1-1-1982

A study of computations on binary decision diagrams.

James Patrick McHugh

A STUDY OF COMPUTATIONS ON BINARY DECISION DIAGRAMS

by

James Patrick McHugh

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Electrical Engineering

Lehigh University
1982

CERTIFICATE OF APPROVAL

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering.

3/12/82
(date)

Professor in Charge

Chairman of Department

ACKNOWLEDGEMENT

I am grateful to Professor A. K. Susskind for his patient guidance in the research and preparation of this paper. I also wish to thank Connie Weaver for typing the final manuscript and my wife, Mary, for drafting the illustrations.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Binary decision diagrams are tools with which boolean functions may be analyzed without the need to resort to intricate algebraic manipulations. Their structure allows them to be represented on a computer in a straightforward manner. This paper explores the costs, in terms of CPU time and memory usage, associated with manipulation of these diagrams within a computer.

An algorithm is developed which uses a diagram of a function to analyze the movements of vertices between the true and false bodies of that function in the presence of stuck-variable faults. From this information a set of fault-detection tests is derived. A computer program which implements this algorithm is then presented. The program is designed to provide a worst-case measurement of the CPU time required for analysis. The performance of this program is measured for binary trees with from one to twelve variables. The results indicate that although the algorithm is effective, further development is needed to arrive at a more efficient implementation of the algorithm.

1. INTRODUCTION AND BACKGROUND

Binary decision diagrams are a tool with which large boolean functions may be manipulated and analyzed without the need to resort to complex algebraic manipulations. They were first introduced by Akers [1]. The key feature of these diagrams is that they can be used to perform functional rather than implementation-dependent analysis of boolean functions. Specifically, they can be used to analyze the effects of faults upon a function and yield results which apply to all implementations of the function. Before pursuing this point, some of the basic nomenclature associated with binary decision diagrams will be introduced.

A binary decision diagram is a directed acyclic graph [1]. Unless otherwise specified, all edges are directed downward, that is, away from the root node. Each node in the diagram may have any number of entering edges. Only and exactly two edges leave all nodes other than the terminal nodes. There are two terminal nodes, labeled zero and one. (These may be repeated at several places on a diagram for clarity.) Associated with each node other than the terminal nodes is a variable which is called the node variable. It is possible for one variable to be associated with more than one node; however, a restriction will be placed on this condition later. Consider the diagram of the function $\overline{A}B + \overline{B}C$ (Figure 1). The function f is depicted as entering the root node of the diagram. From there a number of paths may be traced through the diagram.

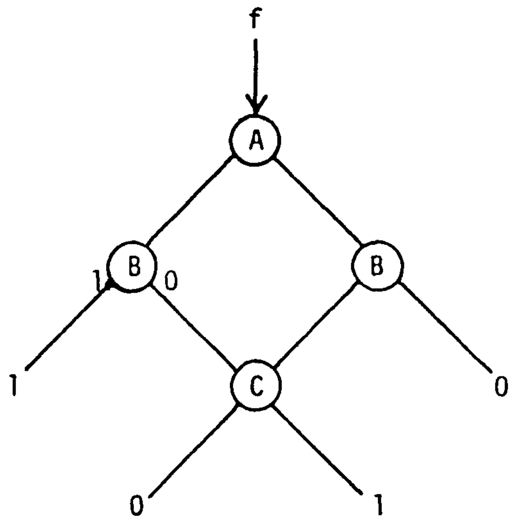

Figure 1 - Binary Decision Diagram Of f = $\bar{A}B + \bar{B}C$

The value of A determines which edge is chosen for traversal to an offspring node. Unless otherwise specified a node variable with value zero selects the left exiting edge. A value of one selects the right exiting edge. Suppose that A is equal to zero. The next node in the path is then the leftmost of the two B nodes. The left and right outgoing edges of this B node are labeled with a one and a zero respectively to indicate that they differ from the convention of zero-edge on the left and one-edge on the right. Such a change is usually made to neaten the diagram. Now suppose that B is equal to zero. The next node encountered is then C. If C is equal to one then the next node encountered is the one terminal node. Thus the value of the function is one for the input combination A = 0, B = 0, C = 1. The path which has been traced through the diagram is called an activation path. Since this activation path ends in a one it represents a minterm of the function. That minterm is $\overline{A}\,\overline{B}C$. Each vertex of the function can be represented by an activation path through the diagram from the root to a terminal node. The value of the terminal node which is reached is equal to the value of the vertex, that is, the value which the function assumes in response to that input combination. For example, the vertex ABC produces an activation path which ends at a zero node. Thus, the value of the function at this vertex is zero, as it is for the vertices $AB\overline{C}$ and $A\overline{B}\,\overline{C}$.

The total number of vertices in the false and true bodies of a function may be determined as follows (see Figure 2). The function $f = \overline{A}B + \overline{B}C$ contains three variables, therefore it has eight vertices. These are depicted as entering the root node from f.

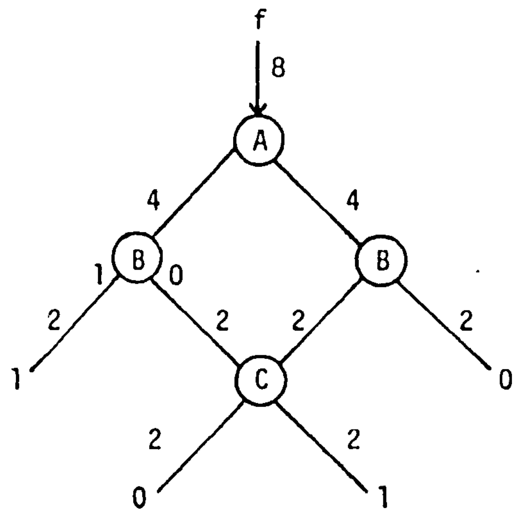

Figure 2 - Counting Of Vertices

Upon exiting the root node, A, these vertices divide equally between the two outgoing edges. This is because one-half of the vertices contain $\bar{A}$ and one-half contain A. The four vertices which enter the leftmost B node are again divided equally, with one-half entering the one terminal node and one-half continuing on to node C. The two vertices which enter the one terminal node are $\bar{A}B\,\bar{C}$ and $\bar{A}BC$. These are in the true body of the function. Likewise, the four vertices which enter the rightmost B node are divided equally. Two ($AB\,\bar{C}$ and ABC) enter the zero terminal node (they are in the false body of the function)and two continue on to node C. Node C has four entering vertices. This number is obtained by summing the numbers of vertices associated with each of the edges which enter node C. The four entering vertices are $\bar{A}\,\bar{B}\,\bar{C}$, $\bar{A}\,\bar{B}C$, $A\,\bar{B}\,\bar{C}$ and $A\,\bar{B}C$. These four divide equally between the two edges which leave the C node. In fact, closer observation shows that if each entering edge is considered separately, the vertices which enter along that edge divide equally between the two outgoing edges. Of the four vertices which enter the C node, two ($\bar{A}\,\bar{B}\,\bar{C}$ and $A\,\bar{B}\,\bar{C}$) pass through to the zero node and two ($\bar{A}\,\bar{B}C$ and $A\,\bar{B}C$) pass through to the one node. The total number of vertices in the true and false bodies of the function may be determined by summing the total numbers of vertices which enter one and zero nodes, respectively. In this case an equal number, four, are in each body.

A procedure for deriving a binary decision diagram from an input-output specification of a function will not be presented here, as several have been described by Akers in [2]. However, there are two situations which may be encountered during the derivation of a diagram which are of importance here. The first relates to the restriction mentioned earlier on the number of nodes which may be associated with a variable. Consider the arbitrary diagram of Figure 3. An attempt to trace the path which has been highlighted

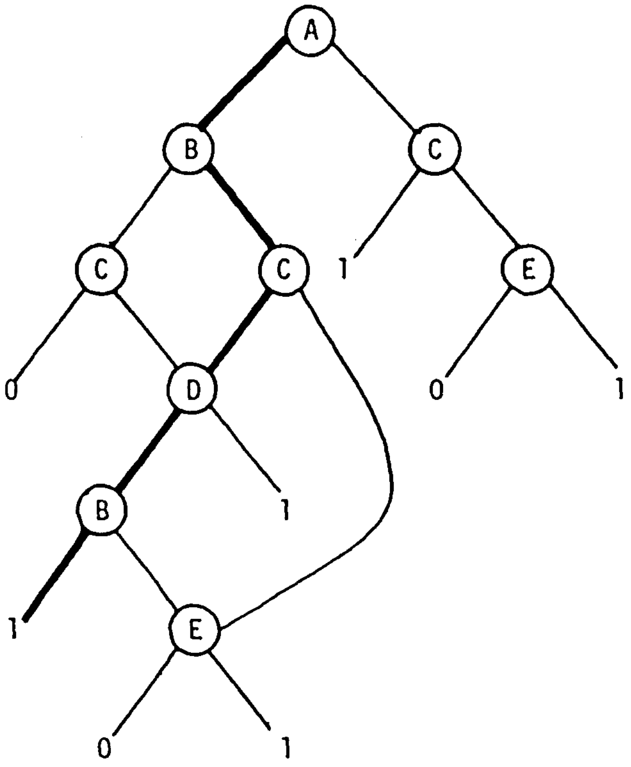

Figure 3 - Binary Decision Diagram With Unfeasible Paths

leads to a contradiction. The first time a B node is encountered the value of B is one. However, the second time B is zero, yet the same variable B is associated with both nodes. This contradiction exists because the variable B appears twice in a path from the root of the diagram to a terminal node. An activation path which requires a single variable to assume both the values zero and one is called an unfeasible path. Such a path can never be activated within the context of a real boolean function, although it may exist within a valid binary decision diagram for that function. Unfeasible paths will not be allowed in the diagrams which will be considered in later chapters because of the confusion they can cause when tracing the movements of vertices through a diagram in the manner described earlier.

A second situation which may be encountered during the derivation of a diagram is depicted in Figure 4. Figure 4(a) can be recognized as another version of Figure 1. Note that both B nodes have the same zero-offspring. Rather than repeat it, it may be drawn once. The zero-edges of both B nodes are then redirected to this one subgraph, as in Figure 4(b). This procedure is valid for any size of subgraph. Then, to make the reduced diagram appear neater, the zero- and one-edges of the leftmost B node are reversed and labeled, as in Figure 4(c). A motivation for reducing diagrams to the greatest extent possible will be presented in a later chapter.

The information presented in this chapter is basic to the understanding of the chapters which follow. More detailed presentations may be found in [1] and [2].

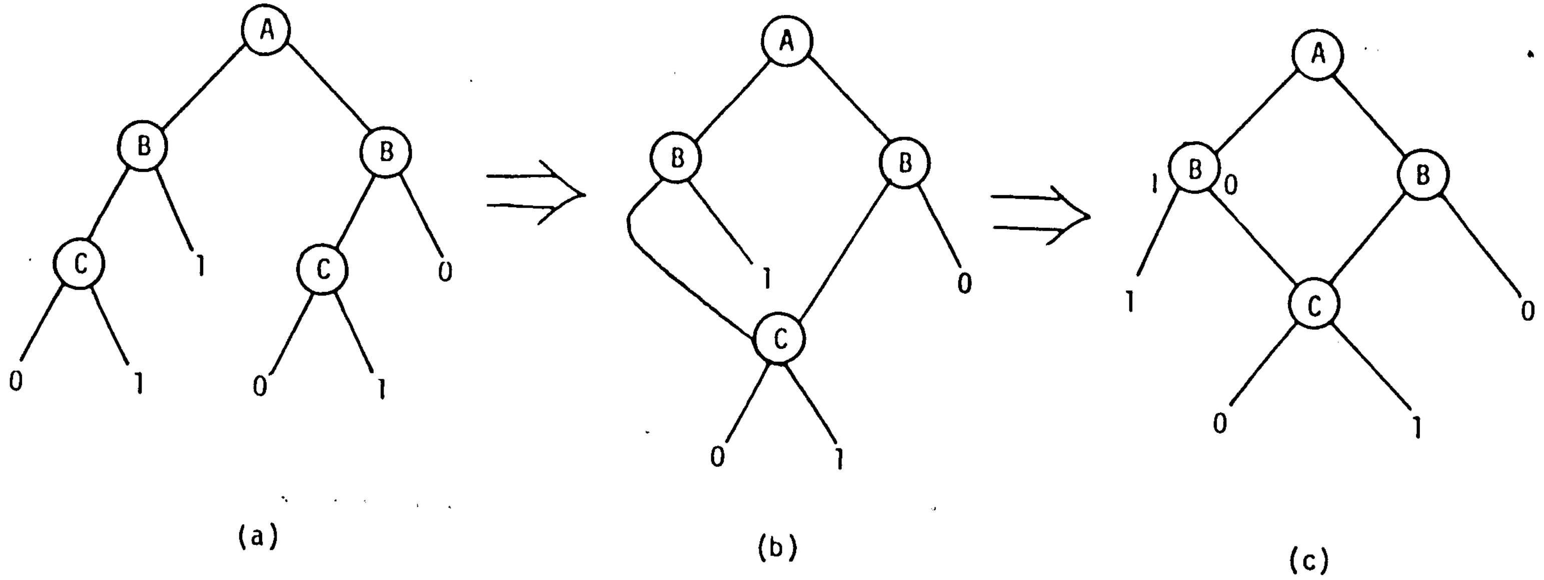

Figure 4 - Merging of Identical Subgraphs

2. OBJECTIVES

The primary objective of this study is to explore the feasibility of computer manipulation of binary decision diagrams. The manipulations to be performed relate to analyzing the effects of stuck-variable faults on the diagram (and therefore on the function) and to the generation of tests which will detect the presence of these faults. Several subgoals are presented by this task. The first of these is to study the movement of vertices between the true and false bodies of a function in the presence of one or more stuck-variable faults. Once this movement has been determined the information gained is to be used to generate tests which will detect the presence of the faults. The third subgoal is to gain familiarity with the data structures and programs needed to manipulate binary decision diagrams within the computer. The fourth and most important subgoal is to gain familiarity with the costs associated with these manipulations. The costs will be specified chiefly in terms of CPU time and memory space used by the programs and data.

In order to meet the goals outlined above, several tasks must be completed. Primary among these is the development of algorithms which perform the desired manipulations. Data structures must also be developed to represent binary decision diagrams within the computer memory. Once the algorithms and data structures have been specified a program package must be developed which implements the algorithms. Finally, a method of evaluating the performances of both program and data structures must be developed and then implemented. The details and implementations of each of these tasks are presented in the next five chapters.

3. ALGORITHMS

The previous chapter introduced two algorithms which are to be implemented on the computer. The first algorithm must determine which vertices of a function move from its true body to its false body in the presence of a fault condition (single or multiple) and which vertices move from its false body to its true body under the same fault condition. This determination must be made in such a way that no cancellation occurs between vertices moving in opposite directions. Only in this way can a reliable count be made of vertices whose values change in the presence of a fault. Once these changed vertices have been isolated,the second algorithm is needed to derive from them a set of tests for the specified fault conditions. A third algorithm, which is peripheral but essential to these two main tasks, is one which will enter the binary decision diagram into computer memory. These three algorithms constitute the main functions of the program package.

3.1 Entry of a Diagram into Memory

The process of entering a binary decision diagram into memory is described first so that the reader may become familiar with the characteristics of the diagrams which are to be manipulated. It is beyond the scope of this thesis to attempt to develop or implement an algorithm which would derive a binary decision diagram from a boolean function or some other input-output specification. Instead, an algorithm is presented which will accept, validate and enter into memory a previously derived diagram so that it can be manipulated. Main memory has been chosen over peripheral mass storage because of its greater access speed. (Full details of the data structures used to represent a diagram are presented in Chapter 5.)

The basic diagram entry algorithm is quite simple. First the root node is entered, then its left subgraph is entered, then its right subgraph is entered. Each subgraph is entered in the same manner. This recursive method of entering the diagram (the entry procedure invokes itself) constitutes a preorder traversal of the diagram [3]. Nodes are created and edges are established during the traversal in response to information entered by the user.

Certain criteria must be met before a new node or edge can be generated. The criteria which a new node must meet are:

1. The node variable must be a legal variable. A legal variable has a name which falls within the user-specified length requirement and, in the case of the root node, is not either the zero or one terminal node.

2. An offspring node variable must not already appear in any path from the root to one of its parent nodes. That is, cycles and unfeasible paths are not allowed in these diagrams.

The restriction against cycles was imposed by Akers [1], presumably to prevent ambiguity in the evaluation of a function by means of a diagram. This and the additional constraint against unfeasible paths are necessary to avoid confusion when vertices of the function are counted and traced through the diagram.

When an offspring of a node already exists within a part of the diagram which has previously been entered, as when identical subgraphs have been merged, the offspring node and the subgraph of which it is the root need not be entered again. They are entered only when they are first encountered. Subsequent connections to the desired node are made by entering an activation path to it from the root node. The activation path is entered wherever it is desired to establish a link to the previously entered node. These links must also meet certain criteria. They are:

1. The activation path must terminate at a node which has already been entered.

2. Variables which appear in the path from the root to the node from which the link is to be made (inclusive) must not appear in any subgraph of the node to which the link is to be made. Otherwise unfeasible paths would exist in the diagram.

3. Links should not be established in this way between any node and the zero or one terminal nodes. A link to one of these two nodes is established automatically when either zero or one is entered as a node variable.

4. A node should not have the same node as both its left and right offspring. This would result in a "dummy" node, which serves no useful purpose.

Once a link has been successfully established, the counts of vertices which enter nodes in the linked subgraph are adjusted to reflect the additional entering vertices.

Now that the basic characteristics of binary decision diagrams in computer memory have been presented, the movements of vertices in response to a fault can be examined.

3.2 Vertex Movement in the Presence of Faults

Two basic types of fault conditions may be placed on a binary decision diagram. These are stuck node variables and stuck edges. Akers [1] describes how to place these fault conditions on a diagram. The primary interest here is in stuck node variables, which may represent faulted logic elements in a circuit. Methods similar to those described here should apply to stuck edge faults as well.

Consider the diagram of Figure 5. If the variable C is stuck-at-0 then the vertices 001, 011, 101 and 111 would all assume a value different from the fault-free case. One could say that the vertices were shifted from the right (or 1) branch of each C node into the left (or 0) branch of that node. Thus, four vertices have had their values changed in Figure 5. Another way to view this

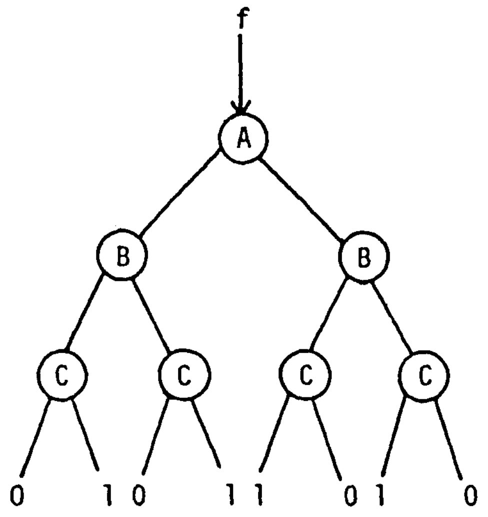

Figure 5 - Example Diagram For Fault Analysis

situation is to say that two vertices have moved from the true body of the function to the false body and two have moved from the false body to the true body. Note that four is also the number of vertices shifted by the fault C stuck-at-1.

Now consider the fault B stuck-at-0. Examination of the diagram shows that no vertices change value in the presence of this fault. The fault A stuck-at-0 causes four vertices to change value. The multiple fault A stuck-at-0, C stuck-at-0 may be analysed by considering first the subgraph isolated by the fault on A, then the subgraphs isolated by the fault on C. It is necessary to consider the faults in order from the root down so that nodes which are isolated by faults at higher levels do not introduce errors in the tally of shifted vertices. Under fault-free conditions the vertices 100 and 110 assume value 1 and vertices 101 and 111 assume value 0. With the faults on A and C in effect these vertices are changed as follows:

Vertex	Becomes:	Fault-free Value	Faulted Value
100	000	1	0
110	010	1	0
101	000	0	0
111	010	0	0

Note that all faults are removed when the fault-free performance of these vertices is studied and all faults are in effect when their performance under faulted conditions is studied. The values of two of these vertices have been affected by the faults.

At this point all vertices in the one-branch of A (the branch isolated by the fault at A) have been accounted for, including those in that branch which are affected by the fault on C. Thus the only vertices which have not been considered are those which are normally on the zero-branch of A (those not affected by the fault on A). Of these, only the vertices 001 and 011 are affected by the fault on C. The effect on these is as follows:

Vertex	Becomes:	Fault-free Value	Faulted Value
001	000	1	0
011	010	1	0

The values of two more vertices have been affected, bringing the total to four for this multiple fault.

Although this example is rather simple, it illustrates the key features of the algorithm used to trace the movement of vertices between the true and false bodies of a function in the presence of one or more faults. The general algorithm, called the Fault-Tracing Algorithm, follows.

3.3 <u>Fault-Tracing Algorithm</u>

1. Start at the root node of the diagram.

2. Continue traversing the diagram in pre-order until a faulted node is encountered or the end of the diagram is reached.

2a. If a faulted node is found, perform steps 3, 4 and 5. Then continue searching for faulted nodes along the edge specified by the fault.

2b. If a node is encountered which is set to a specified state (as opposed to having an undefined state) but it is not faulted, then continue the traversal only along the edge specified by the node state.

2c. If the end of the diagram is reached then stop.

3. Traverse in pre-order that subgraph which is inaccessible because of the fault found in step 2.

3a. Ignore all fault conditions on the subgraph.

3b. If a node is encountered which is set to a particular state but is not faulted, then continue the traversal only along the edge specified by the node state.

3c. Each time a terminal node (zero or one) is encountered, save the following information.

i. The subcube which reaches that node.

ii. The number of vertices which reach that node.

iii. The value of the terminal node (zero or one).

4. Perform step 5 for each of the subcubes saved in step 3c.

5. Traverse in pre-order that subgraph into which all subcubes saved in step 3c are forced by the faulted node. For each node encountered in the subgraph:

5a. If the node is faulted, then continue the traversal along the edge activated by the fault, allowing all vertices which remain from the subcube saved in step 3c to pass through this node.

5b. If the node has been set to some particular state but is not faulted, then perform step 5a as if the node was faulted to the specified state.

5c. If the node is not faulted or set to a particular state but the node variable appears in the subcube under consideration from step 3c, then perform step 5a as if the node was faulted to the state it has in the subcube.

5d. If the node does not meet the conditions of steps 5a, b or c and the node is not a terminal node, then continue the traversal along, first the left, then the right edges from this node, dividing the vertices remaining from the subcube saved in step 3c equally between the two edges.

5e. If the node is a terminal node (zero or one) and its value differs from the value of the terminal node which was saved in step 3c for the subcube under consideration, then add the number of vertices forced to this terminal node by the fault (those remaining after steps 5a through 5d) to the total number of vertices which have their values changed by the fault.

End of Fault-Tracing Alogrithm.

The following points can be made about the algorithm just presented.

1. In the ideal case, subgraphs which have multiple parents, that is, identical subgraphs which have been merged, need be traversed only once. The intermediate results accumulated during this traversal can (ideally) be saved and recalled upon subsequent visits to the root of the subgraph. In this way repeated visits to nodes become unnecessary. If each node is only visited once the execution time of this algorithm becomes linearly dependent upon the number of nodes in the diagram. However, the storage space required for the intermediate results may be prohibitive. This important point will be considered further when the performance of the program is evaluated.

2. The user may, if he wishes, preset one or more node variables without having them treated as faulted. This allows the diagram to be pruned so as to reduce execution time.

3. This algorithm treats each occurrence of a preset or faulted variable separately. Thus, the user could analyze individual node faults if he so desired, but the results obtained would apply only to implementations of the function which are based on the diagram. On the other hand, when all occurrences of each faulted variable are treated together the results are implementation-independent, due to the general functional representation provided by the diagram.

Further details relating to the implementation of this algorithm may be found in chapter 6.

3.4 Test Generation

The algorithm described in the previous section may be modified to generate fault-detection tests. The modifications generate a test by combining a fault-free subcube saved in step 3c with any variables, faulted or not, which: a) are specified in a particular faulted path for that subcube, and b) do not already appear in the subcube. The steps to be modified are:

5. Before beginning the traversal generate a "core" for the test which contains the states of all variables in the subcube under consideration.

5e. If a terminal node meets the condition specified in this step then add to the test "core" the states of any variables in the faulted path which are not already specified there, including faulted nodes. Do not change the states of any variables already specified in the test "core". Display the test to the user, then remove the states just added so that the test "core" is ready for the next encounter with a terminal node.

The comments which applied to the original algorithm also apply to the modified algorithm. It should be noted, however, that the additional recordkeeping involved in keeping track of paths forced by the fault and in generating tests will increase the execution time of the algorithm. This point will also be considered further when the performance of the program is evaluated.

The next chapter illustrates how the fault-tracing algorithm is used to obtain test vectors for a binary decision diagram with faults.

4. EXAMPLE FOR THE FAULT-TRACING ALGORITHM

The following example illustrates an application of the fault-tracing algorithm with test generation.

Consider the diagram of the arbitrary function $f = \overline{A}\overline{B}\overline{C}D + \overline{A}BC + A\overline{B}C + AB\overline{C}D + CE$, which appears in Figure 6. The faults C stuck-at-1 and E stuck-at-0 are assumed to exist. The edges which are forced active by these faults are highlighted.

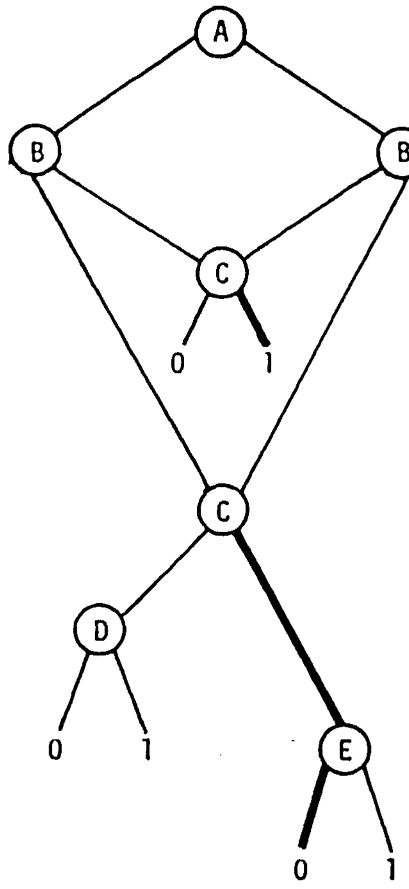

Figure 6 - Binary Decision Diagram for $f = \overline{A}\overline{B}\overline{C}D + \overline{A}BC + A\overline{B}C + AB\overline{C}D + CE$, with C stuck-at-1 and E stuck-at-0

The first step of the algorithm leads to the root rode (A) of the diagram. Step two leads to the C-node which is reached by the path $\overline{AB}$. This is the first faulted node to be encountered in a pre-order traversal. Since C is stuck-at-1, the zero-edge leaving C is traversed in step 3. This edge leads to the D-node, with four vertices entering D. Since D is not preset to a particular state, these vertices divide equally, two to each offspring. The two leaving the zero-edge of D reach the zero terminal node. Step 3c requires that the information in the first line of Table 1 be saved. Similarly, the information in the second line of the table is saved when the one-edge leaving D is traversed to reach the one terminal node. The subgraph which is the zero-offspring of C has now been completely traversed.

TABLE 1 - Information Saved In Algorithm Step 3c for C-node Reached Via $\overline{AB}$

Subcube Reaching Terminal Node	Number of Vertices Reaching Node	Value of Node
$\overline{ABCD}$	2	0
$\overline{ABC}D$	2	1

Step 4 of the algorithm specifies that step 5 is to be executed once for each subgraph saved in step 3c, so step 5 is executed twice. The first time deals with the subcube $\overline{ABCD}$. The test "core" which is constructed from this subcube is 0000-, where "-" signifies that, at this point, the value of E is "don't care" (unspecified). Traversal of the subgraph into which this subcube is forced by the fault on C leads to node E. This node is stuck at zero. Step 5a specifies that all vertices represented by the subcube $\bar{A}\bar{B}\bar{C}\bar{D}$ pass on to the zero-offspring of E. The zero-offspring of E is the zero terminal node. Since its value is equal to the nominal value of the subcube, no test can be generated. Repeating step 5 for the subcube $\bar{A}\bar{B}\bar{C}D$ (test "core" = 0001-) leads to the same terminal node. However, in this case the nominal value of the subcube is one. The two vertices contained within the subcube $\bar{A}\bar{B}\bar{C}D$ have had their values changed. According to step 5e, the variable E with value zero must be added to the test "core". Thus, the test for the change in this subcube is 00010. This is, in fact, one test for the multiple fault C stuck-at-1, E stuck-at-0.

Since the C-node under consideration has multiple parents, it is worth noting at this point that all subcubes which reach this node and nominally exit on the zero-edge exhibit some commonality in their responses to this fault. In fact, as was just demonstrated, one-quarter of all vertices which enter this faulted C-node have their values changed by the fault. The values of C, D and E in all test vectors for these vertices are 010. Thus, we know that in subsequent visits to this node, all subcubes with C = 0 and D = 1 will have their values changed by this fault. Test vectors which detect this change may be generated by prefixing the vector C = 0, D = 1, E = 0 with the values of A and B from the subcubes.

In general, a table may be made describing the effects of a fault at a node upon vertices which nominally enter the subgraph isolated by the fault. This table is made by considering only the subgraph whose root is the faulted node. The table contains:

1. A list of all subcubes within the subgraph whose values are changed by the fault at its root, as determined by steps 3, 4 and 5 of the fault-tracing algorithm.

2. The nominal values of each of these subcubes, from which their faulted values are easily derived.

3. The ratio of the number of vertices in each of these subcubes to the total number of vertices entering the subgraph.

4. A test sub-vector for each subcube which detects the change in the value of the subcube.

Note that this could conceivably be a large amount of information. It must be saved for all faulted nodes which either have multiple parents or whose ancestors have multiple parents. Such a table for the subgraph of C which has just been analyzed would look like Table 2. This table will be used later.

TABLE 2- Partial Test Information for C-Node Reached Via $\overline{AB}$

Changed Subcube	Nominal Value	# Changed / Total Vertices	Test Sub-vector
$\overline{C}D$	1	1/4	--010

Now that steps 3, 4 and 5 of the algorithm have been completed, the search for faulted nodes is continued with the one-offspring of the C-node, as specified by step 2a of the algorithm. The next node encountered is the E-node, which is faulted. Execution of step 3 results in Table 3.

TABLE 3 - Information Saved in Algorithm Step 3c for E-node

Subcube Reaching Terminal Node	Number of Vertices Reaching Node	Value of Node
$\overline{A}\overline{B}CE$	2	1

Step 4 specifies that step 5 is to be executed once. Execution of step 5 with subcube $\overline{A}\overline{B}CE$ (test "core" = 001-1) leads immediately to step 5e. At this point it is determined that the two vertices contained in subcube $\overline{A}\overline{B}CE$ have their values changed by the fault. The test for this change is 001-1. A table similar to Table 2 is generated for this faulted node because one of its ancestors has multiple parents. The new table is Table 4.

TABLE 4 - Partial Test Information for E-node

Changed Subcube	Nominal Value	# Changed / Total Vertices	Test Sub-vector
E	1	1/2	----1

This completes the analysis of the zero-offspring of the B-node reached via $\overline{A}$.

Step 2 now leads to the one-offspring of the B-node reached via $\bar{A}$. Analysis of the subgraph whose root is this C-node results in the detection of four changed vertices, all in subcube $\bar{A}B\bar{C}$. The test for these changes is 010--. The change in value is from zero to one. Since this C-node also has multiple parents, intermediate results are saved. These appear in Table 5.

TABLE 5 - Partial Test Information for C-node Reached Via $\bar{A}B$

Changed Subcube	Nominal Value	# Changed/Total Vertices	Test Sub-vector
$\bar{C}$	0	1/2	--0--

Note that the test sub-vector specifies "don't care" for D and E. This is because the ordering of node variables, which was a necessary part of constructing the graph, implies that D- and E-nodes can not precede this C-node. Inspection shows that they do not follow it. Thus, they are "don't cares".

The next faulted node reached by step 2 is the C-node which is reached via $A\bar{B}$. At this point the partial results which were saved during the first visit to this node may be used (see Table 5). With regard to the subcube $A\bar{B}$, which enters this node, it can be seen that the subcube $A\bar{B}\bar{C}$ has its value changed from zero to one. One-half of the eight vertices which enter C from subcube $A\bar{B}$, or a total of four vertices, have their values changed by the fault. The test for these changes is 100--

Step 2a now leads to the C-node which is reached via AB. Combining the entering subcube AB with the information in Table 2, it is evident that the subcube $AB\overline{C}D$ has its value changed from one to zero. One-quarter of the eight vertices which enter C from subcube AB, or a total of two vertices, have their values changed by the fault. The test for these changes is 11010.

The final faulted node which is encountered is the E-node reached via ABC. Once again, combining the entering subcube ABC with the information previously saved at this node (Table 4) leads to the detection of a changed subcube, ABCE, whose value is changed from one to zero and which contains one-half of four, or two changed vertices. The test for these changes is 111-1.

A summary of the results obtained from the analysis of the diagram of Figure 6 is given in Table 6.

TABLE 6 - Results of Analysis of Figure 6

Changed Subcube	Nominal Value	Faulted Value	Test Vector	Number of Changed Vertices
$\overline{ABC}D$	1	0	00010	2
$\overline{AB}CE$	1	0	001-1	2
$\overline{A}B\overline{C}$	0	1	010--	4
$A\overline{BC}$	0	1	100--	4
$AB\overline{C}D$	1	0	11010	2
ABCE	1	0	111-1	2

The ability of the computer to efficiently analyze binary decision diagrams in the manner just described is influenced greatly by the way in which the diagrams are represented within the computer. The next chapter discusses the data structures used for this purpose.

5. DATA STRUCTURES

The choice of data structures used to represent a binary decision diagram in memory plays a very important role in determining the effectiveness and efficiency of the manipulation program. Properly saving key items of information at each node avoids unnecessary re-derivation of that data each time the node is visited during a series of operations. On the other hand, too much data or unnecessary data saved at each node can lead to a great deal of wasted storage, as there could conceivably be hundreds or even thousands of nodes in a diagram. A similar situation exists with regard to data which applies to the diagram as a whole. Key items of information of this type can be saved globally to avoid unnecessary re-derivation or repetition with each node. However, too much global data can lead to confusion during programming. A balance must be reached among all of these factors. With this in mind a review will be made of the data needed by the computer to perform the desired processing of a diagram.

5.1 Node Representation

The information associated with a node consists of several items. The most basic items, as described by Akers [2], are the node variable, a pointer to the left offspring of the node and a pointer to the right offspring of the node. Some additional information is needed to efficiently implement the algorithms described in chapter 3. The number of vertices which enter a node should be kept with each node. This number is needed by the algorithm to determine the number of vertices affected by fault conditions. Also, the state (value) of the node variable must be stored with the node to determine which subgraphs are to be ignored by the algorithm and which are affected by faults. The state of a node may be zero, one or unspecified. Finally, a flag must be maintained to indicate whether or not the state of the node is to be treated as a fault. Each of these data items should be kept with the node to which they apply so that they are available to the program when they are needed without the need for re-computing them.

The existence of an overall data descriptor (the node) with several detailed data fields indicates that a record structure is an appropriate way to represent a node. An array structure might be used but, since the elements of the array would be of different types (some alphanumeric, some integer, some boolean), an array structure would be difficult to use. Programming languages which allow record structures generally allow the fields of the record to be of different types. Thus a record structure has been chosen. A sample node record is illustrated in Figure 7.

Node					
name	value	Number of changed vertices	Pointer to left offspring	Pointer to right offspring	fault flag

Figure 7 - Sample Node Record

5.2 General Diagram Structure

The node records derived in the previous section must be interconnected in some way to represent a binary decision diagram. The need for pointers from a node to its two offspring implies the use of a linked structure. Two possible implementations of a linked structure are an array, where the pointers to offspring are the indices of other entries in the array, and a directly linked structure, where the pointers to offspring are the actual memory addresses of the offspring. These two implementations would represent a binary decision diagram respectively as an array of records and a directly linked list of records. Either structure is adequate. However, programming languages generally require array dimensions to be declared in advance of any use of the array, whereas those which can directly manipulate memory addresses generally have a facility for dynamically requesting and releasing storage on an item-by-item basis. Node-by-node allocation of memory space for the diagram eliminates the need to estimate in advance the number of nodes in a diagram. In addition, a directly linked list has the advantage of working with actual memory addresses. An array index, on the other hand, must undergo some arithmetic processing to be converted into an actual memory address before it can be used. The time required by this processing is insignificant in an individual case, but may have a significant effect upon execution time when it is accumulated over thousands of node references. For these two reasons a directly linked list of records has been chosen to represent a binary decision diagram.

5.3 Global Data

Most of the information needed by the algorithms is stored within the diagram. However, there are certain items of information which must be available globally. For example, there may be items of information which apply to the diagram as a whole and not to any particular node. There may also be items which relate chiefly to the programmed implementation of the algorithms and not to any particular diagram. A third group of global items are those which are contained within the diagram but are also stored globally for convenient access by the program. Items in the first category include the number of variables in the diagram, a title or identifier for the diagram and a flag which indicates whether or not the diagram has been saved in a permanent storage file. The second category includes such items as the starting and completion times of certain operations on the diagram, the name of and pointer into the file onto which the output from the current operation is to be written, and a flag to indicate when output to the user's terminal is to be suppressed. Items in the third category are the addresses of the zero and one terminal nodes, the address of the root node, a list of the variables in the diagram and the number of vertices in the function which the diagram represents. In addition to these three groups are data items which apply only to particular procedures or subroutines within the program. Items of this nature are stored locally within the procedure or subroutine to which they apply. In this way the memory space used by these items may be returned to the operating system when the individual procedures or subroutines are completed.

The description of data items presented above was not developed fully before the programming task was begun. Rather, the basic needs were outlined in light of the tasks to be performed by the program. Revisions and more detailed descriptions were then made as the details of the program were developed. With this overall description of the data structures in mind the programmed procedures which manipulate them may now be explored in detail.

6. THE PROGRAM PACKAGE FOR DIAGRAM MANIPULATION

The binary decision diagram analysis program is actually a program package consisting of a set of independent routines under the control of a supervisory program. The routines all operate upon a common database, which is the diagram. This chapter discusses some implementation considerations relating to the algorithms described in Chapter 3.

6.1 Implementation Considerations for the Algorithms

The algorithms presented in Chapter 3 center around one or more traversals of a binary decision diagram or some subgraph of one. Techniques for traversing graphs are discussed extensively in the programming literature. A traversal of a diagram generally involves a visit to every node in the diagram. Because of this, program speed and efficiency depend greatly upon the traversal technique which is used. Discussions of several traversal techniques and their implementations appear in [3] and [4]. The technique of primary interest here is preorder traversal, as described in Chapter 3. This technique is required by the algorithms in order to keep proper track of the vertices which are affected by faults. (If preorder traversal were not used it would be necessary to either explicitly store with each node a list of all variables between that node and the root or to backtrack to the root every time such a list is needed.) There are two basic ways in which preorder traversal may be implemented: iteration and recursion [4].

Iteration uses explicit pointers and indices to keep track of the nodes which have been visited. These pointers and indices must be maintained manually by the programmer, that is, explicit source language statements must be included within the traversal program to perform this task. An iterative program traverses the diagram by repeated execution of one or more loops which first process the current node, then alter the pointers and indices so as to advance to the next node to be processed. Recursion takes advantage of two characteristics of binary decision diagrams: (1) Every subgraph of a binary decision diagram is itself another binary decision diagram, and (2) all diagrams are acyclic graphs [1]. The recursive nature of the definition of a binary decision diagram makes implementation of a traversal algorithm very simple and straightforward if the programming language used allows recursion (that is, allows subroutines to call themselves). In this case only one basic subroutine need be written. This subroutine would process the current node, then call itself once to process its left offspring and once to process its right offspring. Of course, tests must be included to prevent it from attempting to process the offspring of a terminal node. A recursive language processor (compiler or interpreter) generates a run-time stack [5] which is used to automatically maintain pointers and indices to keep track of the nodes which have been visited, freeing the user from this task. Unfortunately, the overhead associated with this automatic maintenance can often result in greater execution time than for an equivalent iterative algorithm, where the user has tight control over the pointer maintenance procedures. However, the simplicity of recursive code in an application such as this can greatly reduce program development and debugging time. This is one reason that recursive approach is taken in the implementation of these algorithms. The second reason for choosing recursion is that it allows for concentration on the desired processing of node information, since the burden of maintaining the required pointers into the graph is assumed by the language processor.

6.2 Choice of a Programming Language

The choice of a programming language in a given situation depends largely upon the types of data structures required and the algorithms to be implemented. In order to be suitable for this particular application a programming language should possess these characteristics:

1. A variety of high level data structures. These include records, arrays and directly-linked lists.

2. Dynamic memory allocation so that the memory space used may increase and decrease to suit the sizes of different diagrams, thereby minimizing memory usage.

3. A variety of flexible control structures, including recursion, IF-THEN-ELSE and CASE (conditional branching with more than

two alternatives). These control structures greatly simplify the implementation of the algorithms as compared to languages which lack these structures.

Several languages which possess these characteristics are available. The one most readily available for interactive use at Lehigh University is PASCAL, so this the language of choice.

6.3 Outline of the Program Package

As mentioned previously, the program package for the manipulation of binary decision diagrams consists of several independent subroutines, call procedures in PASCAL, which are under the control of a supervisory program. The supervisor accepts an instruction from the user, invokes the appropriate procedures and monitors the CPU time used by each operation and by the entire program. Error trapping, error diagnostics and instruction lists are available to the user both at the supervisor level and within the individual procedures. These are important to protect the user from accidentally erasing results or diagrams, as well as for instructional and documentation purposes.

CPU utilization is closely monitored throughout the execution of the program so that program performance may be accurately measured. The results of these measurements for several test cases are presented in the next chapter.

7. PERFORMANCE EVALUATION

The statement was made in Chapter 3 that the execution time of the fault-tracing algorithm is ideally a linear function of the number of nodes in the diagram. This hypothesis will now be discussed in more detail. A discussion of memory usage and some peculiarities of this particular implementation will also be presented. This will be followed by the results of some sample runs.

7.1 CPU Time Versus Memory Usage

The ideal implementation which has been emphasized so far is one which minimizes CPU time. However, there are in fact two ideal implementations. The second is one which minimizes memory usage. Reduction of CPU time requires the storage of a significant amount of intermediate data. If increased usage of CPU time can be tolerated, then memory usage can be reduced. The minimum amount of memory space required by the algorithm is the space occupied by the diagram itself plus the maximum storage space required by one execution of step 3 of the fault-tracing algorithm. (This excludes the space occupied by the program itself.) Since no intermediate data is preserved, no CPU time is saved by reducing the diagram. That is, identical subgraphs which have been merged will have to be traversed once for each edge entering the subgraph. Merging of identical subgraphs in this case serves only to reduce the memory space occupied by the diagram.

The main goal of this research is to study the cost, in terms of memory and CPU time, of performing the fault-tracing algorithm on a computer. Memory usage can be calculated in a fairly straightforward manner once data structures have been established. Calculation of CPU time is not so straightforward, unless one has established some benchmark, such as the amount of time (in CPU seconds or fractions thereof) required to process some number of nodes. The major emphasis here has been to obtain such a benchmark. Thus, the program described in chapter 6 is implemented so as to minimize memory usage. This allows a wider range of graph sizes to be analyzed than would be possible if large amounts of intermediate data were saved. It also provides a worst-case upper bound on the CPU time required by any implementation of the algorithm.

7.2 Implementation Details

The program has been implemented on a DECsystem-20 computer using a local modification of the PASCAL-20 compiler. Standard PASCAL features have been used wherever possible. On this machine the memory image of the program fills approximately 14 800 36-bit words of main memory. An individual node record uses five words of memory, plus one additional word for each multiple of five letters allowed in the name of a variable. The maximum number of variables allowed in a diagram is a parameter in the source code which may be varied by the user, as is the number of letters which may be allowed in the name of a variable. The user may change these values simply by changing two statements and recompiling the program, thus allowing the sizes of certain arrays to be minimized for a particular application. The memory usage specified above for the program is for single-character variable names and up to 34 variables in a diagram. The memory space taken by the diagram is determined dynamically during program execution by requesting memory for each node record as it is created, through the use of the PASCAL procedure NEW. When a diagram is erased, the memory space which it occupied is released by the program. For example, a full ten-variable binary tree with single-character variable names requires six 36-bit words per node or 6150 words (including six words each for the zero and one terminal nodes).

The information saved during step 3c of the fault-tracing algorithm is stored as a linked list of records. A new record is created each time a terminal node is encountered. The memory used for each record is:

a. one word each for the value of the terminal node, the number of vertices which reach the terminal node, the number of these vertices which are changed in value and the address of the next record, plus

b. space for the subcube which reaches this terminal node.

The subcube requires (one word for each multiple of five letters allowed in the name of a variable + one word for the state of the variable) * the maximum number of variables allowed in a diagram. Thus, for up to 34 variables with single-character names, each record requires 4+((1+1)*34) = 72 words. With a maximum of 15 variables allowed in a diagram, 4+((1+1)*15) = 34 words are required for each record. It should be evident from these figures that a substantial amount of memory space is required to store intermediate results.

7.3 Test Cases

The wide variety of configurations which are possible among binary decision diagrams, even those which represent the same function, make the selection of test cases an extremely difficult task. Since the goal here is to develop a worst-case estimate of the CPU time required by the fault-tracing algorithm (along with the corresponding memory usage), full binary trees are used as test diagrams. There are several reasons why this choice is appropriate. First, Akers has shown that no binary decision diagram can contain as many nodes for a given number of variables as does a full binary tree [1]. In fact, he has proven that the number of nodes in a diagram with v variables is of the order $(\frac{2^v}{v})$. Second, the number of nodes in a full binary tree with v variables is easy to determine (2^v-1, plus terminal nodes). The third reason comes from a study of worst case fault conditions on a full binary tree.

Several test runs were made using a full ten-variable binary tree, with variables arranged in lexicographical order from the root to the leaves. These test runs involved maximizing the number of changed vertices for stuck-at faults at each of the variables in turn (by varying the pattern of one and zero terminal nodes), then simulating stuck-at faults first on that variable alone, then on all variables. The results of these tests indicate that the worst case stuck-at fault condition (most CPU time required) on a binary tree is all variables stuck, where the terminal nodes are alternate zeros and ones from left to right across diagram (although the variation among different patterns of zeros and ones is small). This arrangement can be modeled very simple and in a very small amount of memory by "folding" the tree. This merging of identical subgraphs results in a diagram similar to Figure 8. Since the program does not take advantage of merged subgraphs, the use of a reduced tree does

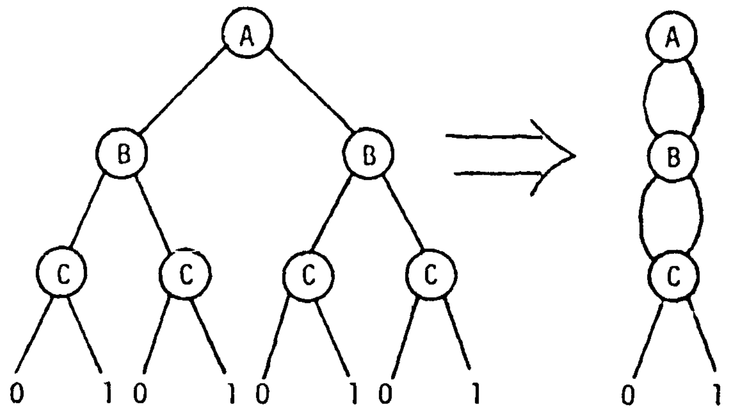

Figure 8 - "Folding" of a Three-Variable Binary Tree

not affect the execution time of the program. It does allow larger trees to be analyzed, since more memory is free for use in step 3c of the algorithm.

It should also be noted that the maximum possible amount of storage required for step 3c is 2^{v-1} records, where v is the number of variables in the diagram. This occurs when analyzing a fault at the root node.

7.4 Test Results

A total of six tests were run on each of twelve diagrams. The twelve diagrams were folded binary trees (as previously described) with from one to twelve variables. The six tests were as follows:

1. List all faults in the diagram, generate fault-detecting tests for the faults and count the number of changed vertices.

   Case A: No faults.

   Case B: All variables stuck-at-zero

2. Generate fault-detecting tests for all faults and count the number of changed vertices. (Do not list the faults.)

   Case A: No faults.

   Case B: All variables stuck-at-zero.

3. Count the number of changed vertices only. (Do not list the faults or generate any tests.)

   Case A: No faults.

   Case B: All variables stuck-at-zero.

Tables 7 through 10 illustrate the listings produced by tests one and two for a five-variable binary tree. Test three produced no output other than to print the number of changed vertices at the user's terminal.

The results of these tests are given in Tables 11, 12 and 13 and in Figures 9 and 10. Several important observations can be made about these results. First, listing of the faults which are present on the diagram is an option which the user may select prior to executing the algorithm. It requires an extra traversal of the diagram and thus increases the amount of CPU time which is used. Second, the generation of fault-detecting tests requires a considerable amount of overhead. This arises from the extra manipulations which are required to generate the tests. Third, when the execution times of the tests exceed approximately 0.5 CPU second, $\log_2$(CPU time) appears to increase linearly with the number of nodes in the diagram. This implies that the CPU time is proportional to $2^{(\text{number of nodes})}$. It is likely that below 0.5 CPU second of execution time the overhead associated with the operating system becomes significant enough to affect the test results. Fourth, while attempting to perform the fault-tracing algorithm on a thirteen-variable binary tree with all nodes faulted the program ran out of memory space. This occurred after generating over 6000 of the records required by step 3c of the algorithm. At 34 words per record (see section 7.2) this represents over 204,000 (decimal) words of memory, which is several orders of magnitude greater than that occupied by the diagram.

A number of conclusions may be drawn from these observations. They are presented in the next chapter.

TABLE 7 - Sample Output From Test 1a

```
Fault simulation on  Five variable binary tree    Today is 31-Jul-80


Faulted variables:  None.

Individual faulted nodes:  None.

Preset variables:  None.

Individual preset nodes:  None.


Tests for the specified conditions:

        Nominal  Faulted
         Value    Value    A, B, C, D, E.
        -------  -------   --------------


TOTAL of 0 vertices change value.

End of simulation.
```

TABLE 8 - Sample Output From Test 1b

```
Fault simulation on  Five variable binary tree    Totay is 31-Jul-80


Faulted variables:  A stuck-at-0, B stuck-at-0, C stuck-at-0
                    D stuck-at-0, E stuck-at-0

Individual faulted nodes:  None.

Preset variables:  None.

Individual preset nodes:  None.


Tests for the specified conditions:

        Nominal   Faulted
         Value     Value    A, B, C, D, E.
        -------   -------   --------------


           1  ===>  0        1  0  0  0  1
           1  ===>  0        1  0  0  1  1
           1  ===>  0        1  0  1  0  1
           1  ===>  0        1  0  1  1  1
           1  ===>  0        1  1  0  0  1
           1  ===>  0        1  1  0  1  1
           1  ===>  0        1  1  1  0  1
           1  ===>  0        1  1  1  1  1
           1  ===>  0        0  1  0  0  1
           1  ===>  0        0  1  0  1  1
           1  ===>  0        0  1  1  0  1
           1  ===>  0        0  1  1  1  1
           1  ===>  0        0  0  1  0  1
           1  ===>  0        0  0  1  1  1
           1  ===>  0        0  0  0  1  1
           1  ===>  0        0  0  0  0  1


Total of 16 vertices change value.

End of simulation.
```

TABLE 9 - Sample Output From Test 2a

```
Fault simulation on  Five variable binary tree    Today is 1-Aug-80

Tests for the specified conditions:

        Nominal  Faulted
         Value    Value    A, B, C, D, E.
        -------  -------   --------------


TOTAL of 0 vertices change value.

End of simulation.
```

TABLE 10 - Sample Output From Test 2b

```
Fault simulation on  Five variable binary tree    Today is 1-Aug-80

Tests for the specified conditions:

        Nominal  Faulted
         Value    Value    A, B, C, D, E.
        -------  -------   --------------

           1   ===>  0      1  0  0  0  1
           1   ===>  0      1  0  0  1  1
           1   ===>  0      1  0  1  0  1
           1   ===>  0      1  0  1  1  1
           1   ===>  0      1  1  0  0  1
           1   ===>  0      1  1  0  1  1
           1   ===>  0      1  1  1  0  1
           1   ===>  0      1  1  1  1  1
           1   ===>  0      0  1  0  0  1
           1   ===>  0      0  1  0  1  1
           1   ===>  0      0  1  1  0  1
           1   ===>  0      0  1  1  1  1
           1   ===>  0      0  0  1  0  1
           1   ===>  0      0  0  1  1  1
           1   ===>  0      0  0  0  1  1
           1   ===>  0      0  0  0  0  1

TOTAL of 16 vertices change value.

End of simulation.
```

TABLE 11 - Results For Test 1:
List all faults in diagram, generate fault-detecting tests and count changed vertices.

Number of:		CPU time (sec.) used with:	
Variables	Nodes	No Faults	All nodes faulted
1	3	0.13	0.12
2	5	0.13	0.13
3	9	0.13	0.15
4	17	0.13	0.19
5	33	0.15	0.26
6	65	0.19	0.45
7	129	0.27	0.90
8	257	0.43	1.89
9	513	0.73	4.06
10	1025	1.36	8.80
11	2049	2.65	19.60
12	4097	5.23	27.24

TABLE 12 - <u>Results for Test 2:</u>

Generate fault-detecting tests
and count changed vertices.
(Do not list faults.)

Number of:		CPU time (sec.) used with:	
Variables	Nodes	No Faults	All nodes faulted
1	3	0.12	0.13
2	5	0.12	0.12
3	9	0.13	0.15
4	17	0.13	0.17
5	33	0.14	0.23
6	65	0.16	0.36
7	129	0.20	0.68
8	257	0.30	1.38
9	513	0.47	2.99
10	1025	0.81	6.59
11	2049	1.53	14.46
12	4097	2.95	33.50

TABLE 13 - Results for Test 3:

Count changed vertices only. (Do not list faults or generage fault-detecting tests.)

Number of:		CPU time (sec.) used with:	
Variables	Nodes	No Faults	All nodes faulted
1	3	0.07	0.06
2	5	0.06	0.06
3	9	0.06	0.06
4	17	0.06	0.07
5	33	0.06	0.08
6	65	0.06	0.12
7	129	0.06	0.17
8	257	0.07	0.30
9	513	0.08	0.54
10	1025	0.10	1.09
11	2049	0.15	2.12
12	4097	0.23	4.35

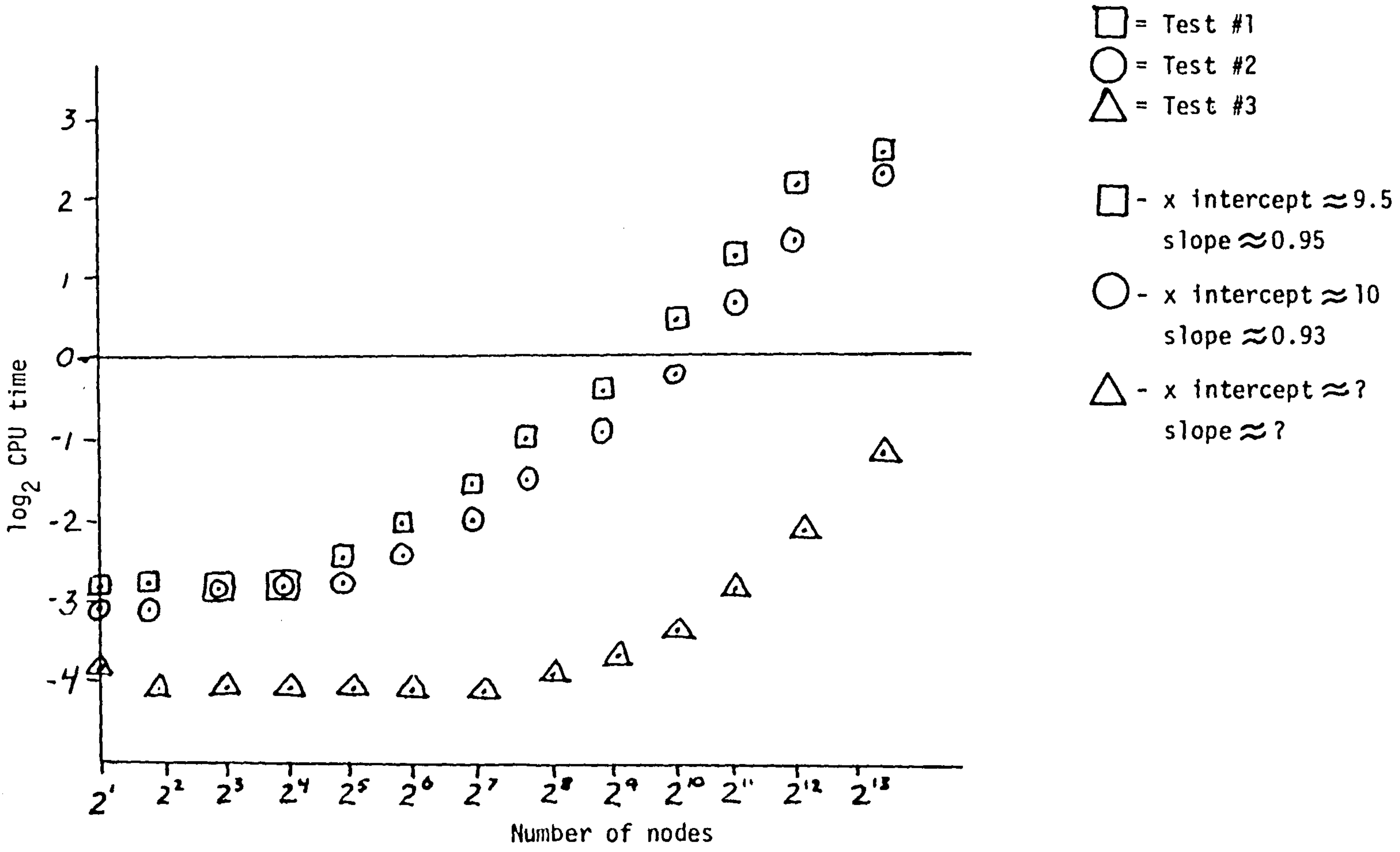

Figure 9 - Results for Tests 1a, 2a and 3a

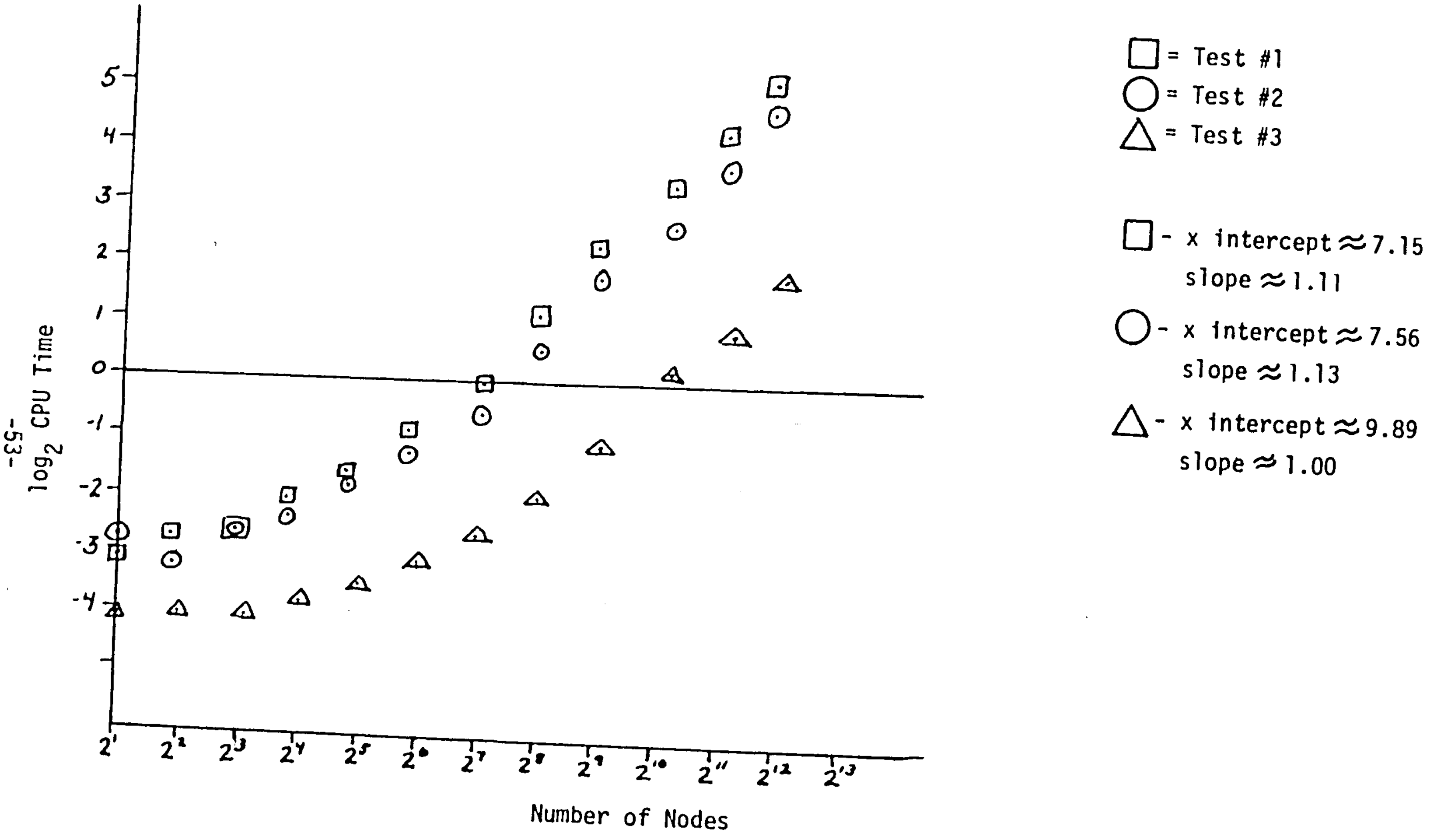

Figure 10 - Results for Tests 1b, 2b and 3b

8. CONCLUSIONS

The test results presented in chapter 7 cause the program presented here to appear as an inefficient means by which to analyze the performance of a function under fault conditions. However, before casting it aside it is appropriate to consider a few points. First of all, absolute worst-case analyses of the performances of both the program and the algorithm have been presented. The case where all variables in a function are simultaneously faulted is rarely tested in practice. In fact, rarely are more than single or double fault conditions tested. The performance of the program is substantially better for these cases. In addition to this, it is reasonable to expect that only a small percentage of the boolean functions encountered in practice will have diagrams which approach a full binary tree in the number of nodes which must be visited. (Recall Akers' limit of $O(\frac{2^v}{v})$ nodes for v variables in a function.)

The performance of the program (this particular implementation of the fault-tracing algorithm) for all cases could be improved directly in a number of ways. Examination of the test results for the case where no faults are present shows that a significant amount of time is spent just traversing the diagram to locate faults (see Figure 9). Faster methods of locating and listing faults would measurably improve program performance. Two possibilities are to maintain a table of addresses of faulted nodes and to use threaded diagrams [3]. The use of threaded diagrams would also improve the execution times of steps 3 and 5 of the algorithm. The amount of storage used by step 3c could be reduced by maintaining explicit links to parent nodes within node records, allowing paths to be traced backwards rather than stored, although this would increase the memory space required by the diagram. Flags could then be set to indicate which of multiple parents are to be considered in a particular instance. Perhaps an efficient storage scheme could also be developed which would store intermediate data, so as to reduce the number of duplicate traversals of identical subgraphs. Any one of these modifications could reduce the overall CPU time used by the program and possibly result in more efficient memory utilization.

The fault-tracing algorithm as presented in chapter 3 still appears to be an effective way of analyzing the performance of a boolean function in the presence of fault conditions. Its most important feature is that it provides a functional analysis rather than an implementation-dependent analysis of the effects of stuck-variable faults upon a function. Even the implementation presented here required reasonable amounts of CPU (and real) time for the worst-case stuck-at fault analyses of functions of up to twelve variables.

REFERENCES

[1] Akers, S. B., "On the Specification and Analysis of Large Digital Functions", Proceedings of the Seventh International Symposium on Fault Tolerant Computing, pp. 88-93, June 1977.

[2] Akers, S. B., "Binary Decision Diagrams", IEEE Transactions on Computers, Vol. C-27, pp. 509-516, June 1978.

[3] Knuth, D. E., Fundamental Algorithms, The Art of Computer Programming, Vol. 1. Reading, MA: Addison-Wesley, 1969.

[4] Goodman, S. E. and Hedetniemi, Introduction to the Design and Analysis of Algorithms. New York, NY: McGraw-Hill, 1977.

[5] Gries, D., Compiler Construction for Digital Computers. New York, NY: John Wiley & Sons, Inc., 1971.

VITA

The author was born on June 19, 1956 in San Jose, California, to Mr. and Mrs. Gerald McHugh. In 1960 the author's family moved to Concord, California, then to Bethlehem, Pennsylvania in 1965. They have resided in Bethlehem since that time. Undergraduate studies leading to the Bachelor of Science degree in Electrical Engineering were undertaken at Lehigh University from the fall of 1974 through the spring of 1978. Graduate studies leading to the Master of Science degree in the same field were undertaken at Lehigh in the fall of 1978. In August of 1979 the author married the former Mary Melber of Allentown, Pennsylvania, who is also a graduate of Lehigh. The author is employed by NCR Corporation in Clemson, South Carolina.