Minimizing the Number of Paths in BDDs

Görschwin Fey Rolf Drechsler
Institute of Computer Science
University of Bremen
28359 Bremen, Germany
{fey,drechsle}@informatik.uni-bremen.de

Abstract

BDDs are used in several fields as e.g. formal verification or synthesis. Minimizing the number of nodes in a BDD is a common technique, to reduce the memory needed to express a function. But recently applications like SAT-solving or synthesis have been shown to benefit from a small number of paths in a BDD. Here we present an algorithm and its implementation to carry out the minimization of a BDD with respect to the number of paths. After showing the existence of functions that can not be represented by a BDD that is minimal in the number of nodes and the number of paths at once, statistical experiments on the ISCAS89 benchmark set show the efficiency of the technique. In another set of experiments the minimization of numbers of paths is compared to that of the number of nodes.

1 Introduction

Binary Decision Diagrams (BDDs) are widely used in VLSI CAD applications. Due to the canonical representation of Boolean functions they are very suitable for formal verification problems and used in a lot of tools to date [3, 4, 5]. It is standard to measure the size of a BDD in the number of nodes since this is proportional to the memory needed. The variable order determines the number of nodes. Many techniques are known for the reduction of the number of nodes (see e.g. [10, 12]).

On the other hand, recently some applications have been studied where it turned out that the number of paths to one, also called one-paths in the following, in a BDD is important. In formal verification SAT techniques play an important role and the number of steps needed to solve a SAT problem can be measured by the number of paths in a BDD [11]. Also in the minimization during synthesis the influence of the number of paths in a BDD was shown [9, 14] and a minimized Disjoint-Sum-of-Product-representation can directly be extracted from the BDD [7].

Therefore, in this paper we present a technique to reduce the number of paths to one in a BDD. Our technique is based on local variable swapping. This allows for an easy integration into other techniques for reducing the size of a BDD.

We show the existence of functions with different BDDs for minimal size and minimal number of paths. Finally in our experiments we show the efficiency of our algorithm with statistical information and compare the minimization of paths to that of size.

The structure of the paper is as follows: Section 2 introduces the necessary notation to make the paper self-contained. In Section 3.1 the swapping procedure and the modification of sifting is explained. Experimental results are given in Section 4 followed by the conclusions in Section 5.

2 Preliminaries

2.1 BDDs

As is well-known a reduced ordered BDD is a directed acyclic graph $G = (V, E)$ representing a Boolean function [2]. The Shannon decomposition is carried out in each of its nodes on a given variable. By using *Complemented Edges* (CEs) the size of a BDD can be further reduced [1]. In the following we refer to reduced ordered BDDs with CEs simply as BDDs.

Formally the order of the n variables of a Boolean function can be given by mapping the variable index to a level in the graph G:

$$\pi : \{0, \dots, n-1\} \to \{0, \dots, n-1\}$$

A BDD with CEs has exactly one terminal node without any successors, denoted by `1`. Each other node $v \in V$ is labeled with an index $index(v) \in \{0, \ldots, n\}$ and has two successors, denoted as $\texttt{ThenChild}(v)$ and $\texttt{ElseChild}(v)$. Due to the order π the inequation

$$\begin{array}{rcl} \pi(index(v)) & < & \min(\pi(index(\texttt{ThenChild}(v))), \\ & & \pi(index(\texttt{ElseChild}(v)))) \end{array}$$

always holds (i.e. a node is always above its children), except if a child is the terminal `1`. For an edge $e \in E$ the attribute $\texttt{ComplementedEdge}(e)$ is true, iff e is a CE.

More than one function can be represented in one BDD by sharing isomorphic subgraphs between the functions, these functions are referred to as outputs in analogy to the outputs of a circuit.

Later on we need to split the predecessors of a node w into those having a CE to w and those having a non CE to w, therefore we define two sets:

$$\begin{array}{l} \mathcal{M}_1(w) := \{v : w \text{ can be reached from } v \text{ via a non CE}\} \\ \mathcal{M}_0(w) := \{v : w \text{ can be reached from } v \text{ via a CE}\} \end{array}$$

2.2 Paths in BDDs

Definition 1.

1. An ordered tuple

$$p = (v_0, e_0, v_1, e_1, \ldots, e_{l-1}, v_l)$$

with $v_i \in V, e_i = (v_i, v_{i+1}) \in E$ is called a *path* from v_0 to v_l. The edges e_i may be complemented or non complemented.

2. Two paths

$$\begin{array}{rcl} p_1 & = & (v_0, e_0, v_1, e_1, \ldots, e_{l-1}, v_l) \\ p_2 & = & (w_0, f_0, w_1, f_1, \ldots, f_{l-1}, w_l) \end{array}$$

with $v_i, w_i \in V$ and $e_i, f_i \in E$ are *identical*, iff

$$\begin{array}{rrl} \forall i \in \{0, \ldots, l\} & v_i = & w_i, \\ \forall i \in \{0, \ldots, l-1\} & e_i = & f_i. \end{array}$$

Otherwise the two paths are called *different*.

3. A path $p = (v_0, e_0, v_1, e_1, \ldots, e_{l-1}, v_l)$ is called *complemented*, iff it leads from v_0 to v_l via an odd number of CEs, i.e.

$$\begin{array}{rl} |\{e & : (e \text{ is an edge in } p) \wedge \\ & \texttt{ComplementedEdge}(e)\}| = 2i + 1 \end{array}$$

for $i \in \mathbb{N}$, otherwise the path is called *non complemented*.

4. A path $p = (v_0, e_0, v_1, e_1, \ldots, e_{l-1}, v_l)$ is called a *one-path from* v_0, iff the path is non complemented and $v_l = \texttt{1}$.

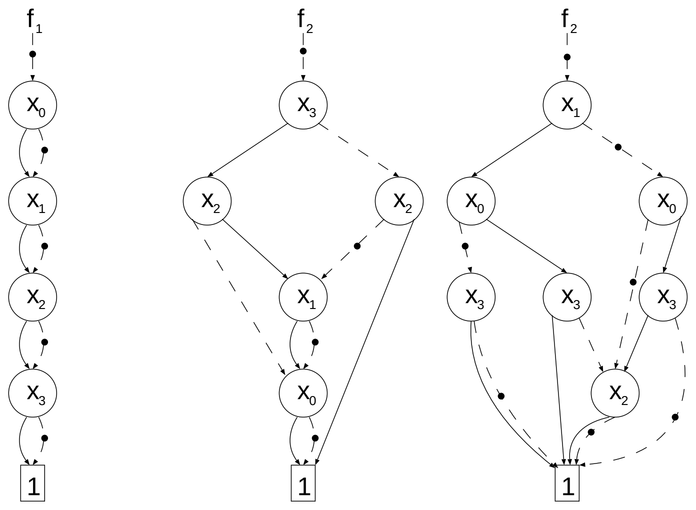
5. A path $p = (v_0, e_0, v_1, e_1, \ldots, e_{l-1}, v_l)$ is called a *zero-path from* v_0, iff the path is complemented and $v_l = \texttt{1}$.

By $P_1(\pi)$ we denote the sum of all different one-paths from any of the outputs (i.e. roots of the BDD) to the terminal `1` with respect to the variable order π. By $P_0(\pi)$ we denote the sum of all different zero-paths from any of the outputs with respect to the variable order π.

3 Minimizing the Number of Paths

3.1 Examples

Two introductory examples briefly clarify the difference between representing a function with BDDs of minimal path number or minimal size. Then the technique for efficiently calculating the number of paths is introduced, the two steps of keeping track of changes and propagating these changes are shown in detail. The section ends with the integration of the technique into Rudell's sifting algorithm [12].



(a) EXOR-function

(b) Minimal number of nodes

(c) Minimal number of paths to one

Figure 1: Examples of BDDs

Example 1. It is well-known that the EXOR-function $f_1 = x_0 \oplus \ldots \oplus x_n$ is represented by a BDD that has a number of nodes linear in n. The corresponding BDD is shown in Figure 1(a). Nonetheless the number of paths is exponential in n and, even worse, all BDDs representing f_1 have a number of paths exponential in n. This can be easily seen, since all BDDs representing f_1 have the same shape regardless of the variable order due to the symmetry of f_1.

Example 2. On the other hand the function $f_2 = \overline{x}_0\overline{x}_1\overline{x}_2 + \overline{x}_0 x_1 x_3 + x_0 x_1 \overline{x}_2\overline{x}_3 + x_0\overline{x}_1 x_2 x_3$ is an example where different variable orders lead to a BDD with either a minimal number of nodes (Figure 1(b)) or a minimal number of paths to one (Figure 1(c)). While the BDD minimal in size has six nodes (including the terminal) and five one-paths, the BDD minimal in the number of one paths has two more nodes, but only four one-paths.

3.2 Swapping Variables

Swapping two adjacent variables x_i, x_j in the variable order π results in the new variable order π' with

$$\pi'(k) = \begin{cases} \pi(k) & \text{if } k \neq i, j \\ \pi(i) & \text{if } k = j \\ \pi(j) & \text{if } k = i \end{cases} .$$

This operation has local effects on the BDD only regarding the number of nodes and is therefore used in most of the common reordering routines for BDDs. The operation influences the nodes in the two levels $\pi(i)$ and $\pi(j)$ but leaves all other nodes untouched. This way it is easy to keep track of the number of nodes in the BDD. One case that occurs during the swap of levels is illustrated in Figure 2. Due to the swap of x_i and x_j three nodes instead of the previous two are needed.

While swapping two variables is a local operation to counting the number of nodes [8], this is not the case in calculating the number of paths. If we know the number $p_1(w)$ of non complemented paths from the outputs to a node w and we also know the number of non complemented paths from w to `1`, the number of paths from the outputs to `1` via w is simply the product of both (of course, to get the total number of one-paths via w one has to also add the number of complemented paths to w multiplied by the number of zero-paths from w). By keeping track of all changes we can calculate $P_1(\pi')$, if $P_1(\pi)$ is also given. But a change in $p_1(w)$ will affect p_1 of all nodes on a path from w to 1 as well and we have to update all this values to be able to do the efficient calculation of P_1 during further swap operations.

Therefore we calculate $P_1(\pi')$ by keeping track of all the changes of $p_1(w)$ for all nodes w in levels below $\pi(i)$ during the swap and propagate these changes down to the terminal `1` afterwards. Obviously the number of paths from the outputs to one, $P_1(\pi')$, is equal to $p_1(\mathtt{1})$ after the propagation.

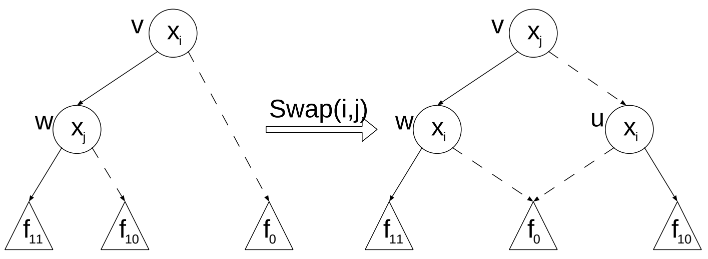


Figure 2: Nodes in a swap operation

For every node w of the BDD the value $p_1(w)$ ($p_0(w)$) denotes the number of non complemented (complemented) paths from the outputs to w. $p_0(w)$ and $p_1(w)$ can be calculated from values p_0 and p_1 of all predecessors of w:

$$p_0(w) = \sum_{v \in \mathcal{M}_1(w)} p_0(v) + \sum_{v \in \mathcal{M}_0(w)} p_1(v) \tag{1}$$
$$p_1(w) = \sum_{v \in \mathcal{M}_1(w)} p_1(v) + \sum_{v \in \mathcal{M}_0(w)} p_0(v) \tag{2}$$

At first we show how to keep track of the changes and then how the propagation is done.

3.3 Keeping Track of Changes

For each node $d_1(t)$ ($d_0(t)$) denotes the difference in the number of non complemented (complemented) paths from the outputs to t before and after the swap. Looking at Figure 2 again one can see the changes in the number of paths. $p_0(w)$ ($p_1(w)$) does not change and therefore $d_0(w)$ ($d_1(w)$) is not changed either. Before the swap the node u did not have v as a predecessor in previous calculations of $p_0(u)$ ($p_1(u)$), so we update

$$p_0(u) \leftarrow p_0(u) + p_0(v) \qquad (p_1(u) \leftarrow p_1(u) + p_1(v))$$

and

$$d_0(u) \leftarrow d_0(u) + p_0(v) \qquad (d_1(u) \leftarrow d_1(u) + p_1(v)).$$

Of course, Figure 2 is just one case of the changes made to nodes in levels below $\pi(i)$ but the schema of the updates applies in all other cases as well.

By iterating the nodes of level $\pi(i)$ all the changes are accumulated in the values d_0 and d_1 of nodes in levels below $\pi(i)$ (consider the case resulting from going through the example in Figure 2 from right to left).

For efficiency a stack $s(k)$ is assigned to each level k of the BDD. Every time a value $d_0(v)$ or $d_1(v)$ is changed the corresponding node v is pushed onto the stack $s(\pi'(index(v)))$. Thereby we assure that during propagation of changes we only look at those nodes in each level of the BDD where a change in the number of paths may have occurred.

This method to keep track of changes does not change the asymptotical time complexity of the swapping operation, since only a constant number of operations (additions and look-ups) on nodes that are touched anyway are added.

3.4 Propagation of Changes

Looking at Equation (1) and (2) the differences $d_0(w)$ and $d_1(w)$ can be calculated from the differences of all predecessors of w, respectively:

$$d_0(w) = \sum_{v \in \mathcal{M}_1(w)} d_0(v) + \sum_{v \in \mathcal{M}_0(w)} d_1(v) \tag{3}$$
$$d_1(w) = \sum_{v \in \mathcal{M}_1(w)} d_1(v) + \sum_{v \in \mathcal{M}_0(w)} d_0(v) \tag{4}$$

The algorithm works in a different way since efficient implementations of BDD packages only store information about the children of a node, but not of its predecessors. All nodes in the stacks are candidates for a change in the number of paths leading to the node from the outputs. Figure 3 shows the algorithm to propagate the changes. `PopFromStacks` returns in v the first element in the topmost stack that is not empty, topmost means associated to the highest level. This leads to a levelwise propagation of changes.

```
(0)  while PopFromStacks($\&v$) do
(1)    if $d_0(v) \neq 0$ or $d_1(v) \neq 0$ then
(2)      $w \leftarrow$ ThenChild($v$)
(3)      $p_0(w) \leftarrow p_0(w) + d_0(v)$
(4)      $d_0(w) \leftarrow d_0(w) + d_0(v)$
(5)      $p_1(w) \leftarrow p_1(w) + d_1(v)$
(6)      $d_1(w) \leftarrow d_1(w) + d_1(v)$
(7)      if not IsTerminal($w$)
           and ($d_0(w) \neq 0$ or $d_1(w) \neq 0$)
         then
(8)        PushOntoStacks($w$)
(9)      fi
(10)     $w \leftarrow$ ElseChild($v$)
(11)     if ComplementedEdge($v$,$w$) then
(12)       $p_0(w) \leftarrow p_0(w) + d_1(v)$
(13)       $d_0(w) \leftarrow d_0(w) + d_1(v)$
(14)       $p_1(w) \leftarrow p_1(w) + d_0(v)$
(15)       $d_1(w) \leftarrow d_1(w) + d_0(v)$
(16)     else
(17)       $p_0(w) \leftarrow p_0(w) + d_0(v)$
(18)       $d_0(w) \leftarrow d_0(w) + d_0(v)$
(19)       $p_1(w) \leftarrow p_1(w) + d_1(v)$
(20)       $d_1(w) \leftarrow d_1(w) + d_1(v)$
(21)     fi
(22)     if not IsTerminal($w$)
           and ($d_0(w) \neq 0$ or $d_1(w) \neq 0$)
         then
(23)       PushOntoStacks($w$)
(24)     fi
(25)     $d_0(v) \leftarrow 0$
(26)     $d_1(v) \leftarrow 0$
(27)   fi
(28) od
```

Figure 3: Algorithm to propagate changes

All predecessors of a node v were visited before the node itself. Thus, Equations (3) and (4) have been evaluated correctly when v is visited.

It is checked if $d_0(v)$ or $d_1(v)$ is unequal zero, because v might be on the stack twice (when pushing v onto the stack it is not checked if v is an element of the stack already).

The case of a CE can only occur when looking at an else-child of a node. Edges to a then-child are never complemented. This is due to the case normalization [1]. If the terminal node is reached no further propagation is necessary and the node is not pushed onto the stacks. Otherwise the node is pushed onto the stacks only if the update led to a difference in the numbers of paths. `PushOntoStacks` pushes the node onto the stack corresponding to the node's level in the BDD.

After propagating all updates from a node v, $d_0(v)$ and $d_1(v)$ have to be reset before visiting the node a second time and before the next swap operation.

Despite our application to calculating the number of one-paths this technique also calculates the number of zero-paths, $P_0(\pi)$ in $p_0(\mathtt{1})$, and the number of all paths in the BDD which is given by $P_0(\pi) + P_1(\pi)$.

3.5 Modification of Sifting

With a swapping procedure that calculates the number one-paths it is no problem to modify Rudell's sifting algorithm to minimize the number of paths instead of the size of a given BDD. In the original algorithm every variable is moved up and down in the variable order. At each position the size of the BDDs is measured and finally the variable is fixed at that position where the BDD was smallest. All the changes in the variable order are done by swapping adjacent variables.

If the described swapping procedure is used, the number of paths in the BDD can be used as the criterion which position to chose for a variable rather than the size of the BDD.

4 Experimental Results

Two experiments were made to demonstrate the difference between the BDDs minimal in the number of paths and those minimal in size. The first experiment was to enumerate all functions of up to four variables, in the second experiment we investigated the benchmark set ISCAS89 and compared the modified sifting algorithm to Rudell's sifting algorithm. For the modified algorithm we also gathered some statistical information to validate the efficiency of the technique.

The experiments were carried out on a Pentium II system at 400 MHz and 128 MB of physical memory. The machine was running under Linux. The algorithms were integrated into the CUDD package [13]. For the comparison with Rudell's sifting the implementation included in this package was used .

4.1 Enumerating Functions

For a given function we iterated through all possible variable orders, collecting all BDDs that were minimal in size or number of paths. Enumerating all functions of two or three variables gives the following result:

Lemma 1. *For all Boolean functions of two or three variables exists a BDD that is minimal in size and in the number of paths at the same time.*

This is not true for functions of four variables. Comparing the BDDs minimal in size with those minimal in the number of paths we found that 2.3 percent of the functions have no BDD that is minimal in size and number of one-paths at the same time. One of the functions was given in Section 3.1 already, the corresponding BDDs are shown in Figure 1(b) and 1(c), respectively. This leads to:

Lemma 2. *There are Boolean functions that do not have a BDD that is minimal in size and in the number of paths to one at the same time.*

Table 1: Statistical results for the modified sifting algorithm

	Number of nodes		
Circuit	all BDDs	below	visited
s1196	2957008	903030	69472
s1238	2957008	903030	69505
s1488	169021	52222	16631
s1494	169021	52222	16453
s208	85715	32565	4031
s27	1033	303	8
s298	42373	12822	730
s344	127781	42715	2236
s349	127781	42715	2240
s382	156606	67255	1484
s386	47334	17871	1484
s400	156606	67255	1475
s444	188077	62300	1594
s510	867230	364534	80065
s526	161321	56969	1818
s526n	161321	56969	1812
s641	4470758	1914101	39815
s713	4470758	1914101	39858
s820	360140	132114	20666
s832	360140	132114	20719
$\sum$	18037032	6827207	392096

Table 2: Comparison of BDDs resulting from Rudell's and modified sifting

	modified		Rudell	
Circuit	size	P_1	size	P_1
s1196	1523	2874	641	3511
s1238	1523	2874	641	3511
s1488	500	369	388	543
s1494	500	369	388	543
s208	62	53	61	79
s27	13	16	10	17
s298	91	70	78	73
s344	104	330	104	330
s349	104	330	104	330
s382	152	238	121	315
s386	158	61	123	70
s400	152	238	121	315
s444	154	243	161	447
s510	184	170	165	206
s526	153	162	141	368
s526n	153	162	141	368
s641	768	1700	629	2167
s713	768	1700	629	2167
s820	310	155	259	184
s832	310	155	259	184
$\sum$	7682	12269	5164	15728

4.2 Benchmark Set

Due to the propagation of changes towards the terminal the minimization of paths is more time consuming than minimizing the size. Not only the descent has to be done but also more memory per node is needed because of the extra information p_1, p_0, d_1 and d_0 associated with every node and in addition the stacks are needed. The sifting algorithms were applied after constructing the BDD of a circuit.

In the first experiment we gathered statistical information to judge the quality of our technique. Table 1 summarizes the results for those circuits of the ISCAS89 benchmarks the algorithm handled within the time bound of one hour. The column "visited" gives the number of nodes our algorithm has visited during sifting to propagate changes downward. Column "all BDDs" is the sum of nodes in all BDDs that were constructed for the different variable orders during sifting, in other words this is the number of nodes a brute force algorithm that visits each node would consider during the calculation of the number of paths. This justifies the use of memory to keep the extra information p_1, p_0, d_1 and d_0 for every node.

Column "below" gives the sum of all nodes that are below level $\pi(j)$ during all swap operations and therefore is the number of nodes that have to be visited without using the stacks, since every level would have to be completely iterated.

Summarized, the modified algorithm only had to visit 2.2% of the nodes a brute force algorithm had to. Furthermore for most of the nodes obviously no change in the number of paths had to be propagated, since the modified algorithm only visited 5.7% of the nodes below the swapped levels.

In a second run our algorithm was compared to Rudell's sifting algorithm. On some circuits only Rudell's sifting algorithm finished within the time bound of one hour, on some benchmarks both algorithms did not finish. In Table 2 we only list results for those circuits both algorithms were able to cope with.

The table lists for both algorithms the number of nodes and the number of one-paths of the BDD for each circuit. In the sum the modified algorithm produced BDDs with only 78% of the paths to one, while the size increased 1.48 times. For all circuits the modified algorithm produced a BDD smaller in the number of one-paths than the original sifting algorithm, as a consequence the size increased in general. Exceptions are the circuits "s344" and "s349" where neither the number of one-paths nor the size changed. For the circuit "s444" not only the number of paths but also the size of the BDD decreased when the modified algorithm was applied.

5 Conclusions and Future Work

In this paper we investigated minimization techniques for the number of one-paths in BDDs. First, we showed that size optimal BDDs do not imply path minimality. Then we gave an algorithm and its implementation. Experiments showed that in some cases more than a factor of two can be saved.

To further speed up the algorithm, it is focus of current work to apply techniques known from size minimization, like lower bounds [6] and variable grouping [10].

References

[1] K.S. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a BDD package. In *Design Automation Conf.*, pages 40–45, 1990.

[2] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.*, 35(8):677–691, 1986.

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

[4] R.E. Bryant. Binary decision diagrams and beyond: Enabeling techniques for formal verification. In *Int'l Conf. on CAD*, pages 236–243, 1995.

[5] R. Drechsler and B. Becker. *Binary Decision Diagrams – Theory and Implementation*. Kluwer Academic Publishers, 1998.

[6] R. Drechsler, W. Günther, and F. Somenzi. Using lower bounds during dynamic BDD minimization. *IEEE Trans. on CAD*, 20(1), 2001.

[7] G. Fey and R. Drechsler. Utilizing BDDs for disjoint SOP minimization. In *45th IEEE International Midwest Symposium on Circuits and Systems*, 2002.

[8] 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.

[9] A. Mishchenko and M. Perkowski. Fast heuristic minimization of exclusive-sums-of-products. In *5th Int'l Workshop on Applications of the Reed-Muller Expansion in Circuit Design*. Mississippi State University, 2001.

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

[11] S. Reda, R. Drechsler, and A. Orailoglu. On the relation between SAT and BDDs for equivalence checking. In *Int'l Symposium on Quality of Electronic Design (ISQED)*, pages 394–399, 2002.

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

[13] F. Somenzi. *CUDD: CU Decision Diagram Package Release 2.3.1*. University of Colorado at Boulder, 2001.

[14] Y. Ye and K. Roy. Graph-based synthesis algorithms for AND/XOR networks. In *Design Automation Conf.*, pages 107–112, 1997.