ACKNOWLEDGEMENTS

I would like to think my thesis advisor, Dr. Thornton, for the opportunity to work on this project, for his many insights into this research, and for the numerous times that he reviewed my thesis. I would also like to think Dr. Andrews and Dr. Lacy for serving on my thesis committee and for their input on this document.

Lastly, I would like to thank the National Science Foundation for funding this research effort. This research was funded under NSF contract MIP-9633085.

Table of Contents

List of Figures

Chapter 1 : Introduction / Background

1.1 Introduction

The manipulation of Boolean equations has long been a necessary part of digital systems design. A number of common design problems including synthesis, verification, and test vector generation can be modeled as a sequence of manipulations on Boolean equations. In addition to these design problems, many traditional computer science problems from domains such as artificial intelligence and combinatorics may be expressed in terms of Boolean equations. The growth in software systems which require Boolean equation manipulation, specifically computer-aided design (CAD) systems which utilize Boolean equations with large numbers of inputs, has created a need for an efficient data structure to represent and manipulate Boolean equations.

Common methods for representing Boolean equations include truth tables and cube sets. Truth tables represent Boolean equations by using one bit of memory to represent every row or minterm in the table where a one indicates the inclusion of that minterm in the cover of the function. Thus any function, even very simple functions, of n input variables requires 2^n bits. This exponential space requirement makes truth tables impractical for Boolean equations with many inputs. Cube sets, on the other hand, represent Boolean equations as a sequence of product terms (*e.g.*, product-of-sums form). This representation has the advantage of being able to represent many (but not all) Boolean equations in less than an exponential number of terms. However, cube sets may include redundant terms and are, therefore, not canonical representations of the equations. To ensure uniqueness, a time consuming reduction process must be applied to the cube sets.

Due to the failure of traditional methods to represent Boolean equations with large input sets efficiently, researchers have developed a new graph-based data structure known as the *Binary Decision Diagram (BDD)*. Introduced by Akers in 1978 [1] and popularized by Bryant in 1986 [2], the BDD has proven to be an efficient representation of Boolean equations in many modern applications [3]. While the BDD has proven to represent a large class of Boolean equations well, it is no panacea for CAD applications. In fact, BDDs can also grow exponentially large if a poor *variable ordering* (defined later) is chosen. This work focuses on the use of heuristics defined in terms of circuit output probabilities to determine efficient variable orderings so that BDDs might represent complex Boolean equations using a reasonable amount of computer memory.

1.2 Definition of Binary Decision Diagrams

In this section, the basic structure of the BDD is defined. It includes a brief history of the BDD as well as descriptions of some of the common modifications and extensions to the basic structure.

1.2.1 BDDs by Akers '78

The roots of the Binary Decision Diagram can be loosely traced to the work of Lee in the late 1950s [4]. Lee used what he termed "*Binary-Decision Programs*" to represent switching circuits. Akers then refined and renamed Lee's structure to create what is now often called the *Free-Form Binary Decision Diagram* [1]. The BDD, as defined by Akers, is a directed graph with two types of vertices, terminal and non-terminal. Terminal nodes have a single attribute which is a Boolean constant. When these nodes

are reached through a normal traversal of the graph for some set of input values, the value of the Boolean equation is given by the label of the terminal node. The non-terminal nodes are characterized by three attributes, a variable index and two outgoing edges. One of the edges is known as the *0-edge*; the other is the *1-edge*. The variable value, which is referenced by the index, determines which edge is traversed. If the variable has a truth value of one, then the *1-edge* will be traversed, otherwise the *0-edge* will be traversed.

As an example, the Boolean Equation $f = x_1 x_2 + \bar{x}_3$ can be represented by the BDD shown in Figure Chapter 1 :-1. If the input set is given by x_1=1, x_2=0, and x_3=0, then the path would be given by the highlighted lines (thick lines). And the value of the equation for that input set would be determined to be a logical-1 or true value.

BDDs, as defined by Akers, can effectively represent many Boolean equations; however, they have two major flaws. First, they are not canonical representations of the equations (*i.e.*, BDDs of different forms can represent the same function). The BDD in Figure Chapter 1 :-2 represents the same function as does Figure Chapter 1 :-1, but it has a substantially different and larger form. This lack of uniqueness makes equivalence testing extremely complex. The second problem is related to variable ordering. Variables under Akers design may be encountered in any order along any path and may even occur twice along a single path, which makes routines to perform basic Boolean operations such as AND and OR extremely difficult.

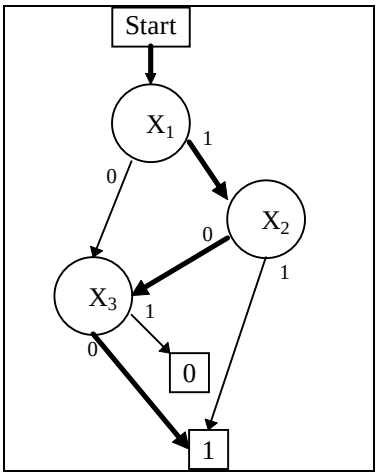


Figure Chapter 1 :-1: Simple BDD Example

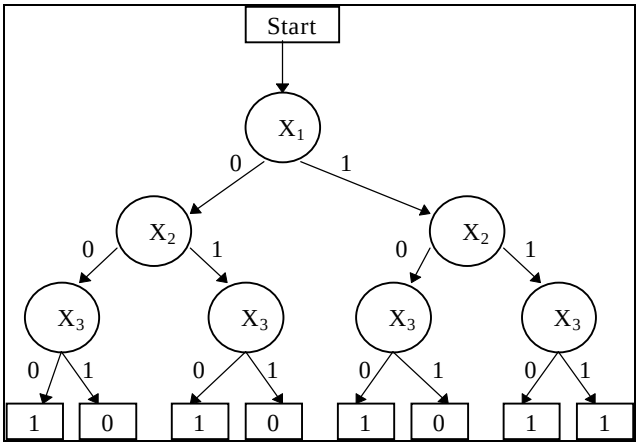


Figure Chapter 1 :-2: Same Simple Equation -- Different BDD

1.2.2 OBDD/ROBDD by Bryant '86

In 1986, Bryant defined the *Ordered Binary Decision Diagram (OBDD)* or *Reduced Ordered Binary Decision Diagram (ROBDD)* which corrected the major flaws in Akers' model [2]. He placed three restrictions on the definition of BDDs. Two of the restrictions are reduction rules. The first reduction rule calls for the removal of any redundant node which has a *1-edge* and a *0-edge* that point to the same node. Any edges

which were incident on the redundant node will now be incident on the node pointed to by the *1-edge* and *0-edge* of the redundant node (the x_3 node on the far right of Figure Chapter 1 :-2 is a redundant node since both edges point to 1). The second reduction rule calls for the removal of any repeated sub-graphs (the three x_3 nodes on the left of Figure Chapter 1 :-2 are identical and can be replaced by a single node). As a final restriction, Bryant created the idea of variable ordering which means that the variable index associated with a given node must be lower than the indexes associated with the nodes referred to by its *1-edge* and *0-edge*. In other words, variables must be encountered in the same order along all paths and may only be encountered once along any given path.

The Reduced Ordered Binary Decision Diagram is a canonical representation of Boolean equations for a fixed variable order. An equivalence test between two ROBDDs is simply a check for isomorphism between the two graphs which, for this restricted type of graph, has a computational time that is linear with respect to the size of the graphs [5]. Bryant's data structure has been shown to represent a large class of Boolean equations efficiently. Furthermore, Bryant's representation affords an efficient implementation of basic Boolean operations such as NOT, AND, and OR with a temporal complexity that is linear with respect to the size of the graphs in terms of the total number of vertices [6].

Bryant's Reduced Ordered Binary Decision Diagram (ROBDD) is the much more common structure in work related to Binary Decision Diagrams. For this reason, they are often referred to as OBDDs or just simply BDDs. Any further reference to the acronym BDD in this work will refer to Bryant's definition unless otherwise noted.

1.2.3 Attributed Edge BDDs

In order to reduce time and memory requirements of the BDD, researchers have proposed the use of *Attributed Edges*. The attribute is a function which is to be performed on the sub-graph referenced by the edge. The most common attribute is the negation or complement attribute (see [7] for a list of other attributes). This attribute simply takes the inverse of the function represented by the sub-graph. Figure Chapter 1 :-3 gives the BDD with negative-edge attributes for the Boolean equation $f = x_1\bar{x}_2 + \bar{x}_1 x_2$. Among the list of advantages for negative edge-attributed BDDs are reduced size (up to a half), constant time negation operation, and accelerated logic through an expanded list of operators [5]. If applied arbitrarily, negative edges can break down the uniqueness property of BDDs, but if the following rules are observed, uniqueness is preserved:

1. Use only the "1" terminal node.
2. Use negative attributes on the *0-edges* only.

Alternatively, one could use the "0" terminal node and attributes on the *1-edge*. As long as a consistent system is used, uniqueness will be maintained [5].

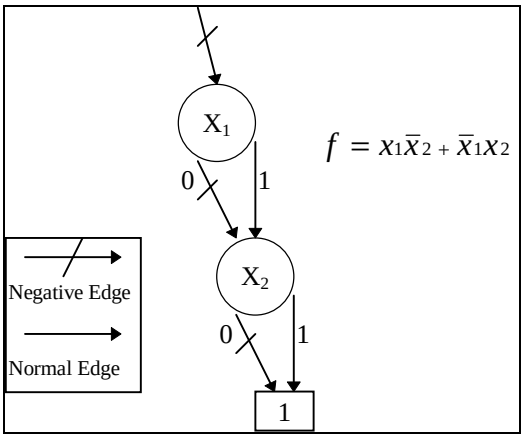


Figure Chapter 1 :-3: Example BDD with Negative Edges

1.2.4 Shared Binary Decision Diagrams (SBDD)

Multiple functions can be represented in a single multi-root graph where no two sub-graphs are isomorphic (*i.e.*, the functions share sub-graphs, and a consistent ordering is maintained for all functions). This structure is known as the *Shared Binary Decision Diagram (SBDD).* An example SBDD is illustrated in Figure Chapter 1 :-4. SBDDs have a number of advantages including allowing multiple functions to be represented compactly and reducing equivalence testing to a single pointer comparison [7].

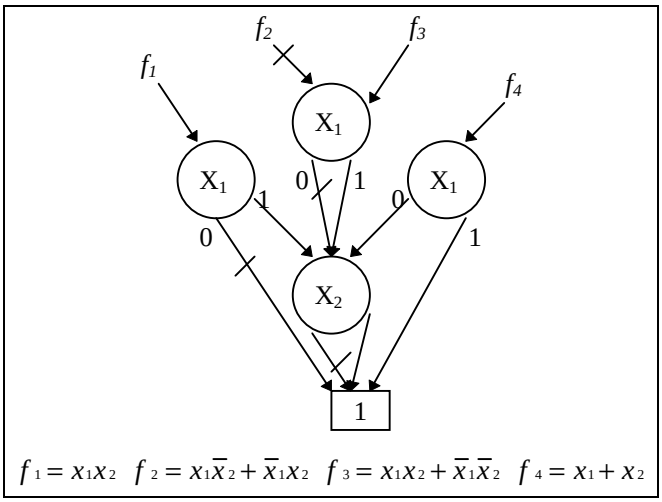


Figure Chapter 1 :-4: Example Shared BDD

1.3 Variable Ordering

As previously discussed, Bryant's definition of the BDD includes the concept of variable ordering which implies that variables will be encountered in the same order along every path and that any one variable will be encountered at most once along a given path. These restrictions ensure that BDDs are canonical representations of Boolean equations for a **fixed** variable order, but if the order is changed, then the structure and size (number of nodes) of the BDD may also change. It has been shown that the size of the BDD, for some functions, may vary widely depending on the variable ordering used -- from a best case of n nodes to a worst case of $2^n/n$ nodes for a function with n primary inputs [5]. Furthermore, it is known that reducing a BDD to its optimum size is an *NP-Complete* problem. To date, the most efficient algorithm known for the determination of the optimal order has a complexity of $O(n3^n)$ [8]. Fortunately, the optimal ordering is not necessary, and we only need to obtain a "good" variable ordering. Hence, we are

motivated to use heuristic based methods to obtain a variable ordering that produces a BDD of reasonable size.

There are two standard approaches to achieving a reasonable variable ordering. The first is to determine some property of the function prior to representing it in BDD form, set the order, and then create the BDD. This approach is known as *variable ordering*. The second approach, *variable reordering*, requires that the BDD be created with some general ordering, and then the variables are exchanged until a reasonable ordering is achieved.

1.3.1 Techniques of Variable Ordering

Since most Boolean functions are given in the form of a circuit description such as a net-list file, variable ordering generally involves some type of circuit analysis to determine various properties about the function and its variables. Two basic properties of variable ordering have been empirically determined:

- related variables should be near each other in the ordering,
- variables which have great control over the function should come early in the ordering[5].

In 1988, Fujita *et al.* proposed an ordering technique which attempted to exploit the first basic heuristic [9]. This method attempted to minimize cross-points in the circuit, and involved a depth-first traversal from outputs to inputs whereby the inputs that were encountered first during the traversal were placed first in the ordering. Fujita's technique

has been applied on a number of practical circuits and often achieved a reasonable ordering – particularly with circuits which have a low degree of fan-out [5].

A second ordering method, proposed by Minato *et al.* in 1990, used a dynamic weight assignment algorithm to attempt to discern those inputs which most greatly affect the output [7]. This algorithm proceeds by first assigning an equal weight to each of the primary outputs. The weight is propagated to the inputs by assigning the weight of each gate to be the sum of the weights of all its fan-out branches, and the weights of the fan-in stems to the gate are then determined by equally dividing the weight of the gate among them. The primary inputs that have the largest weight are assumed to exert the greatest control over the output and are ordered first. Like Fujita's method, this method can produce reasonable results for some circuits, but certainly not all.

1.3.2 Techniques of Variable Reordering

A number of reordering techniques have been proposed. Most of the techniques involve some sort of variable exchange algorithm which searches for a local minimum. One of the more successful reordering methods is known as *sifting* [10]. Sifting begins by taking the first variable in the ordering and exchanging it with each successive variable in the ordering. When this initial pass is complete, the variable which was originally first will be moved to the position which produced the smallest BDD. This process is repeated for all remaining variables. Sifting is essentially an exhaustive search of an n^2 sub-space of the possible $n!$ orderings where n is the number of primary inputs. Sifting can produce excellent results if the original ordering places it in a "good" sub-space; however, it may

be very time consuming or produce poor results or both if the original ordering is very poor [5].

A second reordering technique is *Simulated Annealing*. For a more detailed description of the algorithm, the reader is referred to [11] as only the basic features will be discussed here. Simulated Annealing is a process whereby small random changes are made in the ordering. The ordering is then settled into a local minimum according to some cost function. This process is repeated according to some "cooling schedule" until no improvement in size is achieved. Some of the best known orderings to date have been found using this process, but it carries the disadvantage of being slow. For some large benchmark circuits, simulated annealing algorithms can require several hours to complete. Furthermore, because of the random element, it does not necessarily produce the same result if repeated on the same circuit [11].

1.4 Output Probability as a Measure of Variable Significance

As noted above, one of the central heuristics in many ordering/reordering schemes is the notion that variables which greatly influence the output of the function should be placed near the top of the ordering. The difficulty is in measuring the influence of a particular variable. One primary focus of this work is the use of function output probabilities to measure the influence of variables and then determine an efficient ordering based on these measures.

Function output probability can be defined as the percentage of "ones" in the truth table. In other words, it is the likelihood that a function will produce a true value if all its inputs are selected at random assuming that each input has an equal probability of being a one or a zero. This value can be calculated directly from a BDD representation, and in certain forms can be used as an algebraic measure of Boolean function output correlation to inputs [12]. The following sections define terms and values needed to calculate the ordering heuristics used in this work.

1.4.1 The Shannon Decomposition Theorem

For completeness, the definition of the Shannon Decomposition theorem [13] is given. In each of the following expressions, f is assumed to be a Boolean equation of n input variables given by $x_1, x_2, \ldots, x_i, \ldots, x_n$. Equations 1-1 and 1-2 give the positive and negative cofactors respectively, while Equation 1-3 gives the relationship between these cofactors and f. The positive (negative) cofactor taken about x_i can be thought of as f where x_i is given as a 1 (0). The Shannon Decomposition theorem as given in Equation 1-3 is, in fact, the mathematical basis of the BDD as each node can be viewed as a decomposition about its variable index. The *1-edge* points to the positive cofactor, and the *0-edge* points to the negative cofactor.

$$f_{x_i} = f\left(x_1, x_2, \ldots, x_{i-1}, 1, x_{i+1}, \ldots x_n\right) \tag{1-1}$$

$$f_{\bar{x}_i} = f\left(x_1, x_2, \ldots, x_{i-1}, 0, x_{i+1}, \ldots x_n\right) \tag{1-2}$$

$$f = x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i} \tag{1-3}$$

1.4.2 The Consensus Function

A number of operations can be defined in terms of the Shannon cofactors. One such operator is the *Consensus Function* (universal quantifier) which is defined as the Boolean AND or intersection of the positive and negative cofactors from Shannon Decomposition taken about some variable, x_i (Equation 1-4). This operation gives the portion of f which is independent of x_i [14]. Hence, small values for the probability of the consensus function tend to indicate that the function is heavily dependent on the variable x_i, and therefore, the consensus function can be used as a reasonable metric in determining variable ordering.

$$C_{x_i}(f) = f_{x_i} \bullet f_{\bar{x}_i} \tag{1-4}$$

1.4.3 The Smoothing Function

Another possible ordering metric is the *Smoothing Function* (existential quantifier) which is defined as *Inclusive-OR* or union of the two Shannon cofactors taken about some variable, x_i (Equation 1-5). The smoothing function represents the behavior of a function when the dependence on a particular variable is suppressed [14]. Hence, values for the probability of the smoothing function that are close to the output probability of the function itself indicate that the variable about which the smoothing operator was taken has little influence over the function.

$$S_{x_i}(f) = f_{x_i} + f_{\bar{x}_i} \tag{1-5}$$

1.4.4 The Boolean Derivative

A final useful operator which can be defined in terms of the Shannon cofactors is the *Boolean Derivative* (Boolean difference). This operation is defined as the *Exclusive-OR* of the Shannon cofactors taken about some variable, x_i (Equation 1-6). The Boolean derivative is a measure of the observability of a particular input variable [14]. Thus if the probability of this operator for a particular input is close to zero, then input is relatively unobservable.

$$\frac{\partial f}{\partial x_i} = f_{x_i} \oplus f_{\bar{x}_i} \tag{1-6}$$

1.5 Contributions and Outline of Remaining Chapters

This work examines the use of function output probabilities to define variable ordering and reordering heuristics which can be applied to decision diagrams so that complex Boolean equations can be represented more efficiently. Both a new reordering algorithm (Chapter 3) and a new ordering algorithm (Chapter 4) are defined and empirically tested to determine their effectiveness. These algorithms require that function output probability be computed from a BDD (reordering) and from a circuit description (ordering). Therefore, as a byproduct, a new algorithm to determine output probabilities from BDDs is defined, and previously proposed methods for estimating output

probabilities from circuit descriptions are implemented and empirically tested to determine their accuracy.

In Chapter 2, a new algorithm for determining the function or circuit output probabilities is outlined. This algorithm is compared to a previously developed algorithm in terms of complexity and applicability to various decision diagrams. The new algorithm is found to be superior because it can more efficiently handle the shared sub-graphs of SBDDs and because it can be generalized to work with BDDs that have attributed edges.

Chapter 3 defines a new variable reordering algorithm for SBDDs with negative-edge attributes. The reordering method is a heuristic based method which uses the probability of the consensus function as the primary ordering metric. Experimental results are given for a number of practical benchmark circuits. This new reordering algorithm is shown to significantly reduce the size of SBDDs in reasonable time.

Chapter 4 proposes a new variable ordering algorithm which produces a SBDD with negative-edge attributes from a net-list file. The ordering heuristics are similar to those used in reordering. The primary difference is that the probability of the consensus is estimated from a net-list description rather than being calculated exactly from a BDD. The ordering algorithm is shown to perform in a less efficient manner than did the reordering algorithm because output probabilities are much more difficult to obtain from circuit descriptions. This work highlights the need to develop better algorithms for estimating the output probabilities from circuit descriptions.

Chapter 5 is the concluding chapter. It seeks to summarize this work and highlight its significance as well as give direction for future research.

Chapter 2 : Algorithm to Calculate Function Output Probability

2.1 Algorithm Definition

In the previous chapter, the function output probability was defined as the likelihood that a function will have a truth value of one given that each input has an equal probability of being a one or a zero. This definition is sufficient for the order/reordering experiments performed here; however, function output or signal probability can be more generally defined as the probability that a function or circuit output will have a truth value of one given the probability distributions of each of its inputs. Here, a new algorithm will be defined to calculate these more general output probabilities from a BDD. The algorithm has a linear complexity with respect to the size of the BDD and can work efficiently with shared and attributed edge BDDs.

The utility of this algorithm is not limited to ordering problems. Output probabilities have previously been used to calculate spectral values for Boolean equations, perform combinational logic synthesis, and to detect variable symmetry among Boolean input variables [15]. And recently output probabilities have also been utilized to design and analyze low-power circuitry [16] [17] [18].

2.2 Previous Algorithms

Several exact and approximate methods for the determination of circuit output probabilities have been developed. Algebraic and circuit topology based methods were proposed in the original paper [19]. Some other estimation methods have been developed

including those in [16] [20] [21] [22] [23] [24]. In addition to these methods which are largely based on circuit descriptions, one method that uses BDDs to find function or circuit output probabilities has been proposed [25].

The BDD based method in [25] can be used to compute the output probabilities in a very efficient manner if the candidate circuit is described using a compact BDD. The asymptotic complexity of this method is *O(N)* where *N* is the number of nodes in the BDD representing the function. The basis of this algorithm is a breadth-first traversal of the BDD to calculate the so-called "node" probabilities. Each node probability represents the probability that a given node will be reached during an arbitrary traversal of the graph.

The node probability for any node, P_{node}, can be defined in terms of the node probabilities, $P_{pred\_1}...P_{pred\_z}$, of its predecessor nodes (have an outgoing edge that is incident on the node). If the set $T = \{t_1,...,t_w\}$ represents all nodes which have a *1-edge* that is incident on the node *n*, and if the set $F = \{f_1,...f_x\}$ represents the set of nodes which have a *0-edge* incident on *n*, and if the probabilities of all input variables with which the nodes are associated are known, $p_{t1},...,p_{tw}$ and $p_{f1},...p_{fx}$, then the node probability, P_n, can be defined as:

$$P_n = \sum_{i=1}^{w} \left(p_{ti} \times P_{ti} \right) + \sum_{i=1}^{x} \left((1 - p_{fi}) \times P_{fi} \right) \qquad (2\text{-}1)$$

It is clear that the node probability of the initial node is 1.0 since all paths must begin at the initial node. It is fairly trivial to define a breadth-first algorithm which begins at the initial node and calculates the node probability of each node until the terminal one is reached. The node probability of the terminal one is the output probability of the function. It should also be clear that these node probabilities can be calculated in a single traversal of the BDD, and thus this algorithm has linear complexity. Figure Chapter 2 :-1 shows an example calculation for a simple ordered binary decision diagram (no negative attributes). In this example the probability of each input variable is assumed to be one half, and the node probability of the terminal-1 node is found to be 0.5 which is the correct output probability for the given function (the function covers four out of a possible eight minterms).

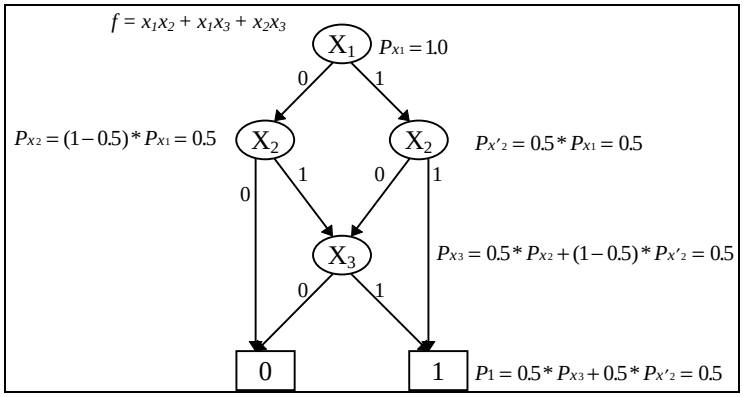


Figure Chapter 2 :-1: Example Breath-First Calculation

This algorithm suffers a number of deficiencies. First only one initial node is assumed which precludes the use of *Shared Binary Decision Diagrams* (SBDD). The algorithm can be applied to SBDDs but at a computational cost. The shared sub-graphs will need to be traversed multiple times (once for each output). Thus the asymptotic complexity for

finding the output probability of all functions in the SBDD becomes *O(m*max_n)*, where *m* is the number of outputs and *max_n* is the size of the largest BDD if the functions were represented individually. The second problem is that the algorithm does not generalize to handle *Attributed Edge BDDs*. The node probability is defined as the probability that a node will be reach during a traversal and does not account for how the node was reached, whether through an attribute or not. The last problem is related to the need to implement this algorithm on machines with finite computational precision. The above algorithm can cause under-flow problems if a proper pre-scaling value is not used since values as small as 2^{-k} can be computed where *k* is the longest path length in the BDD, and in theory *k* can be equal to the number of nodes in the BDD.

2.3 A New Depth-first Algorithm

To overcome the difficulties incurred by the algorithm in [25], a new depth-first algorithm is proposed. This algorithm also has linear complexity, but it can make more efficient use of SBDDs, can be generalized to work with attributed edges, and can avoid the under-flow problem.

Unlike the previous algorithm, this algorithm does not define "node" probabilities. Instead, a new term, "path" probability, is defined. The path probability can be defined at each node as the probability that a path to the terminal one will be taken given that the present node has been reached. To formalize the algorithm the following notation is used: PT_n refers to the path probability of a node, PT_{zero} and PT_{one} refer to the path probabilities of the nodes referenced by the given node's *0-edge* and *1-edge* respectively,

p_n is the probability of the variable which is associated with the given node. Using this notation, the path probability can be defined as:

$$PT_n = (p_n \times PT_{one}) + ((1 - p_n) \times PT_{zero}) \quad (2\text{-}2)$$

The path probability of the terminal one is 1.0, and the probability of the terminal zero (if one exist) is 0.0 since it has no paths that lead to a one. It is obvious that if the BDD is traversed from the terminal one node to the initial node in a depth-first fashion, then the path probabilities of each node can be easily calculated by the expression in Equation 2-2. When the initial node is reached, the path probability of the initial node gives the probability that any path taken in the BDD leads to the terminal one which is, in fact, the function output probability. Like the previous method, this method requires only a simple traversal of the BDD where each node is visited exactly once; therefore, the asymptotic complexity is *O(N)* where *N* is the number of nodes in the BDD. Figure Chapter 2 :-2 seeks to illustrate this algorithm. It contains the same simple BDD as did Figure Chapter 2 :-1, again all input variables are assumed to have a probability value of one half. The path probability of the initial x_1 node is the correct output probability for this function.

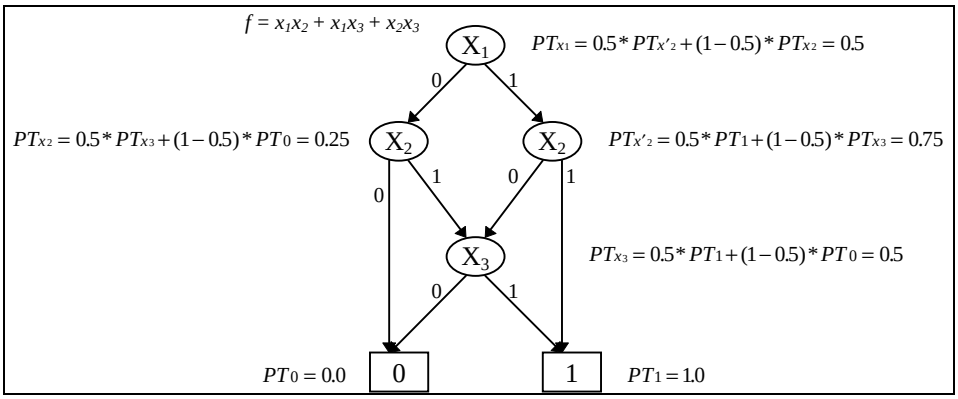


Figure Chapter 2 :-2: Example Depth-First Calculation

2.3.1 Depth-first Algorithm with SBDDs

This new depth-first algorithm can take advantage of the shared sub-graphs of the SBDD to increase efficiency (*i.e.*, the shared sub-graphs will only need to be traversed once). Path probabilities are defined in terms of successor nodes while the node probabilities of the previous algorithm are defined in terms of predecessor nodes. Thus the path probabilities depend only on the structure of the sub-graph to which they refer and not on the overall structure of the graph. The path probabilities for shared sub-graphs can be calculated independent of how many and what types of functions refer to them. The asymptotic complexity for finding the output probability of every function in a SBDD is *O(N)* where *N* is the total number of nodes in the SBDD. Figure Chapter 2 :-3 gives a SBDD with the corresponding path probabilities calculated.

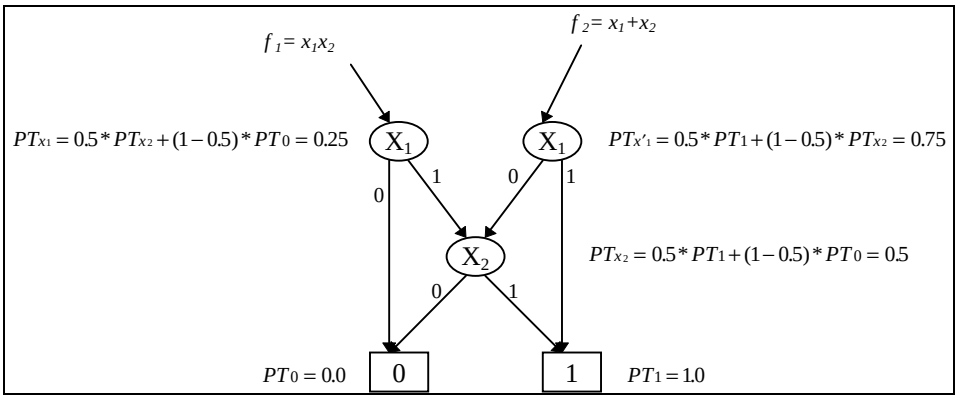


Figure Chapter 2 :-3: Example Calculation with a SBDD

2.3.2 Depth-first Algorithm with Attributed Edges

The depth-first algorithm can be extended to handle BDDs which contain attributed edges. For example, the common negation or inverse attribute can be handled by inserting a simple condition into the path probability calculation for each node. If the zero edge has the negative attribute (by our definition in Chapter 1, only the zero edge can have the negative attribute, but this idea could be extended to handle other cases), then the expression in Equation 2-2 is replaced by that in Equation 2-3; otherwise the expression in Equation 2-2 is used.

$$PT_n = (p_n \times PT_{one}) + ((1 - p_n) \times (1 - PT_{zero})) \qquad (2\text{-}3)$$

In general, the depth-first algorithm can be used with any attribute so long as a function can be defined to map the path probability of a Boolean equation onto the path probability of that equation when the attribute is applied. In other words, if *A* is an attribute which maps one set of Boolean equations onto another, then there must exist a

function, *AP*, that maps path probabilities such that the relationship in Equation 2-4 is maintained.

$$PT(A(f)) = AP(PT(f)) \tag{2-4}$$

The *AP* function for the negation attribute is simply *1-PT(f)* since for any function $PT(\bar{f}) = 1 - PT(f)$.

2.4 Implementation and Example Calculation

The algorithm described above was implemented and used in the ordering/reordering experiments described in the following chapters. For these experiments, the input variables are assumed to have an equal probability of being one or zero thus Equation 2-2 reduces to Equation 2-5.

$$PT_n = \tfrac{1}{2}(PT_{one} + PT_{zero}) \tag{2-5}$$

This algorithm was implemented as a recursive depth-first traversal very similar to the pseudocode in Figure Chapter 2 :-4. The *get_path_prob* routine can be called repeatedly on the multiple outputs of a SBDD.

```
double get_path_prob (BDD curNode)
{
    double zeroProb, oneProb, pathProb;

    if curNode is terminal-1
        return 1.0;

    if curNode has been visited
        return previously found pathProb value;

    zeroProb = get_path_prob (curNode->zeroEdge);
    oneProb  = get_path_prob (curNode->oneEdge);

    if (curNode->zeroEdge has negative attribute)
        zeroProb = 1- zeroProb;

    pathProb = 0.5 * (zeroProb + oneProb);
    store pathProb for curNode;
    mark curNode as visited

   return pathProb;
}
```

Figure Chapter 2 :-4: pseudocode for probability algorithm

As an example of this calculation, consider the BDD in Figure Chapter 2 :-5. Consider first the calculation of the output probability for f_1. The traversal starts at the terminal-1 node (after some initial recursive calls to initialize the frame stack). The path probability for this node is by definition 1.0. The next node to be visited is the x_2 node; the path probability for this node is found to be ½*(1.0+(1.0-1.0))= ½. The final node to be visited in this calculation is the initial x_1 node for f_1; the path probability for this node is ½*(½ +(1.0-1.0))= ¼ which is the correct output probability. Now if the output probability for f_2 is desired, there will be no need to recalculate the path probability of the x_2 node. The path probability for the initial x_1 node is ½ (1.0 + ½) = ¾ which is the output probability for f_2.

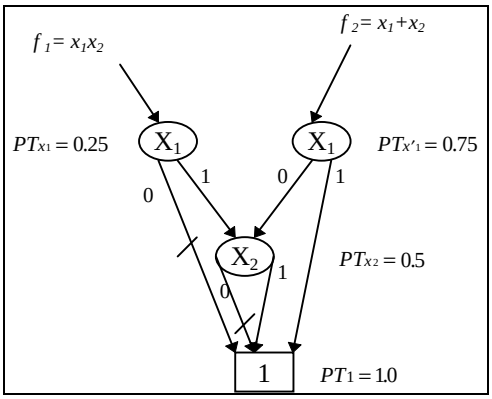


Figure Chapter 2 :-5: Depth-First Calculation with Negative Edge-Attributed SBDD

2.5 Experiments with Benchmark Circuits

Both the original breadth-first algorithm and the new depth-first algorithm were implemented so that a run-time comparison could be made. Figure Chapter 2 :-6 gives the results of this comparison. The column labeled *SBDD size* is the total number of nodes needed to represent the entire circuit in a SBDD, and the columns labeled *depth* and *breadth* give the total time, in seconds, needed to calculate the output probability of all outputs in the circuit. These experiments were performed on a 100 Mhz Sun SPARCstation 20 with 160 MB of RAM using SunOS version 5.5.

Cir. Name	Inputs	Ouputs	SBDD size	depth	breadth
c432.isc	36	7	31172	0.221	0.948
c880.isc	60	26	10348	0.064	0.548
duke2.pla	22	29	973	0.007	0.095
misex2.pla	25	18	136	0.001	0.004
misex3.pla	14	14	1301	0.014	0.152
apex2.pla	39	3	7096	0.068	0.209

Figure Chapter 2 :-6: Depth-first vs. Breadth-first Probability Algorithm

From Figure Chapter 2 :-6, it is clear that the depth-first algorithm is considerably faster for all circuits in the sample set. Unfortunately, a larger set of sample circuits can not be used because the available BDD library (the CMU package by D. Long) utilizes negative-edge attributes, and as previously noted, the breadth-first algorithm cannot account for the negative edges. The circuits in Figure Chapter 2 :-6, as it happens, do not contain any negative-edge attributes; therefore, both algorithms will produce correct results.

Chapter 3 : Reordering for Negative Edge-Attributed SBDDs

As discussed in previous chapters, decision diagrams are capable of representing a large class of Boolean equations in a compact manner [2]. It was further noted that this compactness is generally predicated on the choice of variable ordering. In this chapter a heuristic based reordering technique is defined which takes as initial input a negative edge-attributed SBDD.

3.1 Formulation of Methodology

The following sections define the terms and mathematical relationships needed to formulate the reordering heuristics. In addition, the motivation behind the heuristics will be discussed.

3.1.1 Consensus verses Smoothing and Derivative

In Chapter 1, three possible ordering metrics were outlined – the output probability of the consensus function, the smoothing function and the Boolean derivative (equations 1-4, 1-5, and 1-6 respectively). However, as is shown here, these quantities are closely related, and ordering according to any one of the three will produce similar results. The probability of the consensus function was chosen as the ordering metric in this work because it has a simple form that is easy to calculate. The probability of the smoothing function and the Boolean derivative can be expressed in terms of the output probability of the function itself, $p\{f\}$, and the output probability of the consensus function. The relationships are derived below.

The following lemmas define relationships between the output probabilities of the consensus function, the smoothing function, and the Boolean derivative.

Lemma 1: $2 \times p\{f\} = p\{f_{x_i}\} + p\{f_{\bar{x}_i}\}$

proof of *Lemma 1*:

$f = x_i f_{x_i} + \bar{x}_i f_{\bar{x}_i}$	(Shannon Decomposition – Equation 1-3)
$p\{f\} = p\{x_i f_{x_i}\} + p\{\bar{x}_i f_{\bar{x}_i}\}$	(union of mutually disjoint sets)
$p\{f\} = p\{x_i\} p\{f_{x_i}\} + p\{\bar{x}_i\} p\{f_{\bar{x}_i}\}$	(independent terms–cofactors are independent of x_i)
$p\{f\} = \frac{1}{2} p\{f_{x_i}\} + \frac{1}{2} p\{f_{\bar{x}_i}\}$	(primary inputs are assumed to have probability=½)
$2 \times p\{f\} = p\{f_{x_i}\} + p\{f_{\bar{x}_i}\}$ □	

The consensus function is defined to be the intersection of the two cofactors from Shannon decomposition (Equation 1-4). Therefore, the probability of the consensus function about a given variable, x_i, is defined by the following equation:

$$p\{C_{x_i}(f)\} = p\{f_{x_i} \bullet f_{\bar{x}_i}\} \qquad (3\text{-}1)$$

Lemma 2: $p\{S_{x_i}(f)\} = 2 \times p\{f\} - p\{C_{x_i}(f)\}$

proof of *Lemma 2*:

$p\{S_{x_i}(f)\} = p\{f_{x_i} + f_{\bar{x}_i}\}$	(Equation 1-5)
$p\{S_{x_i}(f)\} = p\{f_{x_i}\} + p\{f_{\bar{x}_i}\} - p\{f_{x_i} \bullet f_{\bar{x}_i}\}$	(probability of an Inclusive-OR)
$p\{S_{x_i}(f)\} = 2 \times p\{f\} - p\{f_{x_i} \bullet f_{\bar{x}_i}\}$	(*Lemma 1*)
$p\{S_{x_i}(f)\} = 2 \times p\{f\} - p\{C_{x_i}(f)\}$ □	(Equation 3-1)

Lemma 3: $p\left\{\partial f / \partial x_i\right\} = 2 \times p\{f\} - 2 \times p\{C_{x_i}(f)\}$

proof of *Lemma 3*:

$p\left\{\partial f / \partial x_i\right\} = p\{f_{x_i} \oplus f_{\bar{x}_i}\}$ (Equation 1-6)

$p\left\{\partial f / \partial x_i\right\} = p\{f_{x_i}\} + p\{f_{\bar{x}_i}\} - 2 \times p\{f_{x_i} \bullet f_{\bar{x}_i}\}$ (probability of an Exclusive-OR)

$p\left\{\partial f / \partial x_i\right\} = 2 \times p\{f\} - 2 \times p\{f_{x_i} \bullet f_{\bar{x}_i}\}$ (*Lemma 1*)

$p\left\{\partial f / \partial x_i\right\} = 2 \times p\{f\} - 2 \times p\{C_{x_i}(f)\}$ □ (Equation 3-1)

Given these relationships and the fact that *p{f}* is constant with respect to the input variables, ordering in ascending order according to the consensus function or in descending order according to the smoothing function or Boolean derivative will yield identical results. Therefore, only the consensus function is considered in this work.

3.1.2 Support of a Boolean Function

In a SBDD, it is not necessary that a given output function depend on all of the input variables. The *Support* of a function, *f*, is defined to be the set of variables on which *f* depends [26], and will further be denoted as *sup(f).* In addition, the set of output functions which depend on a given variable, x_i, can be defined as that variable's dependent set, denoted here as δ_i. In other words, δ_i is defined as all functions, f_j, such that $x_i \in \sup(f_j), \forall j$.

3.1.3 Calculation of the Consensus Metric over Multiple Outputs

Since the output probability of the consensus function provides a direct measure of the dependence of a function on a given variable, one can order a single BDD (not shared) according to the probability of the consensus taken about each individual variable. However, shared binary decision diagrams, specifically SBDDs with negative-edge attributes, provide a more compact means of representing multi-output functions, and the output probability of the consensus function with respect to a given variable may (and indeed likely will) vary across multiple functions. Hence, one must devise a means to combine these multiple probability measures into one ordering metric. Secondly, one must account for cases where a given variable is not a member of the support of all functions (*i.e.*, some functions may not depend on a particular variable). A heuristic based algorithm was developed which utilizes a negative edge-attributed SBDD as an initial input and has the following basic steps:

1. Determine a "Support Matrix" containing information regarding input variable support.
2. Compute a series of consensus function output probabilities.
3. Sort the variables according to the magnitude of the consensus output probabilities which are weighted by information derived from the "Support Matrix".
4. Apply the sifting algorithm [10].

These algorithmic steps are illustrated by the flowchart in Figure Chapter 3 :-1. This flowchart is further explained in the textual description of the algorithm contained in the flowing sections.

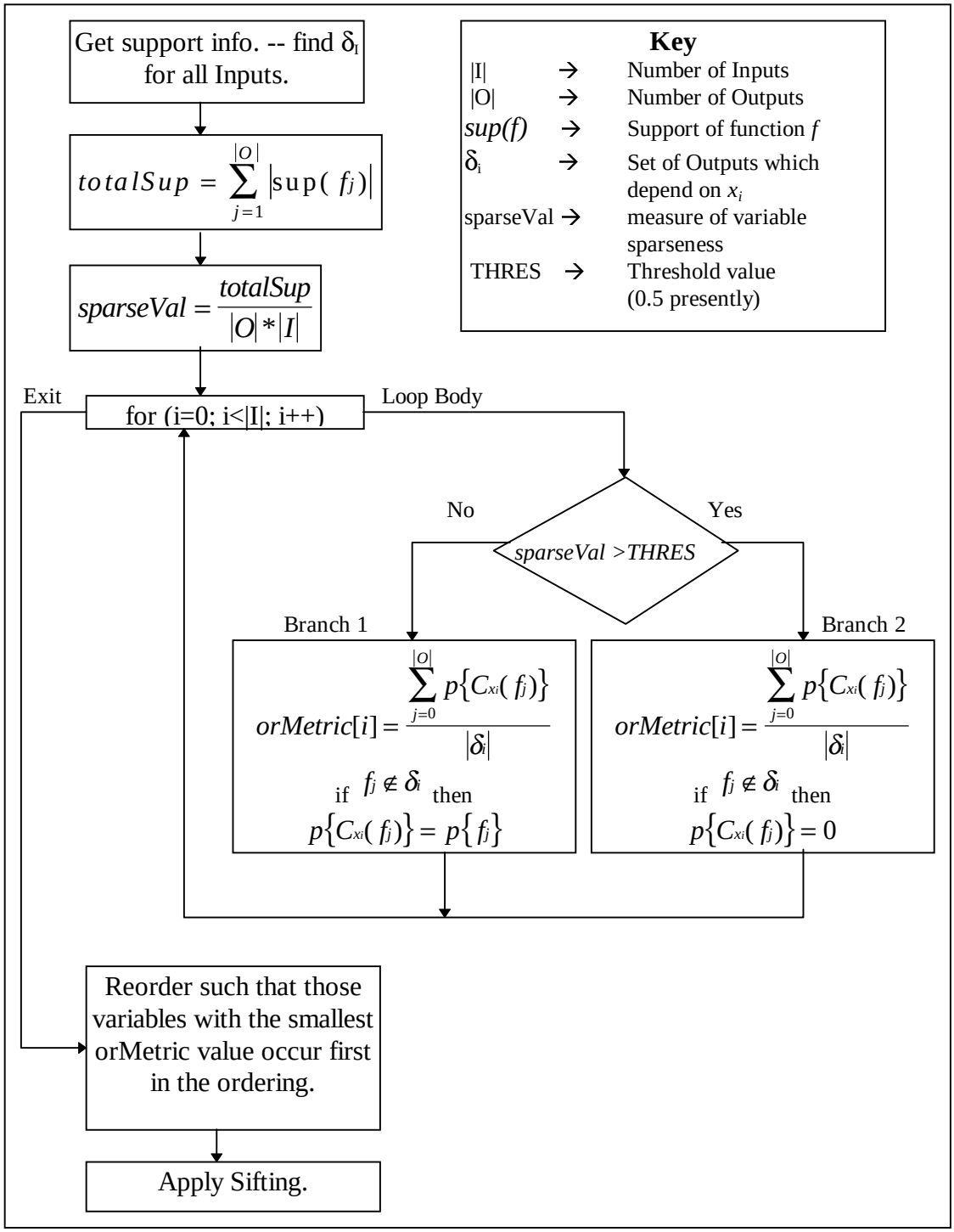


Figure Chapter 3 :-1: Flowchart for Reordering Algorithm

3.2 The Reordering Algorithm

3.2.1 Determination of the Support for Each Function

The first step is to determine the support for each output function. This operation requires a traversal of the SBDD from each output to all dependent inputs. During each traversal a list of inputs in the support is maintained, and each time a node representing a new variable is encountered that variable is added to the support list. The temporal complexity is then bounded by the product of the number of outputs and the average size of the BDDs representing the individual output functions. The support sets for the output functions effectively give the dependent sets for the input variables. A function, *f*, is in the dependent set of the variable, x_i, if and only if x_i is in the support set for *f*. This step is represented as the first item in the flowchart (Figure Chapter 3 :-1).

Also during this step, a measure of the "sparseness of support" is calculated. This measure is the average percentage of the total number of inputs which are in the support of each function. It is found by summing the number of variables in the support of each function and dividing by the product of the number of inputs and the number of outputs. For example, consider the four functions $f_1 = x_1x_2$, $f_2 = x_1 + x_2$, $f_3 = x_3x_4$, $f_4 = x_3 + x_4$, each function depends on two of the four possible input variables, so the sparseness measure is found to be one half. This measurement can be further illustrated by Figure Chapter 3 :-2. In Figure Chapter 3 :-2, if a given variable, x_i, is in the support set for a particular function, f_j, then an '$*$' is placed in the cell *i, j*. The sparseness measure is then the number of asterisks over the total number of cells or the degree to which the support

sets cover the “support area”. In this case, half the area is covered, so the sparseness measure is fifty percent or 0.5.

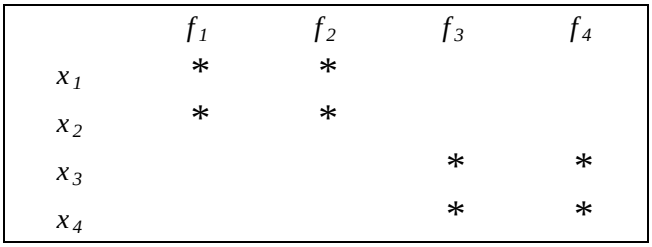


Figure Chapter 3 :-2: Support Area

This measure of support sparseness, denoted simply as sparseness for the remainder of this work, is later used as a threshold value to determine how functions which are not in the dependent set for a given variable affect its position in the ordering. In Figure Chapter 3 :-1, the sparseness value is referred to as *sparseVal*. The total number of variables in support of the functions, *totalSup*, is actually a counter that is maintain during the calculation of the support sets, but is shown in the flowchart as a separate step for clarity.

3.2.2 Determination of ordering metrics

The variables are ordered according to a weighted sum of the probability of the consensus function with respect to a given variable taken over all output functions. Stated differently, each variable is associated with an ordering metric. This metric is found by calculating, for each output function, the probability of the consensus with respect to the associated variable, summing the probability values, and dividing by some weight value. (This phase is contained in the *for* loop of the flowchart in Figure Chapter 3 :-1)

Variables will be ordered in ascending order according to their associated ordering metric.

The weight value for a given variable is the number of functions that the variable supports (*i.e.*, the number of functions which are in the dependent set, δ, for the given variable). The rational behind this weight is reasonably straight forward. The possibility for node sharing between output functions exist only among common variables in the support of those functions. Furthermore, node sharing between output functions can only exist if these common variables are not separated in the ordering by variables which are not common in the support of the functions. The weighting value serves to push variables which occur in few supports down in the ordering, and to pull those which occur in many supports up in the ordering; thus increasing the likelihood for node sharing between output functions.

Since all functions do not necessarily depend on all the input variables, the consensus function must be defined for those cases. If the standard definition for the consensus is used (Boolean AND of Shannon cofactors – Equation 1-4), then the consensus of a function about a variable on which the function does not depend is equal to the function itself ($C_{xi}(f) = f$ if f does not depend on x_i). This observation can be easily seen from the definition of Shannon cofactors and the consensus function. The positive and negative cofactors of a function taken about an irrelevant variable are both equal to the function itself. Therefore, their intersection is equal to the function itself. Furthermore, the output probability of a function, $p\{f\}$, is always greater than or equal to the output probability of the consensus taken with respect any variable, $p\{C_{xi}(f)\}$ (the proof is

trivial). Thus functions which are not in the dependent set, δ_i, for a given variable tend to push that variable down in the ordering. This definition of the consensus is represented by *Branch 1* of the flowchart in Figure Chapter 3 :-1.

This downward push on irrelevant variables can cause problems with circuits which have a high degree of coverage in their "support area", *i.e.*, the sparseness measure is close to one. In these cases, there is a great chance of node sharing between output functions since they have many common dependent variables. Hence, it was discovered that better orderings could be achieved by pulling the less common variables up in the ordering; therefore, the consensus function taken about irrelevant variables was defined to be zero when the function had a high sparseness value. This alternative definition is represent by *Branch 2* in Figure Chapter 3 :-1.

The decision on whether to take branch one or branch two is made by examining the degree of coverage of the "support area". This sparseness value was calculated in step one when the supports were determined. A threshold of one half was determined empirically, and this value worked well in the experiments performed here which utilize a large set of benchmark circuits.

3.2.3 Reordering of variables

At this stage, the variables are reordered such that those with the smallest ordering metric are ordered first and those with larger ordering metrics are ordered last. For the purpose of reordering, the *swap* operator is defined as in [10]. Performing a *swap* on a given

variable, x_i, will cause x_i to be moved to the i+1 position. For example, if the original ordering was $x_1, x_2,\ldots, x_{i-1}, x_i, x_{i+1},\ldots, x_n$, then a call to *swap(i)* will result in an ordering of $x_1, x_2,\ldots, x_{i-1}, x_{i+1}, x_i,\ldots, x_n$. *Swap* itself is a relatively elementary operation which can be implemented with most BDD packages. However, it can be seen that to completely reorder from one arbitrary ordering to another equally arbitrary ordering could require $O(n^2)$ steps where n is the number of primary inputs. It has been argued that this reordering can be done in at most $O(n-1)$ steps if a more complex *jump* operator is used [11]. This *jump* operator would allow any variable, x_i, to be directly moved to the j^{th} position. However, each *jump* operation is itself more complex than the *swap* operation, and its implementation is much more involved. Hence only *swap* operations were used in this work.

3.2.4 Sifting Algorithm

As a final step, we apply standard sifting as defined in [10]. As was discussed in Chapter 1, sifting is a reordering technique which searches a subspace of all possible orderings for a local minimum. In fact, it searches n^2 of the possible $n!$ orderings and keeps that ordering which produces the smallest BDD. This technique works well if the initial ordering is reasonable; therefore, our reordering metrics can be viewed as a pre-filter which selects a proper subspace for sifting to search. Sifting is the final step represented by the flowchart in Figure Chapter 3 :-1.

3.3 Experimental Results

In this section, the results from experiments where the reordering algorithm was applied to a number of benchmark circuits are presented. All experiments were performed on a 100 MHz Sun SPARCstation 20 with 160 MB of RAM using SunOS version 5.5, and all algorithms were implemented in C using the GNU ***gcc*** compiler and the BDD package developed by David Long at CMU. Figure Chapter 3 :-3 contains results from experiments using the ISCAS85 benchmark suite. The column labeled *Original Size* is the size of the SBDD used as input to the reordering routine. The original SBDD was created using a simplistic and not necessarily efficient variable ordering heuristic. This simple ordering technique first sorts all the output functions in descending order according to the number of variables in their support set, then the circuits are traversed in a depth-first fashion where each input is added to the ordering as it is encountered. The column labeled *Order Time* contains the time (in seconds) to parse the file and create the original SBDD using the simple ordering heuristic. The column labeled *Consensus-Sift* contains the size of the SBDD after variable reordering is performed using the reordering algorithm described in this chapter. The column labeled *Time* contains time needed (in seconds) to reorder the SBDD, and the column labeled *SimAnn* contains, for the purpose of comparison, the size of SBDDs obtained by simulated annealing and reported in [11] (and discussed in Chapter 1). Figure Chapter 3 :-4 contains the results of our experiments on several smaller two level circuits – no comparison with simulated annealing is available.

Circuit Name	Orignal size	Order time	Consensus-Sift	Time	SimAnn
c432.isc	31172	17	1691	969	1087
c499.isc	53866	9	34432	239	25866
c880.isc	10348	6	5816	48	4083
c1355.isc	53866	28	34432	242	25866
c1908.isc	13934	11	8762	67	5526
c3540.isc	72858	75	43564	652	23828

Figure Chapter 3 :-3: Reordering Experiments on ISCAS85 Benchmark Circuits

Circuit Name	Original size	Order Time	Consensus-Sift	Time
5xp1.pla	74	0	42	0
9sym.pla	25	0	25	0
alu4.pla	1197	2	567	1
bw.pla	108	0	101	0
con1.pla	18	0	16	0
duke2.pla	973	0	361	1
misex1.pla	41	0	37	0
misex2.pla	136	0	96	0
misex3.pla	1301	2	504	1
rd53.pla	17	0	17	0
rd73.pla	31	0	31	0
rd84.pla	42	1	42	0
sao2.pla	155	0	83	0
vg2.pla	1044	0	156	1
misex3c.pla	828	1	413	1
clip.pla	226	0	95	0
e64.pla	1441	1	231	4
apex1.pla	28336	43	1383	66
apex2.pla	7096	23	455	13
apex4.pla	928	1	893	0
apex5.pla	2679	1	1179	7
seq.pla	142252	31	1295	180

Figure Chapter 3 :-4: Reordering Experiments on PLA Circuits

3.4 Interpretation and Evaluation

As can be seen in Figure Chapter 3 :-3 and Figure Chapter 3 :-4, the reordering technique given here effectively reduced the size of the SBDDs for the set of benchmark circuits in reasonable time. In some cases, this reduction was quite significant (see c432, apex1, and seq). However, this algorithm was outperformed by simulated annealing in every case (where comparison was available). Simulated annealing has produced the best known orderings for many circuits, but this algorithm can require much computational time to complete. No exact times were reported in [11], but simulated annealing algorithms can require from one hour to one day to complete. When computational time is considered, the SBDDs produced by this reordering technique are certainly superior.

Chapter 4 : Ordering for Negative Edge-Attributed SBDDs

4.1 Ordering vs. Reordering

Chapter 3 described a new reordering technique. This technique requires an initial SBDD as input. It then determines some support information and calculates some probability values (probability of the consensus function to be precise). Using this information, the SBDD is reordered before sifting is applied. Ideally, a similar variable ordering technique could be developed where the support and consensus information are determined directly from the circuit description (*e.g.*, a net-list file), and the original ordering determined from that information. Once this SBDD was created, sifting could be applied just as it was in the reordering algorithm. If the probability values could be accurately determined from the circuit description, then the final SBDD should be as small as the final SBDDs created by the reordering algorithm, and the need to create an initial SBDD as input would be eliminated.

The Achilles heel to this approach is the need to calculate output probabilities from a net-list description which is a very difficult problem. In fact, it is a provably hard problem (see [24] for discussion of complexity issues). The following section discusses some of the issues incurred in calculating the output probabilities. Next, the ordering algorithm is further specified and experimental results are given for various ordering experiments.

4.2 Obtaining Output Probability from a Circuit Diagram

The output probability values (probability that an output will be one given the probability distributions of each of its inputs) can be directly calculated from a circuit description (*e.g.*, net-list files); however, the algorithm has exponential complexity with respect to computational time and is impractical for many circuits. Therefore, it has been necessary to develop methods which can estimate these values. A method to compute output probabilities for *tree circuits* (those with no fanout) was given in [21]. This exact method was extended in [23] to include circuits with fanout nodes. [23] also suggested some heuristic alterations to the exact method so that the probability could be estimated in a more reasonable time. The details of these algorithms and their implementation are repeated here to facilitate understanding of the difficulties in variable ordering heuristics based on these algorithms.

4.2.1 Exact method for tree circuits

Two paths in a circuit are reconvergent if they begin at a common node (node A) and end at another common node (node B). The paths fanout at node A and reconverge at node B [23]. Circuits which do not contain this reconvergent fanout are called tree circuits, and their output probabilities can be calculated in linear time by the algorithm given in [21]. If the input signals at each gate are independent as they are in tree circuits, then the output probability at each gate can be defined in terms of the probabilities of its inputs alone. The relationships are given in Figure Chapter 4 :-1. It is clear that the output probability of any other logic gate can be defined in terms of these relationships. Using these relationships, the output probability can be calculated by traversing the circuit from

inputs to outputs, calculating the probability at each gate. Figure Chapter 4 :-2 gives an example tree circuit and shows the probability calculation. This calculation is linear with respect to the number of gates in the circuit since only a simple traversal of the circuit is needed.

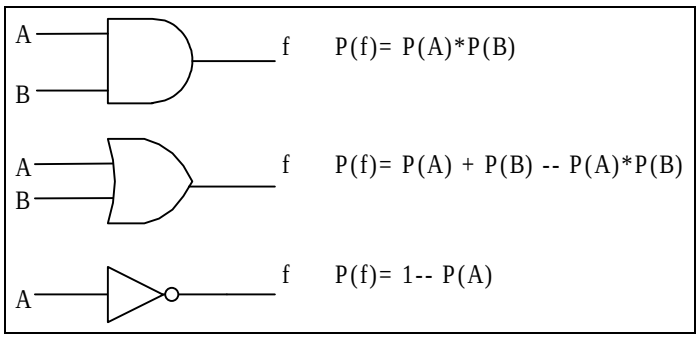


Figure Chapter 4 :-1: Output Probabilities for Simple Gates

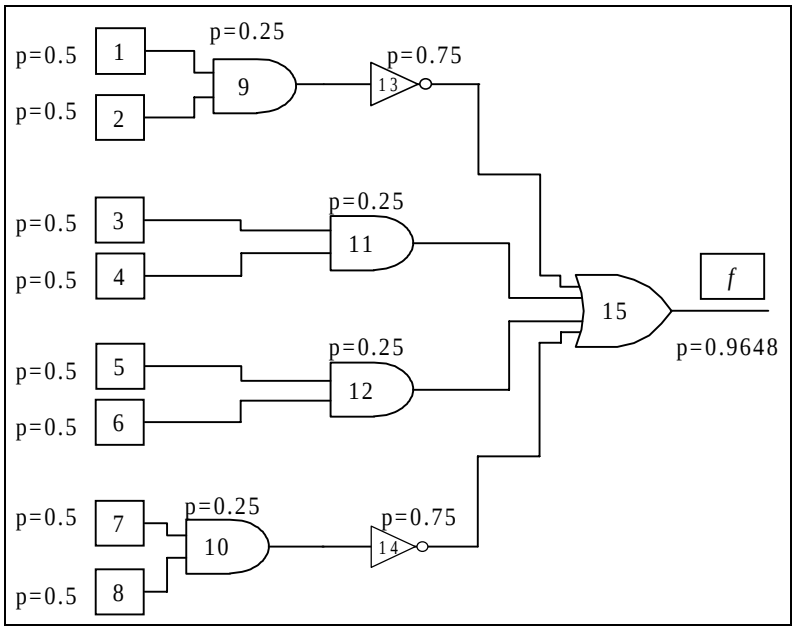


Figure Chapter 4 :-2: Simple Tree Circuit

4.2.2 Exact Method for Fanout Circuits

The above method works only for circuits which have no reconvergent fanout. If a circuit has reconvergent fanout, then the inputs to the reconvergent node (node where fanout paths meet) are not independent. For example, node 14 in circuit in Figure Chapter 4 :-3 is a reconvergent node, and its input signals are not independent. A method for handling this more general circuit was given in [23]. The algorithm involves the following major steps:

1. Represent the circuit in graph form and perform preprocessing.
2. Identify reconvergent nodes.
3. Find the "supergate" for each reconvergent node.
4. Using the supergates, find the output probability at each node.

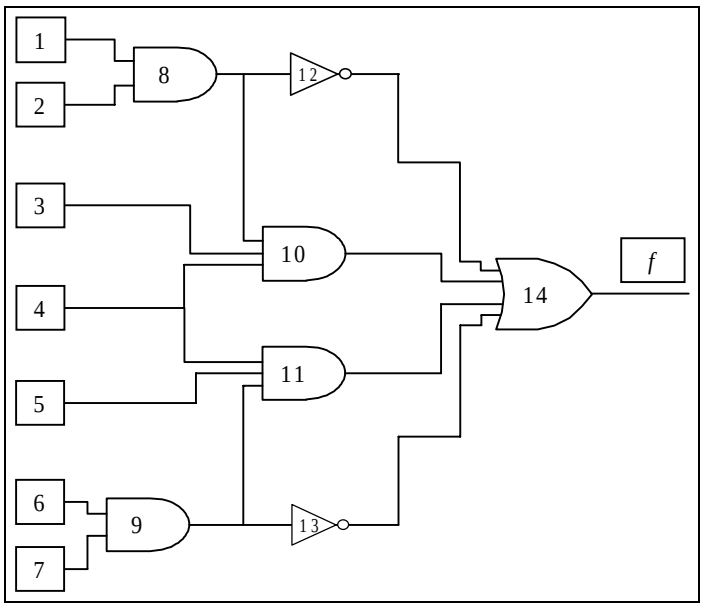


Figure Chapter 4 :-3: Circuit with Reconvergent Fanout

4.2.2.1 Labeling the Graph

When the circuit file (net-list file) is parsed, the circuit will be represented in graph form as in Figure Chapter 4 :-4. As an initial step, the graph nodes are labeled with certain attributes which ease the computational burden of later processes. The first attribute is the dependent input set which is defined at each node as the set of primary inputs in the node's cone of influence. The dependent sets can be easily determined by first labeling all primary inputs as depending only on themselves, and then defining the dependent set of every other node as the union of the dependent sets of its predecessor nodes. These sets can be calculated in a single traversal of the graph from inputs to outputs. At each node, the dependent set is the union of all sets on its incoming arcs. In Figure Chapter 4 :-4, the dependent sets are contained within the brackets.

The second attribute is the depth attribute. The depth is defined for each node as the maximum distance (number of nodes) between the given node and any node in its dependent input set. It can be found during the same pass as the dependent input sets are found. At each node, the depth is the maximum depth of all its predecessor nodes plus one. The depth of the primary inputs is defined to be zero. The depth attributes in Figure Chapter 4 :-4 are labeled as *d*.

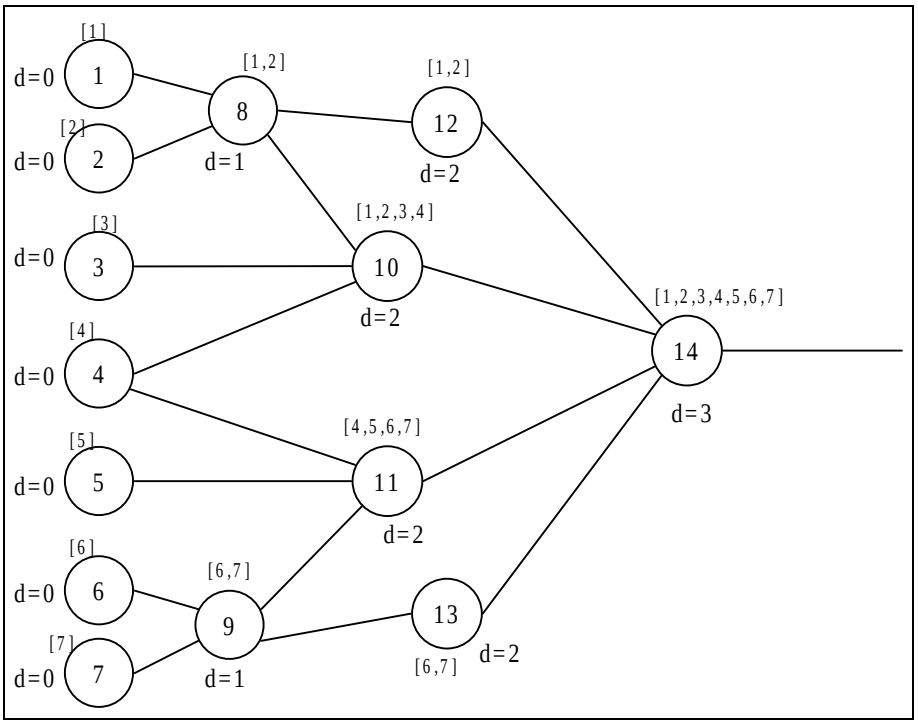


Figure Chapter 4 :-4: Labeled Graph for Circuit in Figure Chapter 4 :-3

4.2.2.2 Identifying the Reconvergent Nodes

Given a labeled graph like Figure Chapter 4 :-4, it is trivial to determine if a node is reconvergent or not. A node is reconvergent if an intersection exists between the dependent input sets of any two of its predecessor nodes. In other words, a node is not reconvergent if the dependent input sets of all its predecessor nodes are mutually disjoint. From Figure Chapter 4 :-4, it is clear that node 14 is the only reconvergent node since two of its predecessor nodes have an intersection (for example, compare node 10's dependent set to that of node 12, they have 1 and 2 in common).

Finding the output probability at a non-reconvergent node is a simple application of the tree formulas in Figure Chapter 4 :-1. The process for reconvergent nodes, on the other

hand, is more involved and is exponential in nature if the exact value is calculated. The following two steps help to explain this process.

4.2.2.3 Finding the Supergates

A formal definition of the term *supergate* is given in [23], but intuitively, the supergate of a given node can be viewed as a sub-graph which has its root at the given node and extends until the input signals to the sub-graph become independent. Consider again node 14 from Figure Chapter 4 :-4, its supergate is given in Figure Chapter 4 :-5. The dependent sets of the inputs to the supergate, 8, 3, 4, 5, 9, are mutually disjoint, and no smaller sub-graph with a root at node 14 exists where the inputs are independent. The supergate can be specified by its root node and its inputs. In this case, root=14 and the inputs I={8,3,4,5,9}. Furthermore, the input set of a supergate can be partitioned into two distinct sets, the fanout-inputs (IF) and non-fanout-inputs (INF). Fanout inputs are defined to be those which themselves fanout or those which are within the cone of influence of a fanout stem; all other inputs are non-fanout-inputs. Here, the fanout set is {8,4,9} and the non-fanout set is {3,4}.

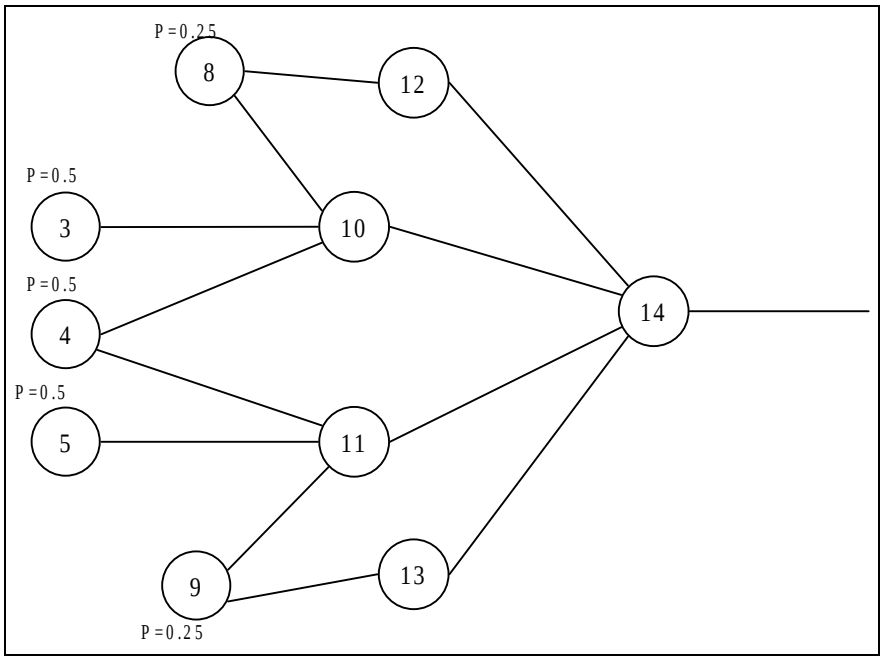


Figure Chapter 4 :-5: Supergate for Node 14

Identifying the supergates is a somewhat involved process, but this burden is lessened by the use of the dependent input sets and the depth attribute at each node. The basis of the process is a backward sweep of the nodes in the cone of influence of the node for which the supergate is to be determined (the supergate node). The process concludes when the dependent input sets become disjoint. To perform this backward sweep, two lists are maintained – an independent list and a non-independent list. The independent list is initialized to *null* and the non-independent list is initialized with the predecessor nodes of the supergate node. At each iteration, every node in the non-independent set is examined to see if it is independent (*i.e.*, its dependent set has no intersection with any other node in the list). If a node is independent, it is removed from the non-independent list and placed in the independent list; otherwise, it remains in the non-independent list. At the end of this process, the remaining non-independent node with the greatest depth is "expanded"

which means that it is removed from the list and its predecessor nodes are added. Care must be taken to ensure that nodes are not processed multiple times. For this reason, the nodes are marked as they are added to the non-independent list and will not be added if they are already marked. Once the node with the greatest depth has been expanded, the process repeats. The process terminates when the non-independent list is empty. At that time, the independent list will contain the inputs to the supergate. This process is linear with respect to the number of nodes in the supergate [23].

4.2.2.4 Finding Probability for Supergate Nodes

Once the supergate for a node has been identified, the output probability for that node can be calculated, assuming that the output probability of each of the inputs to the supergate is known. Let X be the node for which the output probability is desired and the root of the supergate, and let A be a binary vector, $(a_1,a_2,\ldots,a_n)$, such that each bit in the A vector is an assignment to one of the *n* fanout inputs of the supergate. The output probability of X given A can then be found by using the simple tree circuit relationships given in Figure Chapter 4 :-1 and traversing the supergate as if had no reconvergent fanout. This process is repeated for the 2^n possible A vectors, and the output probability of X is given by:

$$P(X) = \sum_{\forall A} P(X|A) * prob(A) \qquad (4\text{-}1)$$

The value *prob(A)* is the probability of a given A vector and is found by taking the product of the probability of each bit in the vector. This calculation is obviously

exponential with respect to the number of fanout inputs to the supergate (see [23] for a formal proof).

As an example, consider the supergate given in Figure Chapter 4 :-5. The probability of each input is given – these values can be easily calculated using the tree formulas. The table in Figure Chapter 4 :-6 gives the conditional probability calculation for each A vector.

A vector				X=Node 14	
Node 8	Node 4	Node 9	prob(A)	P(X|A)	P(X|A)*prob(A)
0	0	0	0.28125	1.00	0.28125
0	0	1	0.09375	1.00	0.09375
0	1	0	0.28125	1.00	0.28125
0	1	1	0.09375	1.00	0.09375
1	0	0	0.09375	1.00	0.09375
1	0	1	0.03125	0.00	0.00
1	1	0	0.09375	1.00	0.09375
1	1	1	0.03125	0.75	0.0234375
				Sum -->	0.9609375

Figure Chapter 4 :-6: Example Supergate Calculation

The circuit in Figure Chapter 4 :-3 is functionally equivalent to that in Figure Chapter 4 :-7, yet the circuit in Figure Chapter 4 :-7 has no reconvergent fanout. The simple probability routine for tree circuits can be applied to this example. And the result is the same as was obtained using the circuit with fanout and the supergate method.

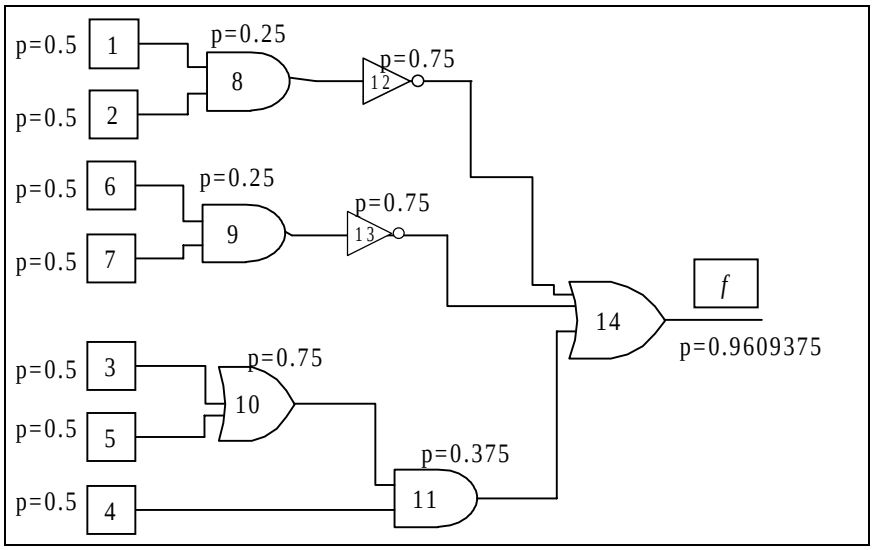


Figure Chapter 4 :-7: Equivalent Circuit without Fanout

4.2.2.5 Creating a Recursive Algorithm

A recursive algorithm can be defined which uses the steps outlined in the previous sections. Given a labeled graph as input, the probability algorithm will examine each node starting at the output node. If the node is not reconvergent, then a recursive call is made to each predecessor node and the output probability is defined by the relationships in Figure Chapter 4 :-1. If the node is reconvergent, the supergate for that node must be identified, and then recursive calls are made to each input to the supergate. Once the probabilities of the inputs to the supergate are known, the calculation can proceed as previously described. The recursion terminates when the primary input nodes are reached which have defined probabilities.

4.2.3 Estimation Techniques

The method described above can calculate the output probability for any circuit exactly. However, it has an exponential complexity with respect to the number of fanout inputs in each supergate. Many circuits have supergates with a large number of fanout inputs (c432.isc has supergates with as many as 35 fanout inputs); therefore, methods to estimate the probability using heuristics are necessary. In [23], two such heuristics were suggested. Both were implemented here, and the results will be discussed following the description of the heuristics themselves.

4.2.3.1 Heuristic One

Heuristic one is a distance based heuristic. Each node is defined to be a distance, D, away from the node for which the supergate is to be determined. The distance from a node, N, to the supergate node, X, is defined to be the length (number of nodes) of the shortest path from N to X (single input buffers and inverters are not counted). If the threshold is a distance, T, then any node at a distance of T or greater is assumed to be an independent signal, and the supergate will terminate at no more than a distance of T from the supergate node. Consider the graph in Figure Chapter 4 :-8. If the threshold is two, then the supergate for node 8 will be determined to be Figure Chapter 4 :-9 (a) which has one fanout input. If the threshold is one, then the sub-graph in Figure Chapter 4 :-9 (b) – which has no fanout inputs – will be considered the supergate, as node 6 and node 7 are a distance of one from node 8 and, thereby, considered independent.

The justification for this heuristic is that the farther a node is from the supergate node then the more it will “mix” with other signals and the less its effect will be on the output [23]. It is clear that if the threshold value is zero then all signals are considered independent and no node will be reconvergent, and if the threshold is larger than the largest supergate, then the routine will calculate the output probability exactly.

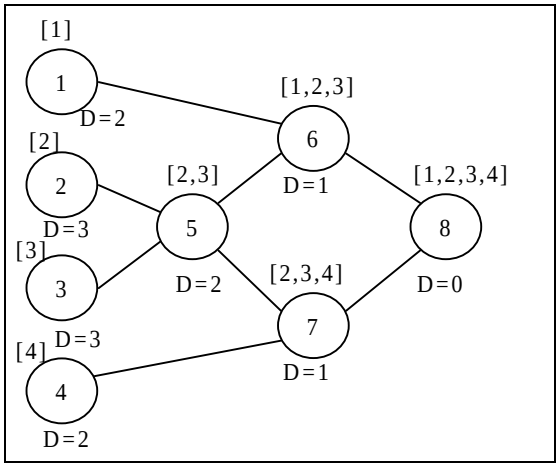


Figure Chapter 4 :-8: Example for Heuristics

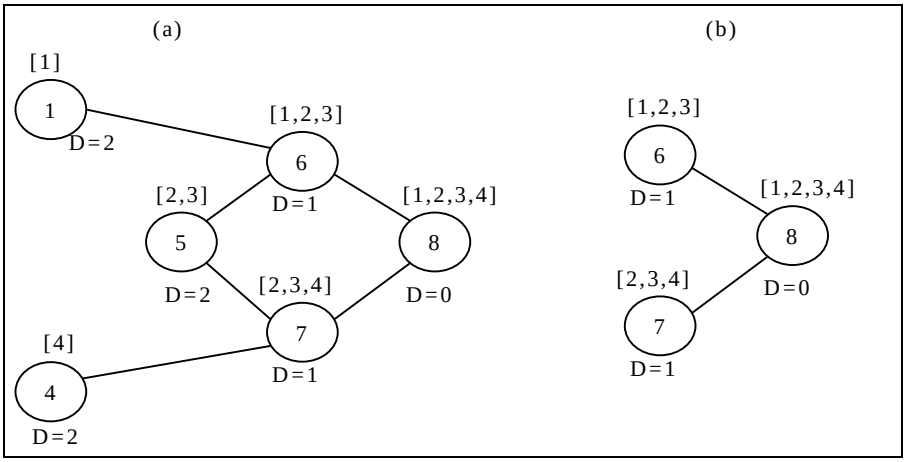


Figure Chapter 4 :-9: Heuristic Based Supergates for Node 8

4.2.3.2 Heuristic Two

The second heuristic proposed in [23] was based on a measure of the degree of independence between two nodes or signals. Let N1 and N2 represent two nodes from a circuit graph, and let Dp(N1) and Dp(N2) be the dependent input sets for the two nodes – input labels on the graph. The degree of independence between N1 and N2 is defined to be:

$$In(N1, N2) = \frac{\left|Dp(N1)\Delta Dp(N2)\right|}{\left|Dp(N1) \bullet Dp(N2)\right|} \qquad (4\text{-}2)$$

In Equation 4-2, |X| is defined as the cardinality of the set X, $X \bullet Y$ is the intersection of the sets X and Y, and $X\Delta Y$ is the symmetric difference between sets X and Y where the symmetric difference is defined as all elements that are in one of the two sets but not in both.

If a threshold is set to be some value, T, then any two signals which have an independence measure greater than T are considered independent. Clearly, if two nodes are truly independent (have no intersection), the denominator in Equation 4-2 goes to zero, and the independence measure goes to infinity. If the nodes have exactly the same dependent set, the numerator and thus the independence measure will be zero.

Consider again the graph in Figure Chapter 4 :-8. The independence measure between nodes 6 and 7 can be found by Equation 4-2 to be 1.0, since they have two dependent inputs in common (2 and 3) and two which are not in common (1 and 4). Therefore, if

the threshold value was equal to 0.5, then the supergate for node 8 would be represented by the sub-graph in Figure Chapter 4 :-9 (b) because nodes 6 and 7 would be considered independent signals, but if the threshold value was 1.5, then the supergate would be the sub-graph in Figure Chapter 4 :-9 (a).

4.2.4 Experimental Results for Probability Algorithm

Both heuristics were implemented and tested using circuits from the ISCA85 set of benchmark circuits. Unfortunately, heuristic two proved impractical as it would not complete in reasonable time for any threshold greater than zero; therefore, no results are reported for that method. The first heuristic, the distance based method, will run on the benchmark circuits for thresholds up to three, and the results from these experiments are reported in Figure Chapter 4 :-10. For every circuit in the sample set, the output probability was estimated for each of its outputs using the defined heuristic and the given threshold. This estimated probability was compared against the actual probability so that the error could be calculated. In Figure Chapter 4 :-10, the column labeled *Dist. Threshold* is the distance threshold in terms of maximum depth of the supergate. The column labeled *Avg. Error* refers to the average absolute error for all outputs in the circuit. *Max Error* gives the maximum error of any of the estimations concerning the given circuit. *Avg. %Error* is the average percentage error found by dividing the absolute error of each probability estimation by the actual probability and taking the average over all outputs of the circuit. And the column labeled *Time* contains the time in seconds to estimate the probability for all outputs in the circuit. All experiments were performed on a 100 MHz Sun SPARCstation 20 with 160 MB of RAM using SunOS version 5.5.

Upon examining the results given in Figure Chapter 4 :-10, one should observe that the estimation algorithm completes in reasonable time. It generally completes in less than a second and requires at most three seconds to complete. However, one should also notice that the error in this estimation is often quite high. Errors of twenty to forty percent are hardly acceptable. Furthermore, these estimations were taken on the output of the functions themselves. When the output probability of the consensus function is estimated, the error will likely be greater because the consensus function introduces additional reconvergent fanout (see algorithm discussion below).

Cir. Name	Inputs	Outputs	Dist. Threshold	Avg. Error	Max Error	Avg. %Error	Time
c432.isc	36	7	0	0.13167	0.3514	19.167	0.007
c432.isc	36	7	1	0.13167	0.3514	19.167	0.007
c432.isc	36	7	2	0.12203	0.32239	17.706	0.013
c432.isc	36	7	3	0.13751	0.31966	20.042	2.497
c499.isc	41	32	0	0	0	0	0.007
c499.isc	41	32	1	0	0	0	0.007
c499.isc	41	32	2	0	0	0	0.008
c499.isc	41	32	3	0	0	0	0.020
c880.isc	60	26	0	0.01915	0.05728	3.492	0.012
c880.isc	60	26	1	0.01915	0.05728	3.492	0.013
c880.isc	60	26	2	0.01867	0.05656	3.359	0.014
c880.isc	60	26	3	0.0073	0.03311	1.36	0.025
c1355.isc	41	32	0	0.00113	0.00113	0.226	0.031
c1355.isc	41	32	1	0.00113	0.00113	0.226	0.032
c1355.isc	41	32	2	0.00042	0.00079	0.085	0.037
c1355.isc	41	32	3	0	0	0	0.118
c1908.isc	33	25	0	0.02994	0.2031	6.102	0.034
c1908.isc	33	25	1	0.02994	0.2031	6.102	0.035
c1908.isc	33	25	2	0.0299	0.20295	6.094	0.162
c1908.isc	33	25	3	0.01327	0.15656	2.144	2.995
c3540.isc	50	22	0	0.17646	0.33187	45.49	0.078
c3540.isc	50	22	1	0.17646	0.33187	45.49	0.081
c3540.isc	50	22	2	0.16712	0.33211	42.98	0.107
c3540.isc	50	22	3	0.07353	0.18904	20.471	0.646

Figure Chapter 4 :-10: Results using Heuristic one on Benchmark Circuits

4.3 The ordering Algorithm

Here, a variable ordering algorithm is described which takes as initial input a circuit description in the form of a net-list file and generates a SBDD which represents the circuit. The algorithm has four basic steps.

1. Create a graph which represents the circuit.
2. Calculate ordering metrics and determine initial ordering.
3. Create SBDD using predetermined variable order.
4. Apply sifting.

4.3.1 Create Graph

The first step is to create a graph which represents the circuit. The net-list file is parsed into a graph where each node in the graph will represent a logic gate and each edge will represent a connection between gates. Node attributes will include gate labels such as its name or address, the logic type for the gate (AND, OR, NOT, XOR, etc.), and two sets of distinct edges – one set which connects the node to its fanin nodes and one which connects it to its fanout nodes. Once the graph has been created, it is "labeled" with the dependent input sets and the depth values as are needed by the probability routine.

4.3.2 Calculate Ordering Metrics

To determine variable ordering, an ordering metric must be associated with each variable. The metric used here is the same as that used in the reordering algorithm given in Chapter

3 – a weighted sum of the output probability of the consensus function taken over all outputs with the weight value being determined by support information. The difference is that the probability values and the support information are found from a circuit graph and not from an initial SBDD. The support information is readily available from the graph. In fact, it is given by the dependent sets of the output nodes which were calculated in the first step. The output probability of the consensus function, on the other hand, is more difficult to obtain. First a structural representation of the consensus function must be constructed. The consensus function can be constructed by copying the circuit graph, adding an AND gate for each output in the circuit, connecting the corresponding outputs of the two graphs (the original and the copy) through the AND gates, and lastly, connecting the inputs in such a way that the consensus is formed about the proper variable. As an example, consider the simple four input circuit in Figure Chapter 4 :-11. The consensus of the output f about the variable x_1 can be structurally represented by the circuit in Figure Chapter 4 :-12. Once the consensus function has been constructed, its output probability can be estimated using the distance based heuristic previously described, but this method for constructing the consensus function creates significant reconvergent fanout as all the inputs except the one about which the consensus is formed now fanout and reconverge at the AND gate, and thus the output probability is difficult to accurately estimate.

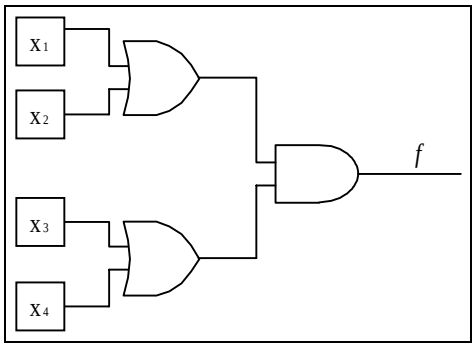


Figure Chapter 4 :-11: Simple Four Input Circuit

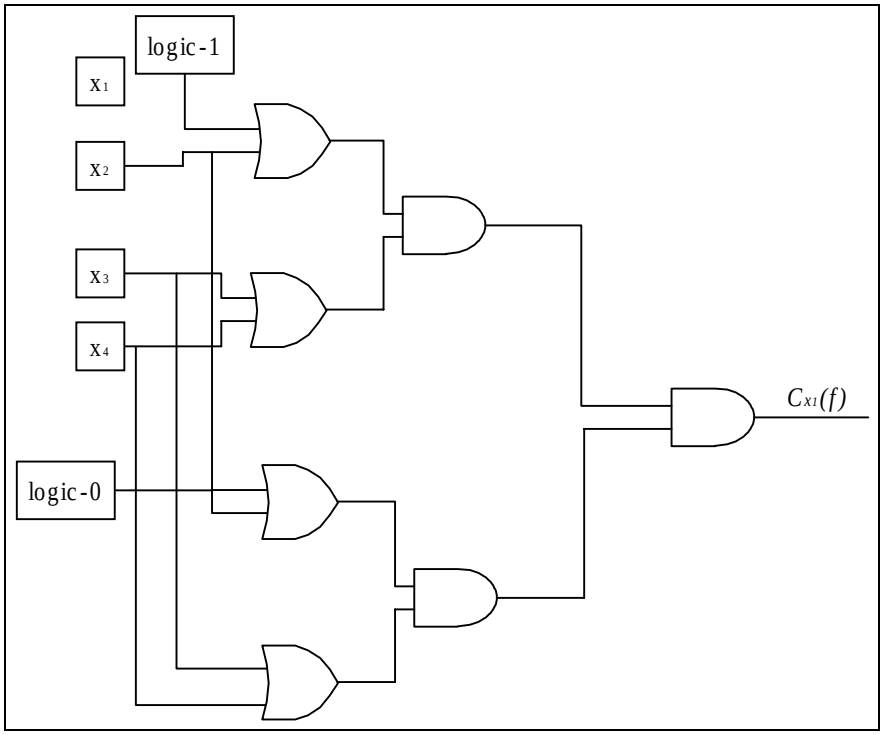


Figure Chapter 4 :-12: Consensus for Circuit in Figure Chapter 4 :-11

4.3.3 Create SBDD and Sift

Once the ordering has been determined (ascending order according to the ordering metrics), the SBDD can be created. Variables which represent the primary inputs are created according to the desired ordering, and the SBDD is created by traversing the

graph representing the circuit from inputs to outputs. At each node, a SBDD representing the given node is created by combining the SBDDs of its predecessor nodes according to the logic function of the gate which the node represents (AND, OR, NOT, XOR, etc.). After this initial SBDD is created, then standard sifting is applied as defined in [10].

4.4 Experimental Results

The ordering algorithm was applied to the ISCAS85 benchmark circuits. All experiments were performed on a 100 MHz Sun SPARCstation 20 with 160 MB of RAM using SunOS version 5.5. The table in Figure Chapter 4 :-13 contains the results of all ordering experiments performed. The column labeled *Threshold* is the distance threshold used in the probability estimation (heuristic one). The column labeled *SBDD Size* is the final size of the SBDD after sifting is applied. The column labeled *Time* is the total time in seconds to parse the net-list file, calculate ordering metrics, create a SBDD using determined ordering, and sift. And the column labeled *Reorder Size* is the size of the final SBDD created by the reordering algorithm defined in Chapter 3.

Cir. Name	Threshold	SBDD Size	Time	Reorder Size
c499.isc	0	30460	93	34432
c499.isc	1	30460	94	34432
c499.isc	2	30460	93	34432
c499.isc	3	30460	95	34432
c880.isc	0	8583	38	5816
c880.isc	1	8583	38	5816
c880.isc	2	8583	38	5816
c880.isc	3	8588	39	5816
c1355.isc	0	30461	106	34432
c1355.isc	1	30461	106	34432
c1355.isc	2	45759	172	34432
c1355.isc	3	41016	146	34432
c1908.isc	0	9709	30	8762
c1908.isc	1	9709	31	8762
c1908.isc	2	9709	40	8762
c1908.isc	3	9534	789	8762
c3540.isc	0	66544	342	43564
c3540.isc	1	66544	342	43564
c3540.isc	2	66954	349	43564
c3540.isc	3	48348	842	43564

Figure Chapter 4 :-13: Ordering Experiments on ISCAS85 Benchmark Circuits

4.5 Interpretation and Evaluation

Ideally, the ordering technique defined here would produce the same orderings and thus the same SBDDs as the reordering technique defined in Chapter 3. These two methods use the same ordering heuristics (output probability of the consensus and support information). However, the probability algorithm used in the ordering technique can only estimate where the algorithm used in reordering can compute these values exactly. While some of the results in Figure Chapter 4 :-13 compare reasonably well to the reordering results, the results of the ordering experiments are not completely satisfying. First, c432 is noticeably absent from the sample set. When the algorithm was attempted on this

circuit, the SBDD became very large and the algorithm would not complete. Also, the resultant SBDD for c3540 is quite large except when the largest threshold is applied, and even then it requires a significant amount of time and most of the reduction is achieved through sifting. It should be noted that c432 and c3540 are the two circuits for which the probability estimation algorithm produced the poorest estimates (see Figure Chapter 4 :-10). Therefore, before this ordering technique can be used in practical applications, a better means of determining the output probability form a circuit description must be developed.

Chapter 5 : Conclusions and Future Work

5.1 Conclusions

This work examined the use of function output probabilities to form heuristics which could be applied to decision diagrams so that they could represent complex Boolean equations in an effective manner. Algorithms, which use the output probability of the consensus function in conjunction with support information, to perform both variable reordering and variable ordering were proposed, and experimental results using benchmark circuits were given. In addition, a new algorithm to compute output probabilities from BDDs was presented. Lastly, previously proposed methods for estimating the output probabilities from circuit descriptions were implemented and empirically tested to determine their accuracy.

In Chapter 2, a new depth-first algorithm for computing output probabilities from BDD structures was detailed and compared to the previously developed breadth-first algorithm. Both algorithms have linear complexity with respect to the size of the BDD on which they operate, but the new algorithm was shown to be superior insofar as it can efficiently handle the shared sub-graphs of SBDDs as well as be applied to attributed edge BDDs. This new algorithm has a general utility which extends beyond the variable ordering/reordering problem; it may find application in such areas as combinational logic synthesis, spectral analysis, and the design and analysis of low power circuitry.

Chapter 3 presents a new variable reordering algorithm for shared binary decision diagrams with negative-edge attributes. This algorithm takes a SBDD as input and

attempts to reduce its size (number of nodes) through variable reordering. This method is a heuristic based method which uses a weighted sum of the probabilities of the consensus function (weights determined by support information) to select an appropriate ordering sub-space for the standard sifting algorithm. Experimental results were given to show that the algorithm is effective in reducing the size of SBDDs used to represent many benchmark circuits. Comparisons were made between the new algorithm and a well known simulated annealing based algorithm. While the simulated annealing algorithm produced smaller SBDDs, the new algorithm was shown to produce reasonable SBDDs in much less time.

In Chapter 4, the reordering heuristics were modified so that they could be applied to the variable ordering problem, and thus eliminate the need to create an initial SBDD for input to the reordering algorithm. The ordering algorithm requires that the output probability of the consensus function be calculated (or at least estimated) from a circuit description. Previously suggested estimation techniques were implemented and tested; however, they proved not to be extremely accurate in all cases. This deficiency caused the ordering algorithm to perform poorly in some cases, but this work should further highlight the need for efficient algorithms to accurately estimate output probabilities from circuit descriptions.

5.2 Future Work

The first area that needs to be addressed by future research is the estimation of output probabilities from circuit descriptions or net-list files. One possible solution is a hybrid

method that actually uses BDDs during the calculation process. The current method, described in Chapter 4, identifies "supergates" for each reconvergent node, and then exhaustively simulates over all fanout inputs to the supergate. This phase could be replaced by using a BDD to represent the supergate and applying the algorithm proposed in Chapter 2 to determine the output probability of the supergate. The depth-first algorithm would have to be implemented in its general form as the probabilities of the inputs to the supergate are not necessarily even distributions. Also, this new estimation algorithm could become impractical if the supergates become too large; therefore, it may have to be combined with the previously defined heuristics to limit the size of the supergate, but the use of BDDs should allow larger thresholds, and thus more accurate estimations.

The second area which requires further investigation is the relationship between the support area and variable orderings. It was determined that for some functions smaller SBDDs could be produced by pulling the variables which occur in few supports up in the ordering, while for other functions smaller SBDDs were produced by allowing these variables to fall into their more natural position. A certain amount of conjecture was offered to explain this relationship, but further investigation is needed before it can be fully specified and an optimum threshold determined (if such an optimum threshold exists).

Lastly, the set of ordering heuristics could be expanded so that they could produce better orderings. One possibility is the use of the Haar spectral coefficients. In a recent paper [27], these values were related to the output probabilities of a set of equivalence

functions. Inherent in these equivalence functions was an implied variable ordering. Perhaps this ordering could be exploited and even combined with the ordering generated by the consensus function to produce a more general ordering metric.

In addition, the ordering heuristics might be extended by exploiting variable symmetry. Variable symmetry among many variables is difficult to detect, but recently it was shown that output probabilities could be used to form a necessary, but not sufficient, condition for variable symmetry [15]. This symmetry might imply a close relationship among variables and suggest that these variables should be grouped together in the ordering.

References:

[1] Akers, S. B., Binary Decision Diagrams, *IEEE Transactions on Computers*, vol. C-27, no. 6, pp. 509-516, June 1978.

[2] Bryant, R. E., Graph-Based Algorithms for Boolean Function Manipulation, *IEEE Transactions on Computers*, vol. c-35, no. 8, pp. 677-691, August 1986.

[3] Bryant, R., Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams, *ACM Computing Surveys*, Vol. 24, No. 3, September 1992.

[4] Lee, C. Y., Representation of Switching Circuits by Binary-Decision Programs, *Bell System Technical Journal*, vol. 38, pp. 985-999, July 1959.

[5] Minato, S., Graph-Based Representations of Discrete Functions, in **Representations of Discrete Functions,** edited by Tsutomu Sasao and Masahiro Fujita, Kluwer Academic Publishers, Boston, Massachusetts, 1996.

[6] Brace K. S., Rudell, R. L., Bryant R. E., Efficient Implementation of a BDD package, *Proceedings 27th ACM/IEEE Design Automation Conference (DAC)*, pp. 40-45, 1990.

[7] Minato, S., Ishiura, N., Yajima, S., Shared Binary Decision Diagrams with Attributed Edges for Efficient Boolean Function Manipulation, *Proceedings 27th ACM/IEEE Design Automation Conference (DAC)*, pp. 52-57, 1990.

[8] Beate, B., Wegener, I., Improving the Variable Ordering of OBDDs Is NP-Complete, *IEEE Transactions on Computers*, vol. 45, no. 9, pp. 993-1002, September 1996.

[9] Fujita, M., Fujisawa, H., Kawato, N., Evaluation and Improvement of Boolean Comparison Method based on Binary Decision Diagrams, *Proceedings IEEE/ACM International Conference on Computer Aided Design*, pp. 2-5, Nov. 1988.

[10] Rudell, R., Dynamic Variable Ordering for Ordered Binary Decision Diagrams, *Proceedings IEEE/ACM International Conference on Computer Aided Design*, pp. 42-47, 1993.

[11] Beate, B., Löbbing M., Wegener, I., Simulated Annealing to Improve Variable Orderings for OBDDs, *Proceedings ACM/IEEE International Workshop on Logic Synthesis*, pp. 5-1 -- 5-10, 1995.

[12] Thornton, M., Nair, V., Efficient Calculation of Spectral Coefficients and Their Applications, *IEEE Transactions on CAD/ICAS*, vol. 14, no. 11, pp. 1328-1341, November 1995.

[13] Shannon, C., A Symbolic Analysis of Relay and Switching Circuits, Transactions AIEE, vol. 57, pp. 713-723, 1938.

[14] De Micheli, G., **Synthesis and Optimization of Digital Circuits**, McGraw-Hill, Inc., New York, New York, 1995.

[15] Thornton M., Moore R., Applications of Circuit Probability Computation Using Decision Diagrams, to appear in *Proceedings 1997 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, August 20-22, 1997.

[16] Ghosh, A., Devadas, S., Keutzer, K., White, J., Estimation of Average Switching Activity in Combinational and Sequential Circuits, *Proceedings of the ACM/IEEE Design Automation Conference*, pp. 253-259, June 1992.

[17] Najm, F., A Survey of Power Estimation Techniques in VLSI Circuits, *IEEE Transactions on Computer Aided Design,* vol. CAD-2, no. 4, pp. 446-455, December 1994.

[18] Predram, M., Power Minimization in IC Design: Principles and Applications, *ACM Transactions on Design Automation of Electronic Systems*, vol. 1., no. 1, 1996.

[19] Parker, K., McClusky, E., Probabilistic Treatment of General Combinational Networks, *IEEE Transactions on Computers*, vol. c-24, pp. 668-670, June 1975.

[20] Goldstein, H., Controllability / Observability of Digital Circuits, *IEEE Transactions on Circuits and Systems*, vol. 26, no. 9, pp. 685-693, September 1979.

[21] Savir J., Ditlow, G., Bardell, P., Random Pattern Testability, *IEEE Transactions on Computers*, vol. C-33, no. 1, pp. 1041-1045, January 1984.

[22] Critic, M., Estimating Dynamic Power Consumption of CMOS Circuits, *Proceedings of the International Conference on Computer-Aided Design*, pp. 534-537, 1987.

[23] Seth, S., Pan, L., Agrawal, V., PREDICT – Probabilistic Estimation of Digital Circuit Testability, *Proceedings of the Fault Tolerant Computer Symposium*, pp. 220-225, 1985.

[24] Krishnamurthy, B., Tollis, I., Improved Techniques for Estimating Signal Probabilities, *IEEE Transactions on Computers*, vol. 38, no. 7, pp. 1041-1045, July 1989.

[25] Chakravarty, S., On the Complexity of Using BDDs for the Synthesis and Analysis of Boolean Circuits, *Proceedings of the 27th Annual Conference on Communication, Control and Computing*, pp. 730-739, 1989.

[26] Jain, J., Arithmetic Transform of Boolean Functions, in **Representations of Discrete Functions,** edited by Tsutomu Sasao and Masahiro Fujita, Kluwer Academic Publishers, Boston, Massachusetts, 1996.

[27] Thornton, M., Modified Haar Transform Calculation Using Digital Circuit Output Probabilities, to appear in *Proceedings of the International Conference on Information, Communication, and Signal Processing*, September 1997.

Appendix A: Supports of Benchmark Circuits

Chapter 3 defines the support of a function as the set of variables on which the function depends. It also defines the ideas of “support area” and “sparseness of support”. The figures in this appendix illustrate the support areas of the ISCAS85 benchmark circuits graphically. In each of the figures, if the function, f_j, is supported by the variable, x_i, then an ‘X’ will be placed in the cell *i, j*. The circuits c499 and c1355 are not shown as they have complete coverage in their support area (*i.e.*, all functions depend on all the variables).

	51	66	60	54	63	75	57	21	6	36	45	27	12	42	15	72	24	9	39	48	69	33	30	3	18
##	X	X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X	X			X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X					X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X			X	X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X		X	X		X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X			X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X				X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X				X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X		X			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X		X		X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X		X		X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##		X	X	X		X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##				X		X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##		X		X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##				X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##			X	X		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X	X	X	X		X	X	X				X		X		X	X			X			X	X
##	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
##	X	X	X	X	X		X			X	X	X		X		X			X	X		X	X		

Figure A-1: Support Area for c1908

	421gat	431gat	430gat	370gat	432gat	329gat	223gat
4gat	X	X	X	X	X	X	X
1gat	X	X	X	X	X	X	X
11gat	X	X	X	X	X	X	X
17gat	X	X	X	X	X	X	X
24gat	X	X	X	X	X	X	X
30gat	X	X	X	X	X	X	X
37gat	X	X	X	X	X	X	X
43gat	X	X	X	X	X	X	X
50gat	X	X	X	X	X	X	X
56gat	X	X	X	X	X	X	X
63gat	X	X	X	X	X	X	X
69gat	X	X	X	X	X	X	X
76gat	X	X	X	X	X	X	X
82gat	X	X	X	X	X	X	X
89gat	X	X	X	X	X	X	X
95gat	X	X	X	X	X	X	X
102gat	X	X	X	X	X	X	X
108gat	X	X	X	X	X	X	X
8gat	X	X	X	X	X	X	
21gat	X	X	X	X	X	X	
34gat	X	X	X	X	X	X	
47gat	X	X	X	X	X	X	
60gat	X	X	X	X	X	X	
73gat	X	X	X	X	X	X	
86gat	X	X	X	X	X	X	
99gat	X	X	X	X	X	X	
112gat	X	X	X	X	X	X	
14gat	X	X	X	X	X		
27gat	X	X	X	X	X		
40gat	X	X	X	X	X		
53gat	X	X	X	X	X		
66gat	X	X	X	X	X		
79gat	X	X	X	X	X		
92gat	X	X	X	X	X		
105gat	X	X	X	X	X		
115gat	X	X	X	X	X		

Figure A-2: Support Area for c432

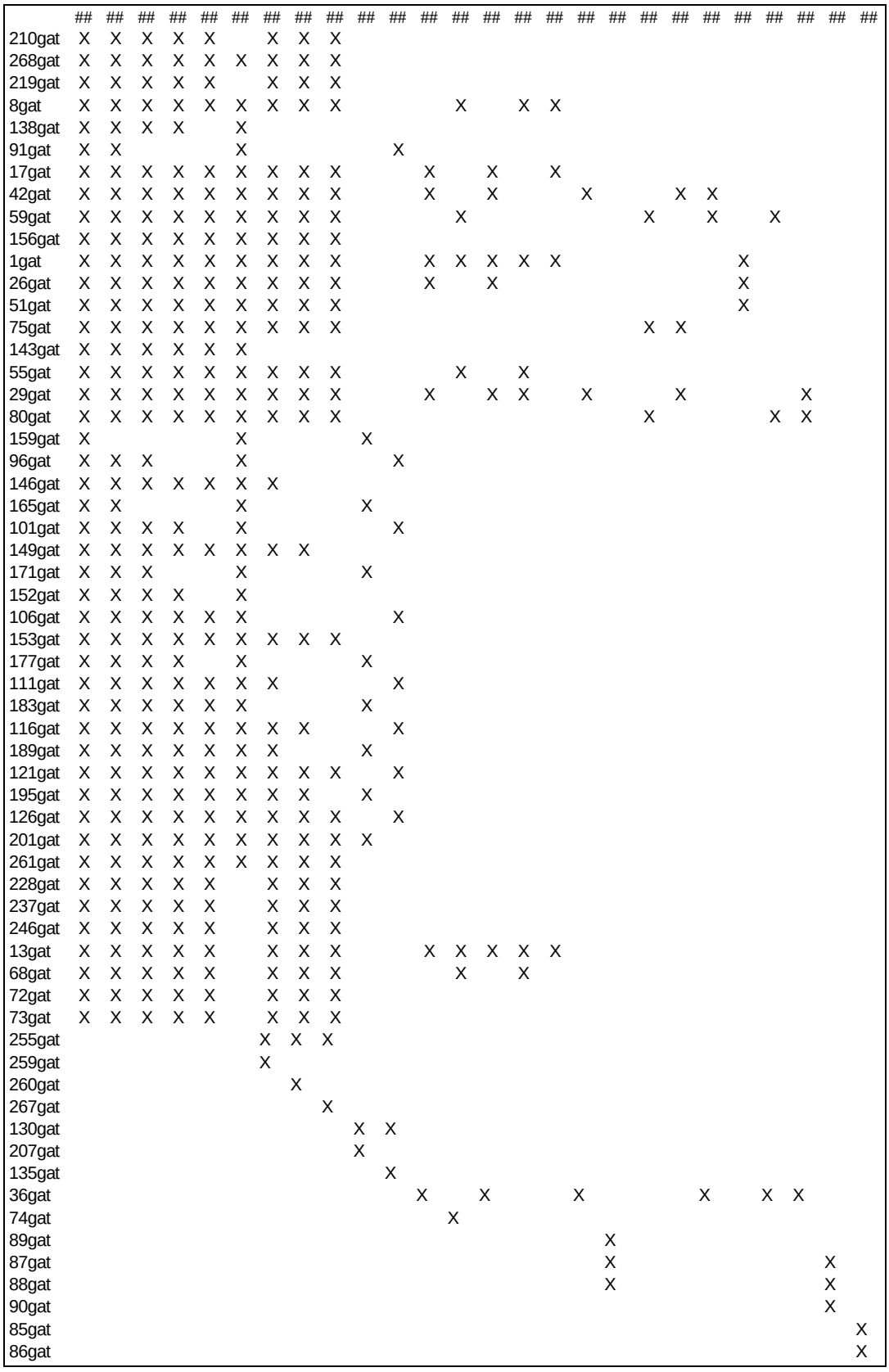
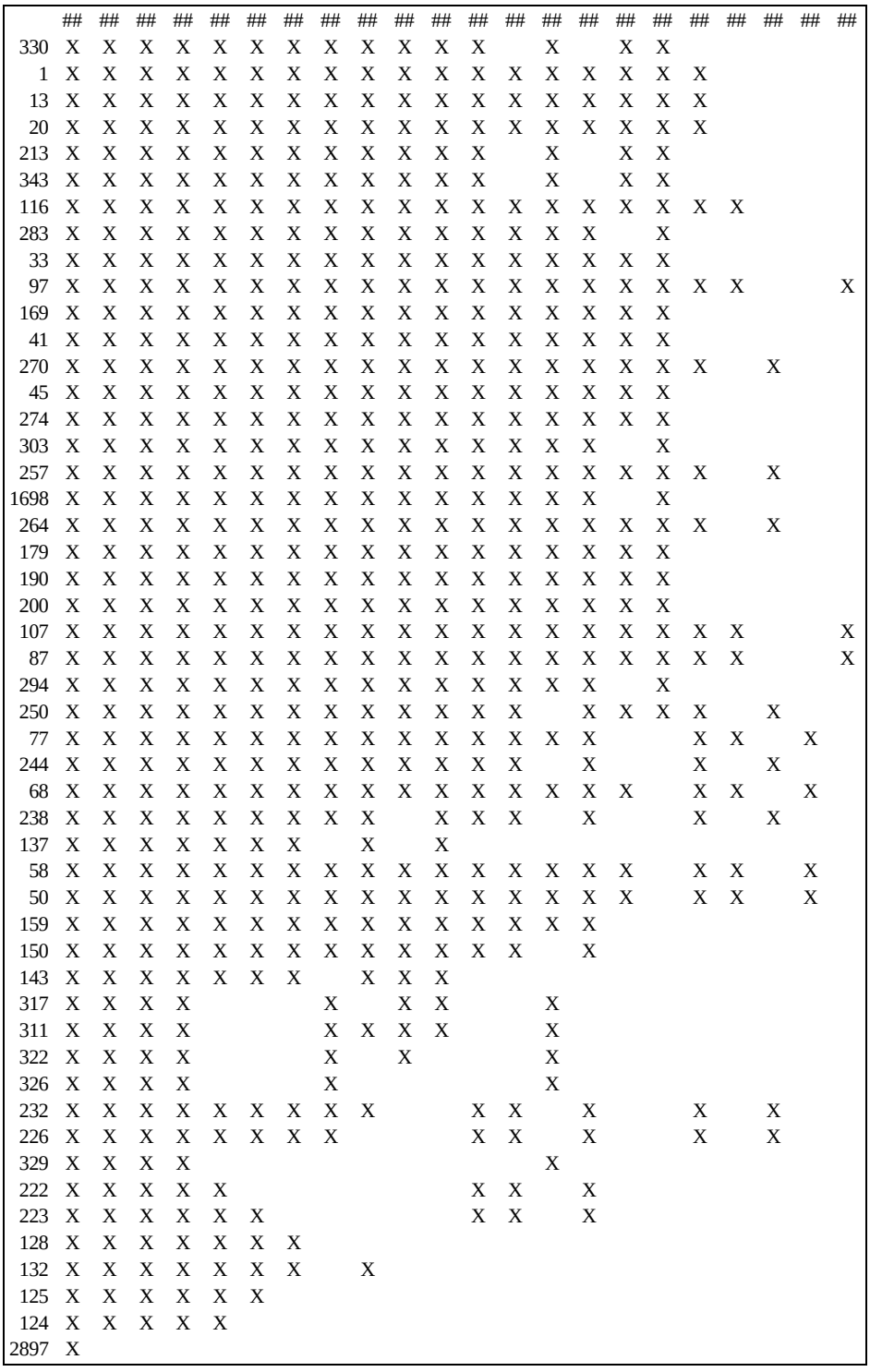


Figure A-3: Support Area for c880



	##	##	##	##	##	##	##	##	##	##	##	##	##	##	##	##	##	##	##	##	##
330	X	X	X	X	X	X	X	X	X	X	X	X		X		X	X				
1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X			
13	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X			
20	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X			
213	X	X	X	X	X	X	X	X	X	X	X	X		X		X	X				
343	X	X	X	X	X	X	X	X	X	X	X	X		X		X	X				
116	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		
283	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X				
33	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X				
97	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X
169	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X				
41	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X				
270	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	
45	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X				
274	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X				
303	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X				
257	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	
1698	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X				
264	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	
179	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X				
190	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X				
200	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X				
107	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X
87	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X
294	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X					
250	X	X	X	X	X	X	X	X	X	X	X	X		X	X	X	X		X		
77	X	X	X	X	X	X	X	X	X	X	X	X	X	X			X	X		X	
244	X	X	X	X	X	X	X	X	X	X	X	X		X			X		X		
68	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X		X	
238	X	X	X	X	X	X	X	X	X		X	X	X		X			X		X	
137	X	X	X	X	X	X	X		X		X										
58	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X		X
50	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X		X	X		X
159	X	X	X	X	X	X	X	X	X	X	X	X	X	X							
150	X	X	X	X	X	X	X	X	X	X	X	X	X		X						
143	X	X	X	X	X	X	X		X	X	X										
317	X	X	X	X				X		X	X			X							
311	X	X	X	X				X	X	X	X			X							
322	X	X	X	X				X		X				X							
326	X	X	X	X				X						X							
232	X	X	X	X	X	X	X	X	X			X	X		X			X		X	
226	X	X	X	X	X	X	X	X				X	X		X			X		X	
329	X	X	X	X										X							
222	X	X	X	X	X							X	X		X						
223	X	X	X	X	X	X						X	X		X						
128	X	X	X	X	X	X	X														
132	X	X	X	X	X	X	X		X												
125	X	X	X	X	X	X															
124	X	X	X	X	X																
2897	X																				

Figure A-4: Support Area for c3540

Appendix B: Definition of Structures

This appendix describes some of the C structures used in the experiments described in the main body of this work. These structures include the basic structures to define Shared Binary Decision Diagrams and those used to specify circuit graphs.

Structure: SBDD

This structure can be used to define a Shared Binary Decision Diagram (SBDD). It contains all the output BDDs, the input variables, and the labels for each input and output.

Defined in: utils.h

Fields:

*bdd * outs*

This *outs* field is an array of BDDs, one for each output. The array is dynamically allocated using the *init_sbdd* routine; however, this initialization is normally done by the routines which parse the input files.

*char * *outLabels*

This field is an array of character strings which give the labels for the outputs contained in the *outs* array. Each string in the array contains the label for the output which has the same index in the *outs* array (*outLabels[i]* contains the label for *outs[i]*). All memory for this array is allocated by *init_sbdd*.

int numOuts

This field contains the number of outputs in the SBDD. It will also be the number of elements in the *outs* array and the *outLabels* array.

*bdd * ins*

This *ins* field is an array of BDDs which contains the input variables. The array is dynamically allocated using the *init_sbdd* routine; however, this initialization is normally done by the routines which parse the input files. The inputs are not necessary ordered in the array to match the ordering of the SBDD. They are generally placed in the array as they are created and will not be moved if reordering is performed.

*char * *inLabels*

This field is an array of character strings which give the labels for the inputs contained in the *ins* array. Each string in the array contains the label for the input which has the same index in the *ins* array (*inLabels[i]* contains the label for *ins[i]*). All memory for this array is allocated by *init_sbdd.*

int numIns

This field contains the number of inputs in the SBDD. It will also be the number of elements in the *ins* array and the *inLabels* array.

int maxLabelLen

This field contains the maximum number of characters allowed for labels. In other words, it is the number of characters allocated for each element in the *inLabels* and *outLabels* arrays.

Functions for manipulating:

init_sbdd	--	Allocates all memory for the SBDD.
free_sbdd	--	Releases all memory contained in a SBDD.
read_file_all	--	Parses both .isc and .pla files and creates a SBDD.
pla2bdd_all	--	Parses .pla files and creates a SBDD.
isc2bdd_all	--	Parses .isc files and creates a SBDD.
dag2sbdd	--	Creates a SBDD from a DAG structure.
sbdd_output_prob	--	Calculates probability of all outputs in a SBDD.
count_nodes	--	Counts the number of nodes in a SBDD.

Structure: DAG

This structure can be used to create a graph which represents a circuit. Each node in the graph represents a logic gate and each edge represents a connection between gates.

Defined in: dagnode.h

Fields:

*DAG_VERT ** nodeTable*

This field is an array of pointers to all nodes in the graph. The nodes will be arranged in the same order as they are encountered in the net-list file. This array is dynamically

allocated when the graph is created. Each DAG node is allocated individually and may be indexed by a number of arrays including *nodeTable*, *inputs*, and *outputs*.

int numNodes

numNodes is the total number of nodes in the graph or the number of elements in the *nodeTable* array.

*DAG_VERT ** inputs*

This field is an array of pointers to the input nodes of the graph. This array is dynamically allocated when the graph is created.

int numIns

numIns is the number of input nodes in the graph or the number of elements in the *inputs* array.

*DAG_VERT ** outputs*

This field is an array of pointers to the output nodes of the graph. This array is dynamically allocated when the graph is created.

int numOuts

numOuts is the number of output nodes in the graph or the number of elements in the *outputs* array.

Functions for manipulating:

net2dag	--	Parses net-list file (.isc) and creates a DAG. (all needed memory is allocated by this routine)
delete_dag	--	Releases all memory associated with a DAG.
dag2sbdd	--	Creates a SBDD from a DAG.
netprob	--	Calculates output probabilities using a DAG.
copyDAG	--	Copies a DAG for the consensus function.
buildConFunct	--	Creates a structural representation of the consensus.
conProb	--	Calculates output probabilities of the consensus.

Structure: DAG_VERT

This structure is used to defined the individual nodes in the DAG structure. Each DAG_VERT will represent a gate in the circuit.

Defined in: dagnode.h

Fields:

int address

This field gives the ISCAS address for the gate. It is copied directly from the net-list file.

*char *name*

This field gives the ISCAS name for the gate. It is copied directly from the net-list file.

int vtype

This field gives the type of gate that the node represents (AND, OR, XOR, etc.). These types are defined in tokdef.h .

int fanout

This field gives the number of fanouts from the node.

int fanin

This field gives the number of fanins to the node.

*int * fanin_lst*

This field is a list of fanin addresses. It will seldom need to be used once the DAG is fully created because the fanin nodes can be easily reached using the *fanin_ptrs*.

*char *finame*

If the node represents a *FROM* stem, then this field will contain the name of the gate from which it fanouts. Again, this field is seldom used because the *fanin_ptrs* provide a more direct route to the fanin stems.

int color

This field is used to "mark" the node during traversals so that it is easy to tell if a node has been visited or not.

int fwd_pnt_num

This field keeps track of how many forward pointers have been assigned. It is used only during the creation of the DAG.

int node_ident

This field is an internal identifier. Each node is assigned a unique ID. These ID's will generally correspond to the nodes position in the *nodeTable* of the DAG.

*DAG_VERT ** fanout_ptrs*

This field is an array of pointers to the given nodes fanout points.

*DAG_VERT ** fanin_ptrs*

This field is an array of pointers to the given nodes fanin points.