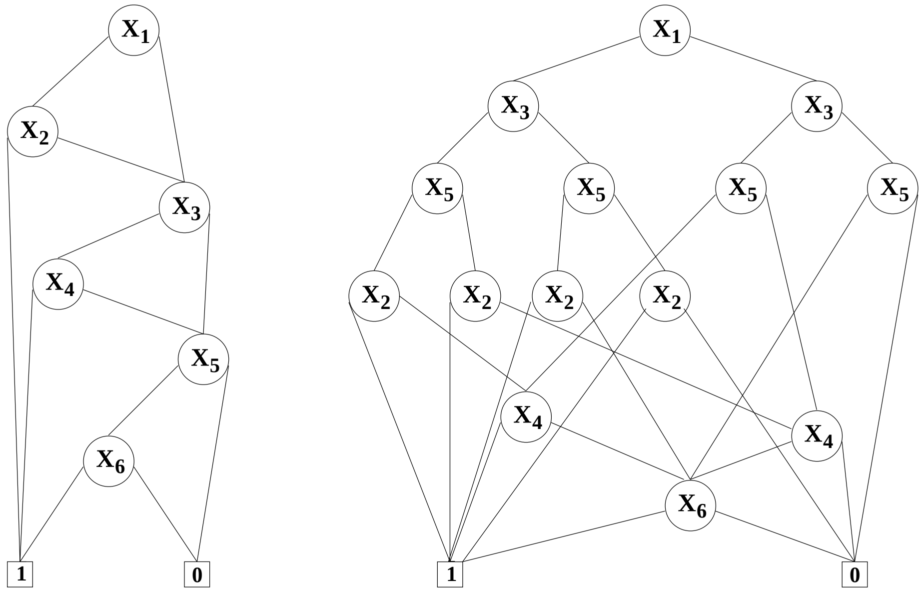

Figure 1: OBDDs of the function $f = x_1x_2 + x_3x_4 + x_5x_6$

A BDD is called *ordered* if each variable is encountered at most once on each path from the root to a terminal and if the variables are encountered in the same order on all such paths. A BDD is called *reduced* if it does not contain vertices either with isomorphic sub-graphs or with both edges pointing to the same node.

Reduced, ordered BDDs are a canonical data structure for Boolean functions and allow efficient manipulations [6]. In the following only reduced, ordered BDDs are considered and for briefness these graphs are called OBDDs.

The example from [6] shows the importance of the variable ordering for OBDDs:

Example 1 *Let $f = x_1x_2 + \ldots + x_{2n-1}x_{2n}$. If the variable ordering is given by $(x_1, x_2, \ldots, x_{2n})$ the size of the resulting OBDD is $2n$. On the other hand if the variable ordering is chosen as $(x_1, x_{n+1}, x_2, x_{n+2}, \ldots, x_{2n})$ the size of the OBDD is $\Theta(2^{n-1})$. Thus the number of nodes in the graph varies from linear to exponential depending on the variable ordering. In Figure 1 the OBDDs of the function $f = x_1x_2 + x_3x_4 + x_5x_6$ with variable orderings $x_1x_2x_3x_4x_5x_6$ and $x_1x_3x_5x_2x_4x_6$ are illustrated. As can be seen the choice of the variable ordering largely influences the size of the OBDDs.*

We now consider the following problem that will be solved using EAs:

> *How can we determine a good heuristic to determine variable orderings for an OBDD representing a given Boolean function f such that the number of nodes in the OBDD is minimized?*

Notice once more that we do *not* optimize OBDDs by GAs. Instead we optimize the heuristic that is applied to OBDD-minimization.

3.2 Dynamic Variable Ordering

It is well-known that in the case of OBDDs, the size of the decision diagram can be minimized by exchange of adjacent variables [17]. The exchange is performed very quickly since only edges must be redirected. This exchange is the basic operation for different algorithms for dynamic variable ordering, e.g the sifting algorithm, window permutation, and exact minimization algorithms [29, 19].

In the following we briefly describe the algorithms that are used as BOMs in the EA in the next section:

Sifting (S) [29]: By the sifting algorithm, the variables are sorted into decreasing order based on the number of nodes at each level and then each variable is traversed through the OBDD in order to locate its local optimum position while all other variables remain fixed.

Siftlight (L): Siftlight is a restricted form of sifting that does not allow the algorithm to do any hill climbing, i.e. the variables are directly located in the next minimum. (The algorithm is much faster than "normal" sifting, but in general the results are worse.)

Inversion (I): The variable ordering of the OBDD is inverted.

4 Evolutionary Algorithm

In this section we briefly describe the *Evolutionary Algorithm* (EA) that is applied to the problem given above.

Representation

In our application we use a multi-valued encoding, for which the problem can easily be formulated. Each position in a string represents an application of a BOM. Thus a string represents a sequence of heuristics. If a string has n components at most n applications of basic elements are possible. (This upper bound is set by the designer and limits the runtime of the heuristic.) Thus, each element of the population corresponds to an n-dimensional multi-valued vector. Using this multi-valued encoding each string represents a valid solution.

In the following we consider a four-valued vector: N means that no operation is performed. This allows that the costs of the heuristics can be reduced. S (L, I) represents sifting (siftlight, inversion) from Subsection 3.2. (We restrict to these simple alternatives, since they have shown to work very well in our application.)

Objective Function and Selection

As an *objective function* that measures the *fitness* of each element we apply the heuristics to several benchmark *training sets*. Obviously the choice of the benchmarks largely influences the (quality of the) results. On the other hand the designer can create several different heuristics for different types of circuits, e.g. a fast but simple heuristic for very large problem instances or a very "time consuming" heuristic for small examples. The function *quality* was calculated by constructing the OBDD and counting the number of nodes for each example. For the function *cost* we used the setting

$$cost(\mathrm{S}) = 4 \quad cost(\mathrm{L}) = 2 \quad cost(\mathrm{I}) = 1.$$

```
evolutionary_algorithm (training set) {
 generate_random_population () ;
 calculate_fitness () ;
 {
   apply_operators_with_corresponding_probabilities () ;
   calculate_fitness () ;
   update_population () ;
 } while (improvement obtained) ;
 return ;
}
```

Figure 2: Sketch of basic algorithm

The selection is performed by *roulette wheel selection*, i.e. each string is chosen with a probability proportional to its fitness. Additionally, we also make use of *steady-state-reproduction* [8]. A part of the best elements of the old population is included in the new one anyway. This strategy guarantees that the best element never gets lost and that a faster convergency is obtained.

Initialization
At the beginning of each EA run an initial population is randomly generated. The fitness is assigned to each element.

Operators
As operators we used *reproduction*, *crossover* and *mutation*. All operators are directly applied to the four-valued strings of length n that represent elements in the population. The parent(s) for each operation is (are) determined by the mechanisms described above. All operators only generate valid solutions, if they are applied to the four-valued strings.

Algorithm
Using the genetic operators our algorithm works as follows:

1. The initial population of size 10 is generated randomly and the length of the strings is set to 20.

2. Then $\frac{pop}{2}$ elements are generated by the genetic operators *reproduction* and *crossover*. The parent elements are selected by the method described above. The newly created elements are then mutated with a probability of 15%. After each iteration the size of the population is constant.

3. If no improvement is obtained for 200 generations the algorithm stops.

A sketch of the algorithm is given in Figure 2.

name	*in*	*out*	*optimal*		$sift_{iter}$		EA_{L1}	
			size	*time*	*size*	*time*	*size*	*time*
add6	12	7	28	19.4	54	0.1	28	0.1
addm4	9	8	163	5.4	163	0.2	163	0.1
cm85a	11	3	27	4.1	35	0.1	27	0.1
m181	15	9	54	332.0	56	0.2	54	0.2
risc	8	31	65	0.5	65	0.1	65	0.1
Learning time								1152.2

Table 1: Training set 1

5 Experimental Results

In this section we present results of experiments that were carried out on a *SUN Sparc 20* workstation. All runtimes *time* are given in CPU seconds. The benchmark functions are taken from LGSynth91.

In a first series of experiments we developed a heuristic for OBDD minimization on *training set 1* that is composed of small functions up to 15 variables. The results are given in Table 1. *in* (*out*) denotes the number of inputs (outputs) and *optimal* denotes the exact algorithm from [13]. *size* gives the number of nodes that is needed for the representation of the benchmark. $sift_{iter}$ denotes the size of the OBDD after sifting has iteratively applied until no further improvement could be obtained. The newly developed heuristic that is learned on *training set 1* by our EA is given in column EA_{L1}. Each OBDD minimization algorithm starts with an initial variable ordering. EA_{L1} **always** obtained the optimal OBDD sizes and the execution times are much better than the runtimes of the exact algorithm. In contrast, $sift_{iter}$ fails to determine the exact solution for 3 out of the 5 benchmarks in *training set 1*. The order of the BOMs of heuristic EA_{L1} is: ILSLL. The learning time for EA_{L1} given in the last row is about 20 minutes. Notice that our EA implicitly makes use of the cost function, i.e. redundant BOM sequences like "II" are filtered out.

Next, we applied the newly developed heuristic EA_{L1} to new small benchmarks that were not in *learning set 1*, i.e. that were not known during the learning process. As can be seen in Table 2 EA_{L1} fails for some examples by more than 10%. Hence, we extend *training set 1* by the examples that are in the bottom part of Table 2, i.e. the examples where EA_{L1} does not obtain satisfying results. Now the EA learns a heuristic on the extented set of benchmarks. The learning results are given in Table 3 in column $\mathrm{EA}_{L1_{ex}}$. Notice, that the OBDD sizes of the newly inserted functions are reduced. The application of $\mathrm{EA}_{L1_{ex}}$ to new benchmarks is also given in Table 2 in column $\mathrm{EA}_{L1_{ex}}$. The order of the BOMs for $\mathrm{EA}_{L1_{ex}}$ is given by: ISILSILLS. Now, the newly developed heuristic generates smaller OBDDs than $sift_{iter}$ and EA_{L1} and furthermore the runtimes are competitive.

In the next experiments we consider *training set 2* that includes larger examples for which the optimal solution can not be determined (within reasonable time bounds) due to the exponential

name	*in*	*out*	*optimal*		$sift_{iter}$		EA_{L1}		$\mathrm{EA}_{L1_{ex}}$	
			size	*time*	*size*	*time*	*size*	*time*	*size*	*time*
Z5xp1	7	10	41	0.6	41	0.1	50	0.1	41	0.1
alu4	14	8	349	1460.0	449	2.5	354	1.4	353	2.8
cm151a	12	2	16	24.8	16	0.1	16	0.1	16	0.3
cm162a	14	5	29	37.1	29	0.1	37	0.1	35	0.1
cm163a	16	5	25	110.0	25	0.1	26	0.1	25	0.1
cmb	16	4	27	176.6	27	0.1	27	0.1	27	0.1
cu	14	11	31	45.8	31	0.1	31	0.1	34	0.1
sqn	7	3	48	0.3	55	0.1	56	0.1	54	0.1
gary	15	11	289	955.0	293	0.3	306	0.2	289	0.5
s1494	14	25	368	615.0	372	1.0	383	0.4	373	0.9
tial	14	8	563	958.0	600	2.0	611	1.0	565	2.1

Table 2: Application to new benchmarks

name	*in*	*out*	$EA_{L_{ex}}$	
			size	*time*
add6	12	7	28	0.1
addm4	9	8	163	0.4
cm85a	11	3	27	0.1
m181	15	9	54	0.3
risc	8	31	65	0.1
gary	15	11	289	0.5
s1494	14	25	373	0.9
tial	14	8	565	2.2
Learning time				1505.3

Table 3: Extended training set 1

name	*in*	*out*	$sift_{iter}$		EA_{L2}	
			size	*time*	*size*	*time*
bc0	26	11	522	1.0	522	2.3
ex7	16	5	77	0.3	77	0.6
frg1	28	3	89	13.0	75	19.8
ibm	48	17	209	2.5	206	1.8
in2	19	10	252	2.9	230	4.0
s1196	32	32	597	3.7	597	4.3
ts10	22	16	145	2.0	145	2.7
x6dn	39	5	243	12.3	239	6.1
Learning time						22783.0

Table 4: Training set 2

name	*in*	*out*	$sift_{iter}$		EA_{L2}	
			size	*time*	*size*	*time*
apex2	39	2	473	58.6	353	61.6
apex7	49	37	292	1.4	291	1.7
bcd	26	38	589	1.5	567	2.1
chkn	29	7	261	4.5	305	4.8
cps	24	109	699	2.2	699	3.1
in7	26	10	76	0.5	76	0.6
pdc	16	40	601	1.6	601	2.2
s1494	14	25	372	1.0	368	0.9
t1	21	23	116	0.1	114	0.2
vg2	25	8	184	0.3	79	0.4

Table 5: Application to new benchmarks

worst case runtime of the exact algorithm. Table 4 shows the results obtained by the resulting heuristic that is called EA_{L2}. The results obtained by the new heuristic are never worse than those obtained by $sift_{iter}$. In contrast, for 50% of the considered benchmarks EA_{L2} creates OBDDs that need less nodes for the representation than sift_{iter}.

Additionally, EA_{L2} is applied to some new benchmarks that were not included in *training set 2*. The results are given in Table 5. As can easily be seen the newly developed heuristic never obtained much worse results than $sift_{iter}$ (3 nodes in the worst case), but in some cases is much better (see e.g. vg2).

6 Conclusions

We presented an *Evolutionary Algorithm* that learns heuristics for OBDD minimization. The EA optimizes the heuristics with respect to size and time. It has been shown by experiments that the heuristics learned by our EA on a training set obtain *very good* results for these elements. The newly developed heuristic also works well on *new* functions that were unknown during the learning process.

The learning algorithm only takes a few CPU hours on a workstation and the resulting heuristics that obtain better results than the best previously known are very fast, i.e. they take less than 10 CPU seconds for most examples.

We also studied the extension of the learning set during the run of the EA. In this case the EA adapts the heuristic in a way that the newly integrated functions are optimized, too, and thus the quality of the heuristic is improved.

References

[1] P. Ashar, S. Devadas, and K. Keutzer. Gate-delay-fault testability properties of multiplexor-based networks. In *Int'l Test Conf.*, pages 887–896, 1991.

[2] P. Ashar, S. Devadas, and K. Keutzer. Path-delay-fault testability properties of multiplexor-based networks. *Integration the VLSI Jour.*, 15(1):1–23, 1993.

[3] T. Bäck and H. Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computation*, 1(1):1–23, 1993.

[4] B. Bollig, M. Löbbing, and I. Wegener. Simulated annealing to improve variable orderings for OBDDs. In *Int'l Workshop on Logic Synth.*, pages 5b:5.1–5.10, 1995.

[5] B. Bollig, P. Savicky, and I. Wegener. On the improvement of variable orderings for OBDDs. *IFIP Workshop on Logic and Architecture Synthesis, Grenoble*, pages 71–80, 1994.

[6] R.E. Bryant. Graph - based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 8:677–691, 1986.

[7] R.E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM, Comp. Surveys*, 24:293–318, 1992.

[8] L. Davis. *Handbook of Genetic Algorithms.* van Nostrand Reinhold, New York, 1991.

[9] R. Drechsler and B. Becker. Learning heuristics by genetic algorihms. In *ASP Design Automation Conf.*, pages 349–352, 1995.

[10] R. Drechsler, B. Becker, and N. Göckel. A genetic algorithm for minimization of Fixed Polarity Reed-Muller expressions. In *Int'l Conf. on Artificial Neural Networks and Genetic Algorithms*, pages 392–395, 1995.

[11] R. Drechsler, B. Becker, and N. Göckel. A genetic algorithm for variable ordering of OBDDs. In *Int'l Workshop on Logic Synth.*, pages P5c:5.55–5.64, 1995.

[12] H. Esbensen. A macro-cell global router based on two genetic algorithms. In *European Design Automation Conf.*, pages 428–433, 1994.

[13] S.J. Friedman and K.J. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *Design Automation Conf.*, pages 348–356, 1987.

[14] H. Fujii, G. Ootomo, and C. Hori. Interleaving based variable ordering methods for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 38–41, 1993.

[15] M. Fujita, H. Fujisawa, and N. Kawato. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *Int'l Conf. on CAD*, pages 2–5, 1988.

[16] M. Fujita, H. Fujisawa, and Y. Matsunaga. Variable ordering algorithms for binary decision diagrams and their evolution. *IEEE Trans. on CAD*, 12:6–12, 1993.

[17] M. Fujita, Y. Matsunga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level synthesis. In *European Conf. on Design Automation*, pages 50–54, 1991.

[18] D.E. Goldberg. *Genetic Algortithms in Search, Optimization & Machine Learning.* Addision-Wesley Publisher Company, Inc., 1989.

[19] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchange of variables. In *Int'l Conf. on CAD*, pages 472–475, 1991.

[20] A. Kuehlmann and L.P.P.P. van Ginneken. Grammar-based optimization of synthesis scenarios. In *Int'l Conf. on Comp. Design*, pages 20–25, 1994.

[21] L. Lavagno, P. McGeer, A. Saldanha, and A.L. Sangiovanni-Vincentelli. Timed shannon circuits: A power-efficient design style and synthesis tool. In *Design Automation Conf.*, pages 254–260, 1995.

[22] V.V. Le, T. Besson, A. Abbara, D. Brasen, H. Bogushevitsh, G. Saucier, and M. Crastes. ASIC prototyping with area oriented mapping for ALTERA/FLEX devices. In *SASIMI*, pages 176–183, 1995.

[23] J. Lienig and K. Thulasiraman. A genetic algorithm for channel routing in VLSI circuits. *Evolutionary Computation*, 1(4):293–311, 1993.

[24] S. Malik, A.R. Wang, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Int'l Conf. on CAD*, pages 6–9, 1988.

[25] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagrams with attributed edges for efficient boolean function manipulation. In *Design Automation Conf.*, pages 52–57, 1990.

[26] R. Murgai, Y. Nishizaki, N. Shenoy, R.K. Brayton, and A. Sangiovanni-Vincentelli. Logic synthesis for programmable gate arrays. In *Design Automation Conf.*, pages 620–625, 1990.

[27] S. Panda and F. Somenzi. Who are the variables in your neighborhood. In *Int'l Conf. on CAD*, pages 74–77, 1995.

[28] D.E. Ross, K.M. Butler, R. Kapur, and M.R. Mercer. Functional approaches to generating orderings for efficient symbolic representations. In *Design Automation Conf.*, pages 624–627, 1992.

[29] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Int'l Conf. on CAD*, pages 42–47, 1993.