A GRAPH-THEORETIC MATRIX-BASED APPROACH FOR A MEASURE OF PROGRAM TESTING AND AN ADAPTIVE TESTING STRATEGY

By

SHANKAR NARAYANASWAMY

Bachelor of Engineering

Bangalore University

Bangalore, India

1984

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the degree of
MASTER OF SCIENCE
July, 1991

A GRAPH-THEORETIC MATRIX-BASED APPROACH FOR A MEASURE OF PROGRAM TESTING AND AN ADAPTIVE TESTING STRATEGY

Thesis Approved:

[illegible]
Thesis Adviser

J Chandler

[illegible]

[illegible]
Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation and thanks to Dr. Mansur H. Samadzadeh for his encouragement and advice throughout my thesis research. I would also like to place on record my thanks to Drs. John P. Chandler and Huizhu Lu for serving on my graduate committee.

I would also like to thank my brothers Selvam and Shekar, my sister Sumathi, and most of all my parents for their unflinching, unquestioned, and constant support in everything that I have done so far. I wish to thank Sharmila for her love and understanding, and my good friend Ravi Mandadi for his invaluable assistance.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Software quality should be a primary concern in any software development effort. The traditional methods of assessing the quality of software are program testing and software evaluation [DEMI87]. Program testing is an important means of achieving an improvement in software quality and reliability [LIN89, ADRI82, BEIZ83, MYER79].

Testing could be visualized as a process whereby a program is executed with the intention of finding the errors contained therein [MYER79]. Testing could also be perceived as "the controlled analysis and/or execution of a program expressed in some language, done to verify the pre-determined (pre-specified) presence of some desired program property" [MILL79]. The general goal of program testing is "to affirm the quality of a software system by systematically exercising the software under carefully controlled circumstances" [MILL81]. In this context, it is interesting to note Dijkstra's comment regarding testing as reported by Miller [MILL79], "program testing can only serve to identify program bugs, never to eliminate them". If it were possible to guarantee the correctness of programs, this would serve as the ultimate goal of program testing [NTAF84].

Adrion, Branstad, and Cherniavsky [ADRI82], cite five essential components of a program test:

1. The program in executable form;
2. A description of the expected behavior;
3. A way of observing the program behavior;
4. A description of the functional domain; and
5. A method of determining whether the observed behavior conforms with the expected behavior.

Of the five essential components of a program test cited in [ADRI82], the second component is the most difficult one to obtain. Ideally, an oracle (a source which for any given input description can provide a complete description of the corresponding output behavior) is required in order to obtain this component [ANDR86].

Miller [MILL81] claims that the primary motivating force for program testing is the considerable cost involved in the process of testing. The veracity of this claim is evident from the abundance of concurrence from other published sources, a few of which are [BEIZ84, ADRI82, MCCA76, MILL84, and ONOM87].

There seems to be a need for some means of quantifying program testing. Such a measure is usually called a metric and is generally defined as any number that is used to measure an interesting property of something [BEIZ83].

In the context of this thesis, the term "metric" applies specifically to a measure used for quantifying the complexity of programs. The development of such a complexity measure or metric would serve to fulfil the need for some objective measures of various aspects of software, such as software quality [PAIG80].

This thesis involves the development of an algorithm used to compute such a complexity metric and another that serves as an adaptive testing strategy. Both of these algorithms rely upon a graph-theoretic, matrix-based approach.

CHAPTER II

LITERATURE REVIEW

2.1 Graph Theory Preliminaries

This section introduces the graph theory preliminaries used throughout this thesis. It is essentially a compilation of all the graph-theoretic terminology used in this document.

<u>DIGRAPH (DIRECTED GRAPH)</u>: A digraph is an ordered pair (V,E) where V is a finite set of vertices, and E is a relation on V. The elements of E are called the edges of the digraph. For every pair of vertices u,v   V, the set of edges E will contain at most one edge (u,v) from u to v, and at most one edge (v,u) from v to u. If (u,v)   E, we say that u precedes v or is an antecedent of v [SKVA86].

<u>STRONG COMPONENT</u>: The set of vertices in a digraph D can be partitioned into equivalence classes, and by giving each equivalence class all the nodes connected to one another, the connected subgraphs of a graph, called its components, can be constructed [SKVA86].

If u is a point in a digraph D then the set of vertices that belong to the equivalence class of u is called the component (or, alternatively, a strong component) of u, which is symbolized by C(u). Since components are equivalence classes, the components defined by two points are either the same or have no points in common [ROBI80].

<u>STRONGLY CONNECTED GRAPH</u>: A digraph with one strong component is called strongly connected.

<u>LINEAR DEPENDENCE</u>: A set of vectors $X_1, X_2, \ldots, X_r$ (over some field F) is said to be linearly independent if for scalars $c_1, c_2, \ldots, c_r$ in F, the expression

$$c_1X_1 + c_2X_2 + \ldots + c_rX_r = 0$$

holds only if $c_1 = c_2 = \ldots = c_r = 0$. Otherwise, the set of vectors is said to be linearly dependent [DEO74].

<u>BASIS VECTOR</u>: If every vector in a vector space W can be expressed as a linear combination of a given set of vectors, this set is said to span the vector space W. The dimension of the vector space W is the minimal number of linearly independent vectors required to span W. Any set of k linearly independent vectors that spans W, a k-dimensional vector space, is called a basis for the vector space W [DEO74].

<u>ADJACENCY MATRIX</u>: Two nodes $v_1, v_2 \in V$ in the digraph D = (V,E) are adjacent if there exists either of the two edges: (v_1, v_2) or $(v_2, v_1) \in E$. Given a digraph D, its adjacency matrix A(D), is defined by

$$A(D) = [a_{ij}]; \qquad i, j = 1, 2, \ldots, n,$$

$$\text{where } a_{ij} = \begin{cases} 1, & \text{if } (v_i, v_j) \in E \\ 0, & \text{otherwise} \end{cases}$$

INCIDENCE MATRIX: The incidence matrix [DEO74] of a digraph D, with n nodes, e edges and no self-loops is an n by e matrix $I[D] = [a_{ij}]$, whose rows correspond to the nodes and its columns correspond to the edges, such that

$$a_{ij} = \begin{cases} 1, & \text{if the jth edge is incident out of the ith node} \\ -1, & \text{if the jth edge is incident into the ith node} \\ 0, & \text{if the jth edge is not incident on the ith node} \end{cases}$$

PATH MATRIX: A path matrix [DEO74], is defined for a specific pair of nodes in a graph, say x and y, and is written as P(x,y). The rows in P(x,y) correspond to the different paths between nodes x and y and the columns correspond to the edges in a digraph D. That is, the path matrix for the nodes x and y is $P(x,y) = [p_{ij}]$, where

$$p_{ij} = \begin{cases} 1, & \text{if jth edge lies in its path} \\ 0, & \text{otherwise} \end{cases}$$

OPEN CHAIN: This term refers to the set of 1's in a specific row of the adjacency matrix linked together as specified in the complexity measure algorithm (see Chapter III).

LINK OF A CHAIN: This term is used to represent the pairs of 1's grouped together as shown in the adjacency matrices of the example digraphs for the complexity measure algorithm (see Chapter III, Section 3.4).

C(G): is the proposed measure of complexity as derived from the adjacency matrix according to the proposed algorithm (see Chapter III).

<u>STATEMENT COVERAGE</u>: Execution of all statements in the graph of a program, as a testing strategy [PRAT87].
<u>NODE COVERAGE</u>: Encountering all decision node entry points in the graph of a program, as a testing strategy [PRAT87].
<u>PATH COVERAGE</u>: Traversing all paths of the graph [PRAT87].
<u>BRANCH COVERAGE</u>: Encountering all exit branches of each decision node in the graph of a program, as a testing strategy. The branch coverage criterion has come to be regarded as a minimal standard of achievement in structured testing and is widely recognized as the basic measure of testing thoroughness [PRAT87].
<u>BRANCH TESTING</u>: A testing method satisfying the coverage criteria that requires that for each decision point each possible branch be executed at least once [ADRI82].
<u>MUTATION TESTING</u>: Mutation testing involves the application of a set of mutation transformations to a user's program. Each transformation results in a mutant. A set of test data is considered complete if, for each mutant, there is at least one test for which the user's program and the mutant generate different output [HOWD81b].

2.2 Testing Strategies and Their Classification

The subject of program testing can be approached from two angles [HOWD78]: theoretical and empirical.

The theoretical approach calls for the characterization of situations where it is possible to use testing to prove formally the correctness of programs. This approach relies upon the application of graph theoretic and algebraic methods. Gourlay [GOUR83] provides a mathematical framework for investigation of testing.

The empirical approach relies upon collection of statistics regarding the frequency with which different testing strategies reveal the errors existing in a collection of programs [HOWD78]. Several testing strategies such as path testing, branch testing, structured testing, special values testing and symbolic testing fall under this category [HOWD78].

Although each of these approaches, theoretical and empirical, have their respective advantages and disadvantages, Howden [HOWD78] contends that the greatest practical benefits could accrue from the continuance of empirical studies rather than theoretical studies.

According to Adrion, Branstad, and Cherniavsky [ADRI82], a program is to be viewed as a representation of a function. This function is considered as being capable of describing the relationship of an input element called a "domain element" to an output element called a "range element". The testing process is then used to ensure that a program faithfully realizes the function that it was originally intended to perform. They go on to say that program test methods can be classified into two broad categories, dynamic and static analysis techniques. This form of classification finds concurrence in many other published sources [MILL84, DEMI87, ANDR86, HOWD81b, ONOM87, and others]. In the case of dynamic analysis, the program is run with some test instances and the results of the program's performance obtained thereby are used to check whether its actual behavior conforms with the expected behavior. Static analysis, on the other hand, typically involves some form of conceptual execution. Static analysis does not usually involve actual program execution.

There are a host of other methods of classifying testing strategies. It would be relevant to mention some of the other prominent methods: black-box and white-box testing [DEMI87, CHOW85, NTAF84, ONOM87], error-driven strategies [DEMI87, NTAF84, DEMI78, GOOD75, LIN89], top-down testing and bottom-up testing [DEMI87], and symbolic testing [DEMI87, MILL77, MILL81, KING76, MILL84, ADRI82] . Another interesting testing strategy is that of domain testing [ONOM87, WHIT80, WEYU80].

The work done by Ntafos [NTAF88], and Basili and Selby [BASI87] offers an interesting insight into the methodology of comparing several testing strategies. The end results of their work is useful in making a comparison among different testing strategies. Ntafos [NTAF88] compares a host of structural testing strategies in terms of their relative coverage of a program's structure and also in terms of the number of test cases needed to satisfy each strategy. He also points out the attendant shortcomings of such comparisons. Also, a study comprising the application of state-of-the-practice software testing techniques such as code reading by stepwise abstraction, functional testing using equivalence partitioning and boundary value analysis, and structural testing using 100 percent statement coverage criteria can be found in [BASI87].

According to Prather and Myers [PRAT87], the theory of program testing diverges into two separate streams: functional testing [WEYU80, HOWD81b, ANDR86, MILL81, CHOW85] and structural testing [PRAT87, FURU87, LIN89, WOOD80, HOWD81c, HOWD76, HOWD81b, HUAN75].

Prather and Myers [PRAT87] point out the highlights of the functional and structural testing strategies. Functional testing involves the use of a program's specification in designing an "adequate test". Structural testing, on the other hand, requires a careful study of the problem at hand, based upon which an attempt is made to partition the problem. In the latter case, an attempt is made to use the program flow graph in designing an "adequate" test. The concept of an "adequate test" appeared first in an article by Goodenough and Gerhart [GOOD75].

From Adrion et al. [ADRI82], a complete verification of a program, at any stage in the software life cycle, can be obtained only by testing the program with every element in the domain. A program is said to have been verified, if and only if each test instance is successful. In the event that the program should fail for even a solitary test instance, an error is said to have been found. Such a method of testing is called "exhaustive testing". Exhaustive testing is the only dynamic analysis technique that will guarantee the validity of a program. However, this technique obviously is not practically feasible [ADRI82]. The failure of this technique on the grounds of practical feasibility could be attributed to the size of the functional domains, which are infinite more often than not.

In the event that the functional domain of a program is finite, it can still be large enough to cause the number of test instances required to be prohibitively large. Therefore, it is necessary to find a way of reducing this potentially infinite exhaustive testing process to a practically feasible one. This can be accomplished by finding a "criterion" for choosing a number of representative elements from the functional domain. This concept of "criteria" (or more specifically "testing criteria") is discussed in greater detail in Section 2.3. At this point it would be sufficient to say that many criteria have been suggested to date. These criteria may act to portray the functional description or the structure of a program.

As pointed out by Adrion et al. [ADRI82], an important part of the testing problem is to find an "adequate test set". The testing process involves the choice of a subset of elements called a "test set". The test set that is chosen should be large enough to span the domain and yet small enough to ensure that the testing process itself can be carried out for each element in the test set. Such a test set is said to be an "adequate test set" [ADRI82].

The first formal treatment for determining when a criterion for test set collection is adequate, appeared in [GOOD75]. Goodenough and Gerhart [GOOD75] define a criterion "C" which is said to be reliable if the test sets T_1 and T_2 chosen by "C" are such that all test instances of T_1 are successful exactly when all test instances of T_2 are successful. The criterion "C" is said to be "valid" if it can produce test sets that uncover all errors. These definitions lead to the fundamental theorem of testing which states [ADRI82]:

> If there exists a consistent, reliable, valid, and complete criterion for test set selection for a program P and if a test set satisfying the criterion is such that all test instances succeed, then the program P is correct.

Since the objective of this thesis is to develop an adaptive, graph-theoretic, and matrix-based testing strategy, it would be relevant to identify the class of testing strategies to which it belongs. Clearly, such a strategy would fall into the broad category of structural testing because of its reliance on the flowgraphs of programs. Consequently, it is appropriate that the emphasis of this discussion from this point onwards, should lie in the field of structural testing.

The structural testing methodology in turn, can be divided into three distinct phases [PRAT87]:

1. program graph construction,
2. test path selection, and
3. test case selection.

These three phases of structural testing are dealt with independently in the following subsections. This discussion is followed by separate sections on adaptive testing strategies, complexity measures (metrics), and automated testing tools.

2.3 Structural Testing Considerations

The structural testing methodology can be divided into three phases [PRAT87] as shown in Section 2.2. The following three subsections deal with these phases.

2.3.1 Program Graph Construction

A graph is a collection of nodes and pairs of nodes called arcs [HO84]. The nodes are used to represent the elements of a structure while the arcs are used to represent their interrelationships.

The program graph construction phase involves the "annotation" of the source code listing to derive the underlying flowgraph as a collection of vertices and edges [PRAT87].

According to Miller [MILL79], the theory of testing relies largely upon two forms of graph-theory-based modelling of program properties. They are known as control flow analysis and data flow analysis. The application of graph-theory in the field of program testing is widespread [HOWD81b]. The adoption of the graph-theoretic approach permits us to analyze programs and infer data about suitable test forms directly from the control and/or data structure of the program [MILL81]. The control flow and data flow in a program can be modelled using graph theory techniques [HOWD81b].

In program testing, the graph-theoretic model used assigns arcs in a directed graph (digraph) to actions or segments in the program, and nodes in the digraph to represent locations in a program. Such a model is obviously well suited to program testing because the control structure of a program in any language with a deterministic decisional structure can be represented as a finite, possibly disconnected, directed graph with a single entry node and a single exit node [MILL79]. Such representations make use of the assumption that a program is constructed purely with the standard structured programming conventions, i.e., succession, alteration, and iteration [MILL79].

There are numerous published sources elucidating the application of graph-theoretic principles to program testing, an excellent example is [HO84] which discusses several classes of models and techniques such as directed graph models of sequential programs, analysis of program structure, and computing network models of reliability.

2.3.2 Test Path Selection

Test path selection, the second phase of the structural testing methodology, involves choosing a finite set $\{p_i\}$ of program paths, with a view towards satisfying one or more "coverage" criteria [PRAT87]. The criteria most often cited in program testing literature are: statement coverage, branch coverage, multiple condition coverage, and path coverage.

According to Tai [TAI79], a "criterion" is needed to select or generate test data and also in the measurement of the level of test thoroughness while testing a program. An ideal test criterion would be one that would guarantee the absence of errors in a program based upon successfully completing execution on test data satisfying the criterion. Howden [HOWD81a] cites the development of a criterion for test completeness. He claims that it is more effective than branch testing and that it incorporates some of the advantages of mutation testing [HOWD81b, ADRI82, HOWD81a].

Three of the most commonly used testing criteria in generating test data and in measuring the level of test thoroughness [TAI79] are:

1. each and every statement is executed at least once,
2. each and every branch is executed at least once, and
3. each and every path is executed at least once.

Goodenough and Gerhart [GOOD75] propose a fundamental theorem of testing, basic definitions for a theory of testing, and criteria for the selection of test items from the domain of possible inputs to a program. In this connection the work done by Gourlay [GOUR83], and Weyuker and Ostrand [WEYU80] are particularly interesting.

2.3.3 Test Case Generation

The final phase of the structural testing methodology is test case generation which involves the determination of a set of test inputs $X = \{x_i\}$ that will "drive" the program through the indicated paths, given that we have selected a set $P = \{p_i\}$ of program paths based upon their having satisfied some test coverage criteria [PRAT87].

The test data generation problem is stated by Miller [MILL81] as follows : "given a part of a program that has not yet been tested, construct specific test data that will cause that part to be executed". This problem is addressed by Goodenough and Gerhart [GOOD75], Weyuker and Ostrand [WEYU80], and Demillo et al. [DEMI78]. Goodenough and Gerhart note that test data selected solely on the basis of program structure in general will be inadequate for the purposes of thorough testing.

2.4 Adaptive Testing Strategies

Conventional test case generation methods are severely limited by their reliance on a set of preselected complete paths to be traversed [PRAT87]. This is because, we are forced to return to the path selection phase in the event that even one of the preselected paths proves to be infeasible. Consequently, Prather and Myers [PRAT87] contend that there is an intrinsic interplay between the path selection phase and the test case generation phase. They go on to say that the virtue of the adaptive approach to testing lies in its ability to exploit this interplay between phases even while acknowledging its existence. As before, this strategy still relies heavily upon the use of a program flowgraph. However, the idea here is to add just one new test path (and hence, one new input test) at a time, using previously traversed paths (inputs) as a guide to the selection of subsequent paths (inputs), in accordance with some inductive strategy [PRAT87].

For the purposes of this thesis the "inductive strategy" referred to by Prather and Myers is defined on the basis of the adjacency matrix developed for the program flowgraph of a program. The motivation for the adaptive testing strategy in question largely accrues from the work done by Prather and Myers and from the book written by Beizer [BEIZ83]. Beizer suggests that successive test paths could be selected as small variations of previously traversed paths while attempting to change only one thing at a time.

2.5 Complexity Metrics

There is a need for developing some objective measure of software, particularly structural complexity which can be considered as an indicator of software "quality" as captured in the structure of a program. In response to this need, several different complexity measures (or metrics) have been proposed [see, for example, PAIG80, HALS77, CHEN78, MCCL78, and SAMA88]. According to Chen [CHEN78], "program complexity is the least known factor in programming activity and it is not easily measured or described and is often ignored during the system planning process".

Some of the complexity-based metrics proposed are: McCabe's cyclomatic complexity [MCCA76], Halstead's software science metrics [HALS77], Chen's maximal intersect number [CHEN78], McClure's invocation complexity [MCCL78], Paige's metrics [PAIG80], and Samadzadeh and Edwards' residual complexity [SAMA88].

McCabe [MCCA76] defines cyclomatic complexity by finding the graph theoretic "basis set". A maximal set of linearly independent paths in a program graph is called a basis set. From well-known results in graph theory, the cyclomatic number of a graph, $V(G)$ is given by

$$V(G) = e - n + p$$

for a graph G with n nodes, e edges, and p connected components. The number of linearly independent program paths through a program graph is given by $V(G) + p$. McCabe calls this number the cyclomatic complexity of the program. The cyclomatic complexity, can therefore be calculated from a program graph as

$$C = e - n + 2p$$

Halstead's metrics [HALS77], rely upon four easily-measured parameters of a program

n_1 = the number of distinct operators in the program,

n_2 = the number of distinct operands in the program.

N_1 = total program operator count

N_2 = total program operand count

Halstead defines the estimated program length in tokens, which is different from the number of statements in a program, by

$$H = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

Halstead's metrics treat paired operators such as "BEGIN...END", "DO...UNTIL", and "FOR...NEXT" as single operators.

The actual Halstead length is calculated as

$$N = N_1 + N_2$$

Halstead also defines a program's vocabulary as the sum of the number of distinct operators and operands given by

$$n = n_1 + n_2$$

Paige [PAIG80], cites four metrics which he claims have found some utility in software test environments. They are

1. The cyclomatic number (C).
2. The level of effort (E) to implement a software module based on the mental discriminations or comparisons required (E is one of Halstead's software science metrics).
3. The nesting level (NX) which indicates the maximum

nesting structure utilized in the program.

4. The iteration level (IX) which indicates the maximum iteration structure utilized in the program.

Of these four metrics, C, NX and IX are structure related measures while E is a syntactic token count measure. Paige concludes, on the basis of the work done by him, that the metrics NX and E are found to be very useful. The utility of the measure NX arises from its ease of determination and also because of its direct relationship to C. On the other hand, the utility of the measure E is obvious since it is the only available measure of the difficulty and the time needed to derive each test [PAIG80].

Chen [CHEN78], proposes a measure of program control complexity from an information theory viewpoint while pointing out the factors which determine the complexity of a computer program. McClure [MCCL78], discusses the probable sources of complexity in a well-structured program and presents a methodology for measuring and controlling the complexity of such programs.

2.6 Automatic Testing Tools

The need for automated testing tools is obvious. In most cases software systems are far more complex than the programmers who developed the system would think they are. In addition to this, the "work" involved in testing is not a very enjoyable one, since it is tedious and time consuming. Several automated testing tools have been developed to date.

Osterweil and Fosdick [OSTE76] developed a static analysis tool, DAVE, for FORTRAN programs. Ramamoorthy and Ho [RAMA75] described the FACES software analysis system. Browne and Johnson [BROW78] described a FORTRAN analysis system which is implemented using a commercially available database-management system (System 2000). Howden [HOWD79] presented the DISSECT system - a symbolic evaluation and program testing system built at the University of California on a PDP-10 LISP environment. Clarke [CLAR79] described a a system that attempts to generate test data automatically for programs that are written in ANSI FORTRAN. Jessop [JESS79] presented the ATLAS system used at Bell Laboratories to test one of their Electronic Switching Systems. This system used a high level of automation to achieve acceptable levels of quality assurance. Finally, Budd and Lipton [BUDD78] discussed a program testing system which relies upon the relatively new concept of program mutation analysis.

CHAPTER III

COMPLEXITY MEASURE ALGORITHM

3.1 Complexity Measure Algorithm Preliminaries

This chapter focuses on the development of a graph-theoretic, matrix-based approach to devise a complexity measure for program testing.

This approach relies upon using the basic number of paths in the control flow graph of a program. The adoption of this means of arriving at a measure is largely dictated by the fact that it is impractical to consider the total number of paths in the graph in question [MCCA76]. Although, a number of algebraic expressions which yield the total number of paths in the graph are either readily available or could be developed easily, it is still not a feasible proposition to consider all the possible paths in a given graph. Even a simple program with a solitary backward branch presents us with the possibility of an infinite number of paths. Consequently, the adoption of a means which utilizes the basic number of paths seems appropriate. It is to be noted that the basic paths in a graph could be utilized to form any other path in the graph by forming appropriate linear combinations [MCCA76].

The approach used in this thesis makes the following assumptions:

1. For a given program we can draw a directed graph (known as the program control flow graph) with unique entry and exit nodes;
2. Each node in the graph corresponds to a block of code in the program with the flow within each block being sequential;
3. Each edge in the directed graph corresponds to the branches taken in the program; and
4. Each node can be reached from the entry node and each node can reach the exit node.

3.2 Complexity Measure Algorithm

This algorithm is aimed at computing the complexity of a structured program from the adjacency matrix of its control flow graph. The algorithm is outlined below.

1. Develop the directed graph representation (i.e., the control flow graph) of a given program.
2. Develop the adjacency matrix of the control flow graph.
3. Add another column to the adjacency matrix after the last existing column and label it "# of links in the open chain".
4. Starting from the top row and working downwards identify all rows which contain two or more "1" entries. The existence of two or more "1" entries in any particular row signifies the fact that the node label against that

row represents a decision node.

5. Disregard all other rows which have either a single "1" entry or none at all. This is because a row which exhibits such a feature corresponds to a node that is not a decision node. It could be a node which appears sequentially in the control flow graph, it can be a collecting node, or a sink node.
6. Starting with the first identified row in Step 5 and working downwards carry out the following procedure:
   6.1. Locate the first "1" entry in that row. Then locate the next occurrence of a "1" in the same row. Encircle these two siblings which need not necessarily be consecutive entries of the same row of the adjacency matrix. They could have one or more "0" entries separating them, in which case the intervening "0" entries are disregarded.
   6.2. Look for the next sibling. Encircle the last and the next siblings. Obviously, the second circle overlaps the first one since a sibling is shared between the two circles.
   6.3. Continue this procedure until all the siblings are exhausted. At this point there should be an "open chain" consisting of one or more circles linked together, with the two outermost circles each having one sibling apiece which is not shared. Each circle in the chain will be

called a "link of a chain" hereafter.

6.4. Count the number of circles in the "open chain". Enter the number so obtained, in the same row and in the last column that was added to the original adjacency matrix.

7. Enter a "0" against all rows that were disregarded (because they had only one "1" entry or none at all) in the last column labelled "# of links in the open chain".

8. Compute the sum of all entries in the last column of the modified adjacency matrix. Add 1 to this sum. Call this value "C(G)". C(G) is the cyclomatic complexity of the graph in question.

An examination of the adjacency matrix and the algorithm shows that the complexity is not dependent directly on the actual size of the program (e.g., in terms of the number of lines of code).

3.3 Identification of a Set of Basic Paths

The set of basic paths identified by following the algorithm outlined below is by no means unique [PAIG80]. The algorithm outlined in this subsection identifies a set of basic paths from the adjacency matrix of the control flow graph of a program.

1. Begin with the unique entry node for each basic path, that is, start with the first row of the adjacency matrix each time around.

2. Look for the "1" entry/entries in the first row of the adjacency matrix and note down the corresponding row label first. Then write down the corresponding column label next to it. Move down to the row having the same row label as the column label of the "1" entry just identified. Look for the occurrence(s) of "1" entries. Then, note down the corresponding column label next to the list of node labels. Continue this procedure until the unique exit node is reached. No single graph node is to be traversed more than twice in any single basic path. this double traversal is permitted in order to provide for the possible existence of backward loops.
3. Repeat this procedure with the next occurrence of a "1" entry in the first row. Continuation along these lines will eventually yield a set of node label lists each of which corresponds to a basic path and the number of such paths, should be equal to the value of C(G) previously computed (Section 3.2).

The complexity measure algorithm outlined in Section 3.2 yields a measure of the complexity of a program by computing the value C(G) from the adjacency matrix of its graph. This value corresponds to the number of linearly independent paths in the graph. The procedure outlined above identifies a set of basic paths for the graph being considered.

3.4 Examples

The application of the complexity measure algorithm to some example graphs from McCabe's work [MCCA76] appears in Appendix A.

CHAPTER IV

THE ADAPTIVE TESTING STRATEGY

The testing strategy proposed in this chapter is adaptive in nature. A graph-theoretic, matrix-based approach was adopted in arriving at this strategy. This strategy utilizes the adjacency, incidence, and path matrices of the program flow graph of a structured program. This strategy is hinged upon a few modifications that are made to some of these matrices. In the case of the adjacency matrix, the modifications made are useful in demonstrating the achievement of "branch coverage". The path matrix is constructed using the paths generated by the application of the adaptive testing strategy. The modifications made to the path matrix are useful in illustrating the attainment of complete "node coverage" and "edge coverage".

4.1 Adaptive Testing Strategy Preliminaries

This section deals with the preliminaries required for the discussion of the adaptive testing strategy. As mentioned before, this testing strategy required that some modifications be made to the adjacency matrix. These modifications are dealt with in Subsection 4.1.1. Another matrix called the "Branch Coverage Matrix" is also required, which is dealt with in Subsection 4.1.2.

4.1.1 Modifications Proposed for the Adjacency Matrix

The basic adjacency matrix is constructed using the directed graph representation of a given program (i.e., the control flow graph has unique entry and exit nodes). The basic adjacency matrix has as many rows and columns as the number of nodes in the control flow graph. This basic adjacency matrix is then modified as follows:

1. Add three more columns to the basic adjacency matrix after the last column and label them "base value column", the "weighted digital signature column", and "enhanced value column".
2. Starting from the top-most row and working downwards, identify all the rows which contain two or more "1" entries (signifying decision nodes). Count the number of "1" entries in all the rows identified thereby and enter the values so obtained in the "base value column" against the respective row. In this process of row identification disregard all rows which have a either a single "1" entry or none at all. However, a "0" entry is to be made against such rows in the "base value column".
3. Identify all the non-zero entries in the "base value column". Fill all locations in the "weighted digital

signature column" with corresponding non-zero entries in the "base value column" with the value "3" (called the "key value" hereafter). Insert "0" entries in all other locations.
The "key value" of "3" could be replaced by any other positive number. This is because the purpose of using this "key value" is merely to have a recognizable quantity once the strategy has run through its full course. The significance of the use of a "key value" will become apparent when the algorithm is outlined in detail(see Section 4.2).

4. In the last column labelled as the "enhanced value column", make an entry equal to the sum of the values in the "base value column" and the "weighted digital signature column" against the respective rows.

When this process has been completed, the last three columns of the modified adjacency matrix should contain non-zero entries against all the rows identified in Step 2 above (representing decision nodes), and "0" entries against all other rows (representing sequential nodes). Obviously, the non-zero entries in the "base value column" represent the number of children that the respective decision nodes possess.

4.1.2 Branch Coverage Matrix

Another matrix called the "Branch Coverage Matrix" is also constructed which is an important part of the adaptive testing strategy. This matrix is constructed as follows:

1. Set up the matrix with as many rows as there are nodes in the program flow graph.
2. Identify the decision node with the largest number of children (easily recognized by observing the values in base value column of the modified adjacency matrix described in Section 5.1.1). Then the number of columns required for this matrix is computed as follows:

   # of columns = (largest # of children as above) + 2

   The numeral "2" in the above expression is not a magic number. This number in fact represents the need for two additional columns. One of these is used to carry a replica of the "enhanced value column" from the modified adjacency matrix, and the other column is required to house the "residual digital signature". The existence of a tie for the largest number of children does not affect the situation in any way. This is because the number of columns required would be the same as would have been needed in the absence of a tie.

If, for example, the decision node with the maximum number of children were to have 2 children (could even be a case statement) and if we had several other binary decision nodes in a 10 node decision matrix, the corresponding "Branch Coverage Matrix" would probably look like the one shown in TABLE I on the next page.

TABLE I

SAMPLE BRANCH COVERAGE MATRIX

	EVC	1	2	RDS
1	5	$\cancel{2}$-1	$\cancel{3}$-1	3
2	0	0	0	0
3	5	$\cancel{4}$-1	$\cancel{5}$-1	3
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	0	0	0	0
10	0	0	0	0

row labels represent node numbers
column labels represent child node counts
EVC represents Enhanced Value Column
RDS represents Residual Digital Signature

Now that the number of rows and columns required for this matrix have been computed, the task of filling up the matrix remains. This matrix is then filled by following the procedure outlined below:

1. Fill the first column with a replica of the "enhanced value column" from the modified adjacency matrix.
2. Identify the rows representing decision nodes (rows containing non-zero entries).
3. Fill the node labels of the children of all the

decision nodes in the corresponding rows from left to right after the EVC entry.

4. The decision(s) with fewer children than the one with the maximum number of children will have some vacant spaces. Pad these spaces with "0" entries.
5. Fill the rows against all the non-decision nodes with "0" entries. This includes the corresponding locations on the "residual digital signature column" which is the last column in this matrix.

This process should leave a matrix completely filled except for the locations against the decision nodes in the last column called "residual digital signature column". The contents of these remaining locations will be decided in the course of the application of the adaptive testing strategy.

4.2 Adaptive Testing Strategy

The proposed adaptive strategy is expected to yield a set of program paths, $P = \{p_i\}$ which meet the "branch coverage" and "node coverage criteria". Now, let the set of test inputs required to drive the program through the indicated paths be $X = \{x_i\}$.

This strategy is adaptive in nature because a clearly recognizable digital signature called the "residual digital signature" is left behind whenever a particular path is traversed. On subsequent searches for other paths, repeated traversals of previously traversed nodes is avoided by recognizing the digital signature, left behind during previous traversals. So, in effect, the choice of a path helps us to determine subsequent paths without the attendant threat of making wasteful and expensive repetitions. The adaptive strategy is outlined in detail below:

1. All paths begin at the unique source node (a column of all zeros) and terminate at the unique sink node (a row of all zeros).
2. Consider the incidence matrix of the program flow graph in question. Start the traversal at the source node. It is possible that the source could be a sequential node (i.e., not a decision node). Make a record of the corresponding node label.
3. The row representing the source node should contain one or more "1" entries. Locate the first instance of a "1" entry in this row. The traversal begins at this entry.
4. Traverse the column containing the entry identified in the previous step in a downward fashion until a "-1" entry is encountered. Then, record the node label that corresponds to the row containing the "-1" entry, next to the node label previously recorded (i.e., the source node in this case). The edge connecting the source node and the node identified in this step is the first edge in the path.
5. At this point, start a horizontal search, along the same row until a "1" entry is reached. It is possible that more than one such "1" entries could exist in a row

(i.e., in the case of a decision node). At the first occurrence of a "1" entry in this row, drop down until a "-1" entry is reached lower down in the column containing this entry. Then add the node label of the row containing the "-1" entry to the list of node labels being maintained (which presently consists of the source node and another node). Continue this procedure, recording node labels along the way in the manner specified above.

This procedure will terminate when the unique exit node is reached. The exit node is easily identified when a "-1" entry is encountered and for which no "1" entry can be found along the same row.

6. During the process of traversing a path, whenever a child node of a decision node is traversed go back to "Branch Coverage Matrix" (which also accounts for node coverage), and replace the corresponding node label by a value of "-1". If however such a node is traversed more than once, this replacement is to be carried out only the first time around.
7. This "-1" entry replacing the node labels serve as the "recognizable digital signatures" which are useful in serving as a reminder of the fact that the node in question has been traversed previously. Thus, when the path is being identified the node number which bears the signature of "-1" is avoided and instead another branch is chosen for traversal.

8. This process is carried out starting from the first decision node encountered after traversing the unique source node down to the decision node before the unique sink node and until each child node of every decision node bears the digital signature "-1".
9. A backward loop is identified in this traversal when it is no longer possible to find a "-1" entry upon dropping down from a "1" entry. In such a case look for a "-1" entry above the "1" entry and continue as before with the only difference being that the horizontal search at this juncture is now directed from right to left in the incidence matrix.
10. If at some decision node the children are placed such that one node is in the forward direction (identified when a "-1" entry is reached by dropping down from a "1" entry in the incidence matrix) and the other is reached by looping backwards (identified when a "-1" entry is reached by moving upwards from a "1" entry in the incidence matrix), choose the node in the forward direction the first time around through that decision node. Record the corresponding path as was outlined before. For the next path (with one child node obtained by looping backwards), start out by traversing the path as before. This process is initiated at the unique source node as before and is carried out until the decision node is reached. At this point the backward looping branch is chosen (the forward going branch is

avoided upon encountering the signature value of "-1") and the node labels are recorded as before. It is important to ensure that this loop is traversed only once. This is accomplished easily by avoiding repetitive traversals whenever the signature value is encountered. Then, when the traversal procedure returns to the decision node encountered previously (the branch that was traversed previously is avoided and a branch which was not traversed previously is chosen), simply copy the rest of the path from that point onwards, from the previous path through that decision node. (For example, see path 2 on page 72.)

11. This procedure is completed when all the non-zero node label entries in the child node columns of the node coverage matrix (i.e., branch coverage matrix) bear the digital signature "-1".
12. At this point compute the sum of all the elements in each row of the Branch Coverage Matrix (the sum is zero for all non-decision nodes and has been entered previously) and enter these values in the corresponding locations in the "residual signature column". This column should now consist of only "0" entries and "3" entries (i.e., the key value). This column vector so obtained is called the "residual digital signature".

When the adaptive strategy has run through its full course, it returns the pre-assigned "weighted digital signature" (Section 4.1.1). The "residual digital signature" generated by this strategy should in fact match the previously assigned "weighted digital signature" exactly. Furthermore, complete node and branch coverage are achieved when this strategy is applied. Although it is obvious that edge coverage follows from branch coverage, an additional means of demonstrating edge coverage is illustrated in Section 4.3.

4.3 Modifications Proposed for the Path Matrix

The basic path matrix is constructed with the path numbers representing the rows and the edge numbers representing the columns [DEO74]. If an edge is part of a path, a "1" entry is made against the path in question and in the column assigned for the edge being considered; and "0" entries are made against the edges that are not part of the path. In order to demonstrate the achievement of edge coverage, the basic path matrix is modified slightly. The only modification needed is the addition of a row. The modified path matrix, which is constructed as specified above, would probably look like the one shown in TABLE II on the next page.

In this context it would be relevant to discuss the interpretation of the basic path matrix [DEO74]. In the basic path matrix a column consisting of all "0" entries corresponds to an edge that does not lie on any path between the source node and the sink node. A column of all "1" entries corresponds to an edge that lies in every path between the source node and the sink node. There is no row with all "0" entries because a row in the path matrix represents a path which is made up of edges and there cannot be a path made up of no edges. It is seen that every column in this matrix has at least a single "1" entry since each node in the graph is traversed when the adaptive testing strategy has run through its full course (see TABLES XII and XVI in Appendix B).

TABLE II

SAMPLE MODIFIED PATH MATRIX

	1	2	3	4	5	6	7	8	9
1	1	0	0	1	0	0	1	0	1
2	0	1	1	0	0	1	0	0	1
3	1	0	0	0	1	0	0	1	0
	1	1	1	1	1	1	1	1	1

row labels represent path numbers
column labels represent edge labels
last row indicates coverage of all edges (row of 1's)

4.4 Complete Edge Coverage

Every column of the modified path matrix (see Section 4.3) is checked for the presence of one "1" entry. In the event that a "1" entry is found in a particular column, a "1" entry is made in the last row of the same column, otherwise a "0" is entered at this position. When the process of searching the columns of the modified path matrix for "1" entries is completed, the last row of the modified path matrix should consist of only "1" entries signifying that every edge in the graph is included in at least one path.

Thus, the adaptive testing strategy yields a set of program test paths that provide complete node coverage, path coverage, and hence edge coverage. The residual digital generated by the adaptive testing strategy at the conclusion of its application is indicative of the fulfillment of the said coverage criteria. Relabelling of the nodes in the control flow graph of a program does not produce a different set of paths. The set of paths generated remains the same, the only difference being that the node labels get changed due to the relabelling.

4.5 Examples

The application of the adaptive testing strategy to some example flowgraphs from McCabe's work [MCCA76] appears in Appendix B.

CHAPTER V

SUMMARY, CONCLUSIONS, AND FUTURE WORK

The main theme of this thesis was the development of an algorithm to compute the complexity of structured programs and an adaptive testing strategy using a graph-theoretic matrix-based approach. The approach used in this thesis relies upon the following assumptions:

1. For a given program we can draw a directed graph (known as the program control flow graph) with unique entry and exit nodes;
2. Each node in the graph corresponds to a block of code in the program with the flow within each block being sequential;
3. Each edge in the directed graph corresponds to the branches taken in the program; and
4. Each node can be reached from the entry node and each node can reach the exit node.

Essentially, these assumptions convey the notion that the algorithms developed as part of this thesis apply only to structured programs.

The complexity measure calculated would be useful, amongst other things, in assessing software quality as captured in the structure of a program. A low complexity value is considered desirable and is indicative of high quality. The adaptive testing strategy that has been developed is expected to offer several advantages over conventional testing strategies. These advantages are likely to manifest themselves in the form of significant savings in the cost of the testing process and in having fewer computational requirements when compared with its conventional counterparts which involve the application of costly path selection techniques.

However, the graph-theoretic matrix-based approach adopted for this thesis introduces some attendant limitations. This approach relies heavily upon the use of the incidence matrix of the program control flow graph. The definition of the incidence matrix does not accommodate the existence of self-loops (a node in the graph is a child of itself). This limitation is in turn imposed upon the adaptive testing strategy, thereby limiting its applicability to only structured programs which are devoid of self-loops.

Suggestions for future work include finding a way around the limitation imposed upon the adaptive testing strategy so as to accommodate the existence of self-loops which are fairly commonplace in actual programs. Further, time and space complexity analyses which were not conducted as part of this thesis could be carried out.

Other future work might include the development of an automated testing tool which relies upon the adaptive testing strategy developed as part of this thesis. Such an automated testing tool would be useful in relieving the tedium of testing and possibly contribute towards reducing the amount of time spent in the testing process.

REFERENCES

[ADRI82]
W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky, "Validation, Verification, and Testing of Computer Software", ACM Computing Surveys, pp. 159-192, June 1982.

[ANDR86]
Stephen J. Andriole (Editor), Software Validation, Verification, Testing, and Documentation, Petrocelli Books, 1986.

[BASI87]
Victor R. Basili and Richard W. Selby, "Comparing the Effectiveness of Software Testing Strategies", IEEE Transactions on Software Engineering, Vol. SE-13, No. 12, pp. 1278-1296, December 1987.

[BEIZ83]
Boris Beizer, Software Testing Techniques, Van Nostrand Reinhold Company, 1983.

[BEIZ84]
Boris Beizer, Software System Testing and Quality Assurance, Van Nostrand Reinhold Company, 1984.

[BERG73]
C. Berge, Graphs and Hypergraphs, Amsterdam, The Netherlands North-Holland, 1973.

[BROW78]
J. C. Browne and David B. Johnson, "FAST: A Second Generation Program Analysis System", Proceedings of the 3rd International Conference on Software Engineering, pp. 142-148, Atlanta, Georgia, May 1978.

[BUDD78]
Timothy A. Budd, Richard J. Lipton, Frederick G. Sayward, and Richard A. DeMillo, "The Design of a Prototype Mutation System for Program Testing", Conference Proceedings, National Computer Conference, pp. 623-627, 1978.

[CHEN78]
Edward T. Chen, "Program Complexity and Programmer Productivity", IEEE Transactions on Software Engineering, Vol. SE-4, No. 3, pp. 187-194, May 1978.

[CHOW85]
Tsun S. Chow, "Part 6: Technical Issues: Testing and Validation", IEEE Tutorial: Software Quality Assurance - A Practical Approach, pp. 269-274, 1985.

[CLAR79]
L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs", IEEE Tutorial: Automated Tools for Software Engineering, IEEE Computer Society, pp. 211-218, 1979.

[DEMI78]
Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", Computer, pp. 34-41, April 1978.

[DEMI87]
Richard A. DeMillo, W. Michael McCracken, R. J. Martin, and John F. Passafiume, Software Testing and Evaluation, The Benjamim/Cummings Publishing Company, Inc., 1987.

[DEO74]
Narsingh Deo, Graph Theory with Applications to Engineering and Computer Science, Prentice-Hall Inc., N.J., 1974.

[FURU87]
Zengo Furukawa and Kazuo Ushijima, "A Model for the Testing Support Method with Sequences of a Directed Graph", Proceedings of COMPSAC87, IEEE Computer Society's Computer Software and Applications Conference, pp. 311-316, 1987.

[GOOD75]
John B. Goodenough and Susan L. Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, pp. 156-173, June 1975.

[GOUR83]
John S. Gourlay, "A Mathematical Framework for the Investigation of Testing", IEEE Transactions on Software Engineering, Vol. SE-9, No. 6, pp. 685-709, November 1983.

[HALS77]
M. H. Halstead, Elements of Software Science, Elsevier Publishers, 1977.

[HO84]
Hon S. Ho, "Graph Theoretic Modeling and Analysis in Software Engineering", Handbook of Software Engineering, Van Nostrand Reinhold Company, pp. 26-37, 1984.

[HOWD76]
William E. Howden, "Reliability of the Path Analysis Testing Strategy", IEEE Transactions on Software Engineering, Vol. SE-2, No. 3, pp. 208-215, September 1976.

[HOWD78]
William E. Howden, "Theoretical and Empirical Studies of Program Testing", Proceedings of the 3rd International Conference on Software Engineering, Atlanta, Georgia, May 10-12, pp. 305-311, 1978.

[HOWD79]
William E. Howden, "Dissect - A Symbolic Evaluation and Program Testing System", IEEE Tutorial: Automated Tools for Software Engineering, pp. 207-210, 1979.

[HOWD81a]
William E. Howden, "Completeness Criteria for Testing Program Functions", IEEE Tutorial: Software Testing and Validation Techniques, pp. 67-75, 1981.

[HOWD81b]
William E. Howden, "A Survey of Dynamic Analysis Methods", IEEE Tutorial: Software Testing and Validation Techniques, pp. 209-231, 1981.

[HOWD81c]
William E. Howden, "Functional Testing Design Abstractions", IEEE Tutorial: Software Testing and Validation Techniques, pp. 281-287, 1981.

[HUAN75]
J. C. Huang, "An Approach to Program Testing", ACM Computing Surveys, pp. 113-128, September 1975.

[JESS79]
W. H. Jessop, J. R. Kane, S. Roy, and J. M. Scanlon, "ATLAS-An Automated Software Testing System", IEEE Tutorial: Automated Tools for Software Engineering, pp. 219-225, 1979.

[KING76]
James C. King, "Symbolic Execution and Program Testing", Communications of the ACM, Vol. 19, No. 7, pp. 385-394, July 1976.

[LIN89]
Jin-Cherng Lin and Chyan-Goei Chung, "Zero-One Programming Model in Path Selection Problem", Proceedings of COMPSAC89, Thirteenth Annual International Conference on Computer Software and Applications Conference, Orlando, Florida, pp. 618-627, 1989.

[MCCA76]
Thomas J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, pp. 308-320, December 1976.

[MCCL78]
Carma L. McClure, "A Model for Complexity Analysis",

Proceedings of the 3rd International Conference on Software Engineering, pp. 149-157, Atlanta, Georgia, May 1978.

[MILL77]
Edward F. Miller, "Program Testing: Art Meets Theory", IEEE Computer, pp. 42-51, July 1977.

[MILL79]
Edward F. Miller, "Program Testing Technology in the 1980s", The Oregon Report: Proceedings of the Conference on Computing in the 1980s, pp. 72-79, 1979.

[MILL81]
Edward F. Miller, "Introduction to Software Testing Technology", IEEE Tutorial: Software Testing and Validation Techniques, pp. 4-16, 1981.

[MILL84]
Edward F. Miller, "Software Testing Technology: An Overview", Handbook of Software Engineering, Van Nostrand Reinhold Company, pp. 359-379, 1984.

[MYER79]
G. J. Myers, The Art of Software Testing, A Wiley-Interscience Publication, John Wiley & Sons, 1979.

[NTAF84]
Simeon C. Ntafos, "On Required Element Testing", IEEE Transactions on Software Engineering, Vol. SE-19, No. 6, pp. 795-803, November 1984.

[NTAF88]
Simeon C. Ntafos, "A Comparison of Some Structural Testing Strategies", IEEE Transactions on Software Engineering, Vol. 14, No. 6, pp. 868-874, June 1988.

[ONOM87]
Akira K. Onoma, Tsuneo Yamauara, and Yoshio Kobayashi, "Practical Approaches to Domain Testing: Improvements and Generalization", Proceedings of COMPSAC87, IEEE Computer Software and Applications Conference, pp. 291-297, 1987.

[OSTE76]
L. J. Osterweil and L. D. Fosdick, "DAVE - A Validation Error Detection and Documentation System for FORTRAN Programs", Software Practice and Experience, pp. 473-486, October-December 1976.

[PAIG80]
Michael Paige, "A Metric for Software Test Planning", Proceedings of COMPSAC80, IEEE Computer Society's Fourth International Computer Software and Applications Conference, October 27-31, pp. 499-504, 1980.

[PRAT87]
Ronald E. Prather and J. Paul Myers, Jr., "The Path Prefix Software Testing Strategy", IEEE Transactions on Software Engineering, Vol. SE-13, No. 17, July 1987.

[RAMA75]
C. V. Ramamoorthy and S. F. Ho, "Testing Large Software with Automated Software Evaluation Systems", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, pp. 46-58, March 1975.

[ROBI80]
D. F. Robinson and L. R. Foulds, Digraphs: Theory and Techniques, Gordon and Breach Science Publishers, New York, 1980.

[SAMA88]
M. Samadzadeh and W. Edwards, "A Classification Model of Software Comprehension", Proceedings of the 21st Annual Hawaii International Conference on System Science (HICSS 21), Hawaii, 1988.

[SKVA86]
Romualdas Skvarcius and William B. Robinson, Discrete Mathematics with Computer Science Applications, The Benjamin/Cummings Publishing Company, Inc., 1986.

[TAI79]
Kuo-Chung Tai, "On Program Testing Criteria", Proceedings of COMPSAC79, IEEE Computer Society's Third International Computer Software and Applications Conference, pp. 494-499, 1979.

[TEMP81]
H. N. V. Temperley, Graph Theory and Applications, Ellis Horwood Series in Mathematics and Its Applications, England, 1981.

[WEYU80]
Elaine J. Weyuker and Thomas J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains", IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, pp. 236-246, May 1980.

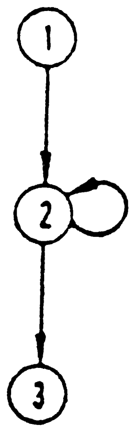
[WHIT80]
Lee J. White and Edward I. Cohen, "A Domain Strategy for Computer Program Testing", IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, pp. 247-257, May 1980.

[WOOD80]
Martin R. Woodward, David Hedley, and Michael A. Hennell, "Experience with Path Analysis and Testing of Programs", IEEE Transactions on Software Engineering, Vol. SE-6, No. 3, pp. 278-285, May 1980.

APPENDIXES

APPENDIX A

EXAMPLES FOR COMPLEXITY MEASURE ALGORITHM



C(G) = 2

Figure 1. Control Flow Graph for Example 1

TABLE III

ADJACENCY MATRIX FOR EXAMPLE 1

	1	2	3	#
1	0	1	0	0
2	0	1	1	1
3	0	0	0	0

row labels represent node numbers
column labels represent node numbers
# represents the number of links in the open chain

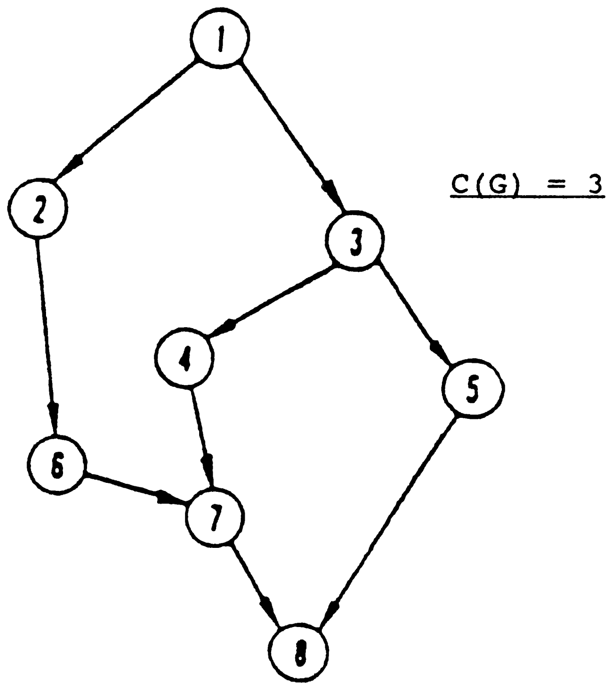


Figure 2. Control Flow Graph for Example 2

TABLE IV

ADJACENCY MATRIX FOR EXAMPLE 2

	1	2	3	4	5	6	7	8	#
1	0	1	1	0	0	0	0	0	1
2	0	0	0	0	0	1	0	0	0
3	0	0	0	1	1	0	0	0	1
4	0	0	0	0	0	0	1	0	0
5	0	0	0	0	0	0	0	1	0
6	0	0	0	0	0	0	1	0	0
7	0	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	0	0	0

row labels represent node numbers
column labels represent node numbers
# represents the number of links in the open chain

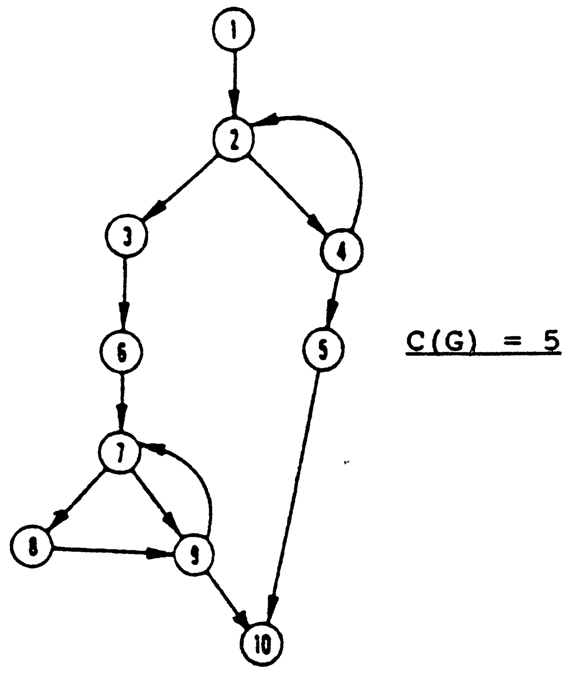


Figure 3. Control Flow Graph for Example 3

TABLE V

ADJACENCY MATRIX FOR EXAMPLE 3

	1	2	3	4	5	6	7	8	9	10	#
1	0	1	0	0	0	0	0	0	0	0	0
2	0	0	1	1	0	0	0	0	0	0	1
3	0	0	0	0	0	1	0	0	0	0	0
4	0	1	0	0	1	0	0	0	0	0	1
5	0	0	0	0	0	0	0	0	0	1	0
6	0	0	0	0	0	0	1	0	0	0	0
7	0	0	0	0	0	0	0	1	1	0	1
8	0	0	0	0	0	0	0	0	1	0	0
9	0	0	0	0	0	0	1	0	0	1	1
10	0	0	0	0	0	0	0	0	0	0	0

row labels represent node numbers
column labels represent node numbers
# represents the number of links in the open chain

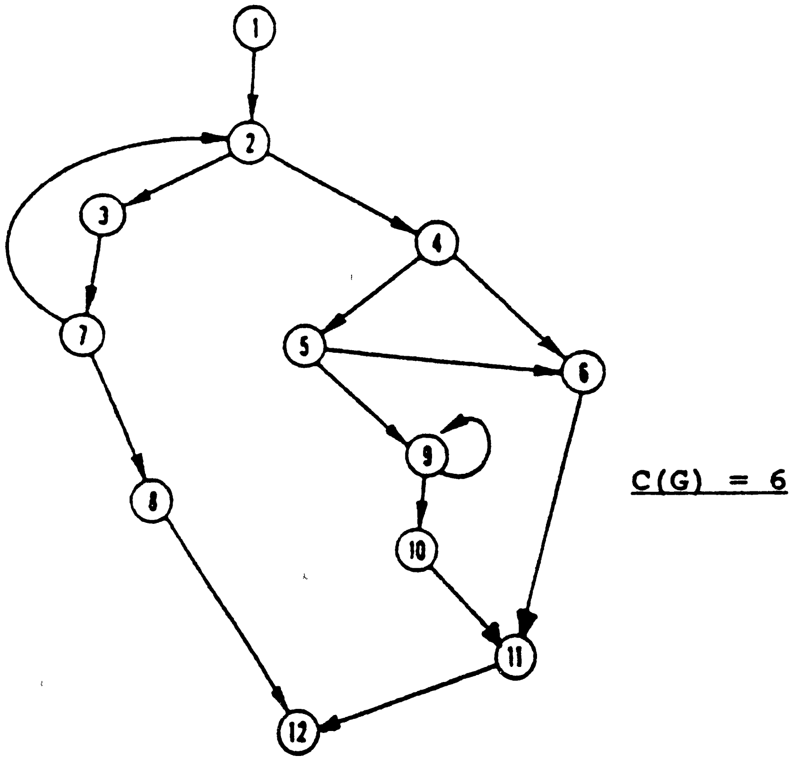


Figure 4. Control Flow Graph for Example 4

TABLE VI

ADJACENCY MATRIX FOR EXAMPLE 4

	1	2	3	4	5	6	7	8	9	10	11	12	#
1	0	1	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	1	0	0	0	0	0	0	0	0	1
3	0	0	0	0	0	0	1	0	0	0	0	0	0
4	0	0	0	0	1	1	0	0	0	0	0	0	1
5	0	0	0	0	0	1	0	0	1	0	0	0	1
6	0	0	0	0	0	0	0	0	0	0	1	0	0
7	0	1	0	0	0	0	0	1	0	0	0	0	1
8	0	0	0	0	0	0	0	0	0	0	0	1	0
9	0	0	0	0	0	0	0	0	1	1	0	0	1
10	0	0	0	0	0	0	0	0	0	0	1	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0

row labels represent node numbers
column labels represent node numbers
# represents the number of links in the open chain

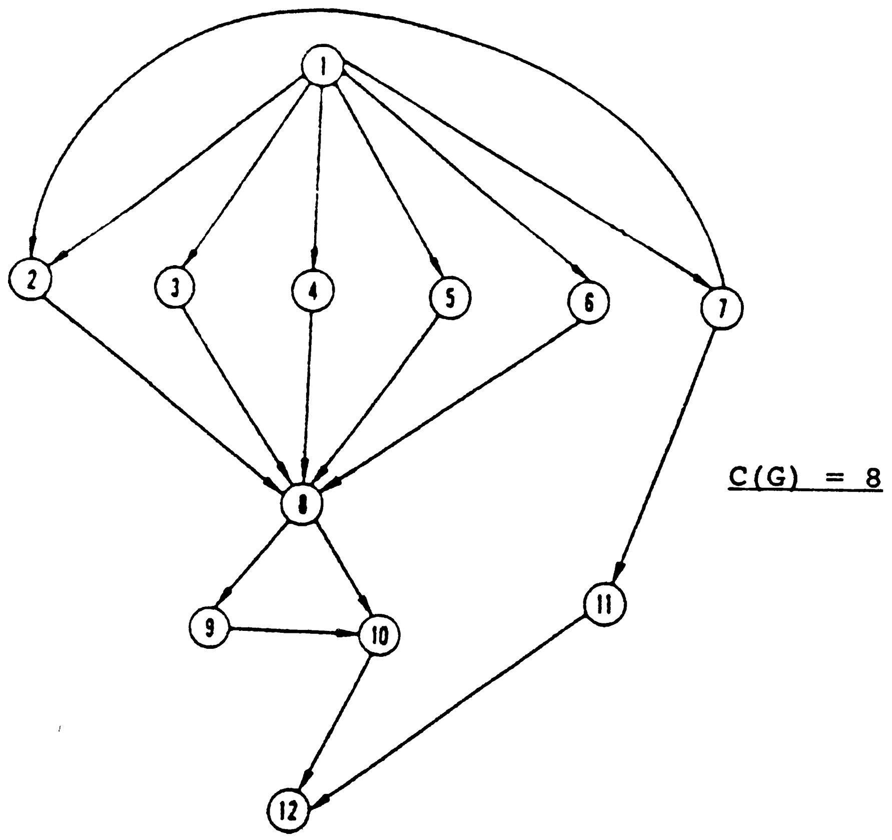


Figure 5. Control Flow Graph for Example 5

TABLE VII

ADJACENCY MATRIX FOR EXAMPLE 5

	1	2	3	4	5	6	7	8	9	10	11	12	#
1	0	1	1	1	1	1	1	0	0	0	0	0	5
2	0	0	0	0	0	0	0	1	0	0	0	0	0
3	0	0	0	0	0	0	0	1	0	0	0	0	0
4	0	0	0	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	0	0	1	0	0	0	0	0
6	0	0	0	0	0	0	0	1	0	0	0	0	0
7	0	1	0	0	0	0	0	0	0	0	1	0	1
8	0	0	0	0	0	0	0	0	1	1	0	0	1
9	0	0	0	0	0	0	0	0	0	1	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	1	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0
12	0	0	0	0	0	0	0	0	0	0	0	0	0

row labels represent node numbers
column labels represent node numbers
# represents the number of links in the open chain

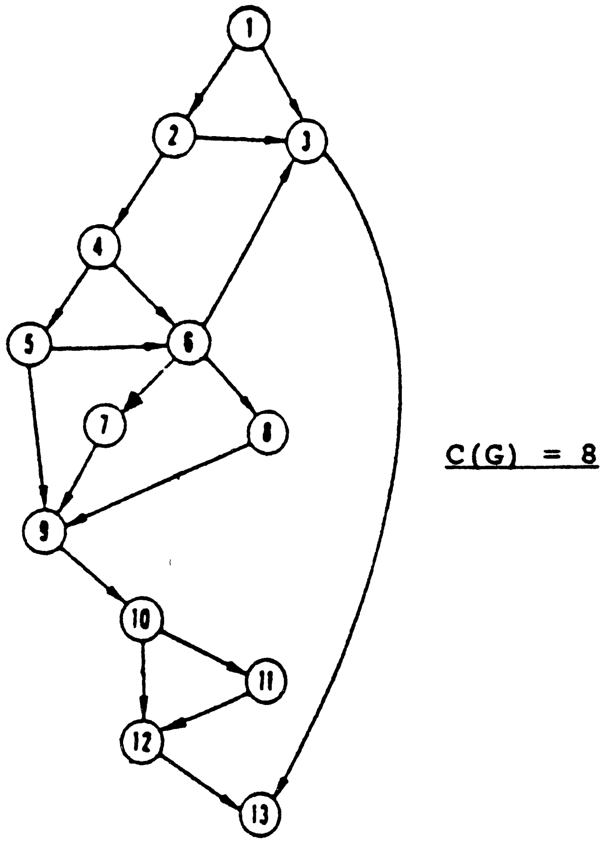


Figure 6. Control Flow Graph for Example 6

TABLE VIII

ADJACENCY MATRIX FOR EXAMPLE 6

	1	2	3	4	5	6	7	8	9	10	11	12	13	#
1	0	1	1	0	0	0	0	0	0	0	0	0	0	1
2	0	0	1	1	0	0	0	0	0	0	0	0	0	1
3	0	0	0	0	0	0	0	0	0	0	0	0	1	0
4	0	0	0	0	1	1	0	0	0	0	0	0	0	1
5	0	0	0	0	0	1	0	0	1	0	0	0	0	1
6	0	0	1	0	0	0	1	1	0	0	0	0	0	2
7	0	0	0	0	0	0	0	0	1	0	0	0	0	0
8	0	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	1	0	1
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0

row labels represent node numbers
column labels represent node numbers
# represents the number of links in the open chain

APPENDIX B

EXAMPLES FOR ADAPTIVE TESTING STRATEGY

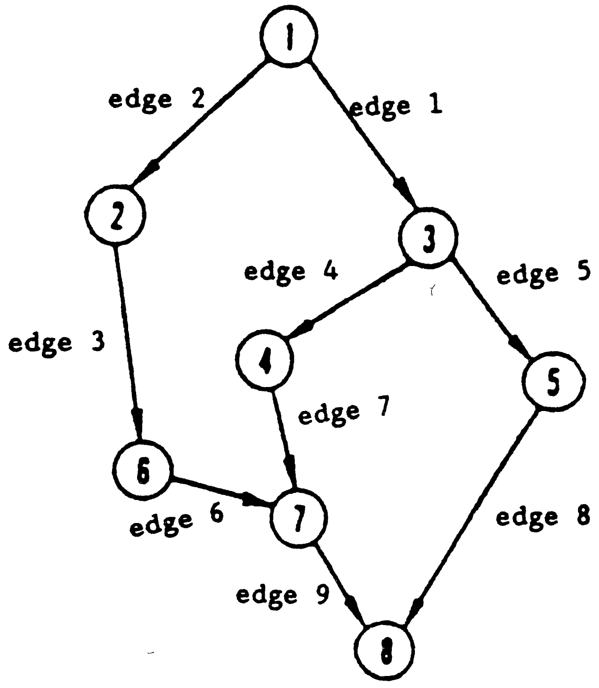


Figure 7. Control Flow Graph for Example 7

TABLE IX

MODIFIED ADJACENCY MATRIX FOR EXAMPLE 7

	1	2	3	4	5	6	7	8	#	BVC	WDS	EVC
1	0	1	1	0	0	0	0	0	1	2	3	5
2	0	0	0	0	0	1	0	0	0	0	0	0
3	0	0	0	1	1	0	0	0	1	2	3	5
4	0	0	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	0	0
7	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0

row labels represent node numbers
column labels represent node numbers
# represents the number of links in the open chain
BVC represents Base Value Column
WDS represents Weighted Digital Signature
EVC represents Enhanced Value Column

TABLE X

INCIDENCE MATRIX FOR EXAMPLE 7

	1	2	3	4	5	6	7	8	9
1	1	1	0	0	0	0	0	0	0
2	0	-1	1	0	0	0	0	0	0
3	-1	0	0	1	1	0	0	0	0
4	0	0	0	-1	0	0	1	0	0
5	0	0	0	0	-1	0	0	1	0
6	0	0	-1	0	0	1	0	0	0
7	0	0	0	0	0	-1	-1	0	1
8	0	0	0	0	0	0	0	-1	-1

row labels represent node numbers
column labels represent edge labels

TABLE XI

BRANCH COVERAGE MATRIX FOR EXAMPLE 7

	EVC	1	2	RDS
1	5	~~2~~-1	~~3~~-1	3
2	0	0	0	0
3	5	~~4~~-1	~~5~~-1	3
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0

Path 1: 1,3,4,7,8

Path 2: 1,2,6,7,8

Path 3: 1,3,5,8

row labels represent node numbers
column labels represent child node counts
EVC represents Enhanced Value Column
RDS represents Residual Digital Signature

TABLE XII

MODIFIED PATH MATRIX FOR EXAMPLE 7

	1	2	3	4	5	6	7	8	9
1	1	0	0	1	0	0	1	0	1
2	0	1	1	0	0	1	0	0	1
3	1	0	0	0	1	0	0	1	0
	1	1	1	1	1	1	1	1	1

row labels represent path numbers
column labels represent edge labels
last row indicates coverage of all edges (row of 1's)

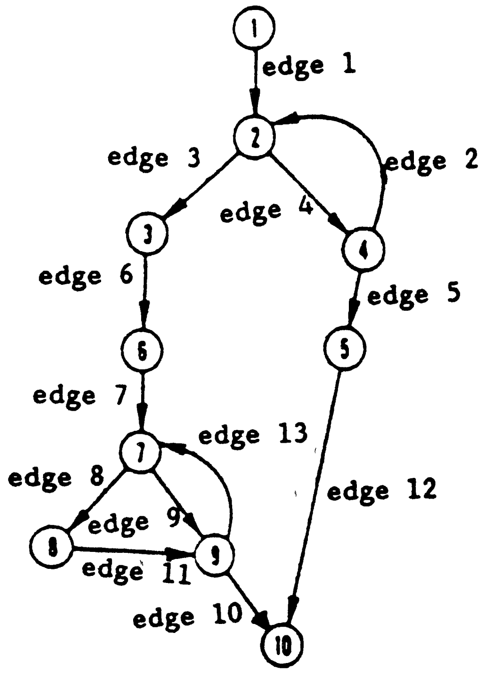


Figure 8. Control Flow Graph for Example 8

TABLE XIII

MODIFIED ADJACENCY MATRIX FOR EXAMPLE 8

	1	2	3	4	5	6	7	8	9	10	#	BVC	WDS	EVC
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	1	0	0	0	0	0	0	1	2	3	5
3	0	0	0	0	0	1	0	0	0	0	0	0	0	0
4	0	1	0	0	1	0	0	0	0	0	1	2	3	5
5	0	0	0	0	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	1	0	1	2	3	5
8	0	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	1	0	0	1	1	2	3	5
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0

row labels represent node numbers
column labels represent node numbers
# represents the number of links in the open chain
BVC represents Base Value Column
WDS represents Weighted Digital Signature
EVC represents Enhanced Value Column

TABLE XIV

INCIDENCE MATRIX FOR EXAMPLE 8

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	-1	-1	1	1	0	0	0	0	0	0	0	0	0
3	0	0	-1	0	0	1	0	0	0	0	0	0	0
4	0	1	0	-1	1	0	0	0	0	0	0	0	0
5	0	0	0	0	-1	0	0	0	0	0	0	1	0
6	0	0	0	0	0	-1	1	0	0	0	0	0	0
7	0	0	0	0	0	0	-1	1	1	0	0	0	-1
8	0	0	0	0	0	0	0	-1	0	0	1	0	0
9	0	0	0	0	0	0	0	0	-1	1	-1	0	1
10	0	0	0	0	0	0	0	0	0	-1	0	-1	0

row labels represent node numbers
column labels represent edge labels

TABLE XV

BRANCH COVERAGE MATRIX FOR EXAMPLE 8

	EVC	1	2	RDS
1	0	0	0	0
2	5	$\cancel{3}-1$	$\cancel{4}-1$	3
3	0	0	0	0
4	5	$\cancel{2}-1$	$\cancel{5}-1$	3
5	0	0	0	0
6	0	0	0	0
7	5	$\cancel{8}-1$	$\cancel{9}-1$	3
8	0	0	0	0
9	5	$\cancel{7}-1$	$\cancel{10}-1$	3
10	0	0	0	0

<u>Path 1</u>: 1,2,3,6,7,8,9,7,9,10

<u>Path 2</u>: 1,2,4,2,4,5,10

row labels represent node numbers
column labels represent child node counts
EVC represents Enhanced Value Column
RDS represents Residual Digital Signature

TABLE XVI

MODIFIED PATH MATRIX FOR EXAMPLE 8

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	1	0	1	0	0	1	1	1	1	1	1	0	1
2	1	0	0	1	1	0	0	0	0	0	0	1	0
3	1	1	0	1	1	0	0	0	0	0	0	1	0
	1	1	1	1	1	1	1	1	1	1	1	1	1

row labels represent path numbers
column labels represent edge labels
last row indicates coverage of all edges (row of 1's)

APPENDIX C

COMPLEXITY MEASURE PROGRAM LISTING

```
{***********************************************************}
{*                                                         *}
{*           Complexity Measure Program Listing            *}
{*                                                         *}
{***********************************************************}
{*                                                         *}
{*           Author:  Shankar Narayanaswamy                *}
{*           Date:    05/30/91                             *}
{*           Class:   COMSC 5000 - Thesis                  *}
{*           Adviser: Dr. Mansur Samadzadeh                *}
{*                                                         *}
{***********************************************************}
{*                                                         *}
{* Procedures Used:                                        *}
{* ----------------                                        *}
{* pause, clearscreen, printlines, initialize,             *}
{* read_matrix_values, print_matrix, print_entryexit_nodes*}
{* read_incidence matrix, print_incidence_matrix,          *}
{* print_proof_matrix, print_child_matrix,                 *}
{* readin_child_matrix, main program.                      *}
{*                                                         *}
{* Input for Program:                                      *}
{* ------------------                                      *}
{* 1.  Adjacency Matrix for the Control Flow Graph.        *}
{* 2.  The number of nodes in the Control Flow Graph.      *}
{* 3.  The Incidence Matrix for the Control Flow Graph.    *}
{* 4.  The number of edges in the Control Flow Graph.      *}
{*                                                         *}
{* Output of Program:                                      *}
{* ------------------                                      *}
{* 1.  Prints the Adjacency Matrix on the screen.          *}
{* 2.  Prints the value of C(G), i.e., the Complexity.     *}
{* 3.  The identity of the Unique Entry Node.              *}
{* 4.  The identity of the Unique Exit Node.               *}
{* 5.  Prints the Incidence Matrix on the screen.          *}
{* 6.  Generates the child matrix (each row contains       *}
{*     the number of children possessed by each node       *}
{*     followed by the node labels of the children of the *}
{*     respective node) which is used by the adaptive      *}
{*     testing strategy program.                           *}
{*                                                         *}
{* Program Function                                        *}
{* ----------------                                        *}
{* The program accepts input in the format specified above*}
{* Given the input in this format the program generates    *}
{* the complexity number for the program in question       *}
{* according to the Complexity Measure Algorithm.          *}
{* Adaptive Testing Strategy.                              *}
{*                                                         *}
{* Debugging Tools Used:                                   *}
{* ---------------------                                   *}
{* Turbo Pascal Debugger                                   *}
{*                                                         *}
{***********************************************************}
```

```
program adaptive_test;

const
  MAX_SIZE = 100;


var
  matrix : array [1..MAX_SIZE,1..MAX_SIZE+1] of integer;
  childmatrix : array [1..10,1..10] of integer;
  incmatrix : array [1..MAX_SIZE,1..MAX_SIZE] of integer;
  proofmatrix : array [1..MAX_SIZE,1..MAX_SIZE] of integer;
  ndary : array [1..2] of integer; {saves entry/exit}
                        {node labels}
  n : integer;   {# of nodes in the control flow graph}
  totedges : integer; {# of edges in the control flow graph}
  total_sum : integer; {total # of links in the open chains}
  entrynodeflag,exitnodeflag : boolean;
  setentry,setexit,selfloopset,doneonce : boolean;
  uen,uxn : integer; {save unique entry and exit nodes}
  bigchild : integer; {to save the maximum # of children}
                {possessed by any decision node in}
                {the control flow graph}
  maxcols : integer; {# of columns for proof of branch}
               {coverage matrix}
  origin  : integer;
  edge_direction : integer;

{**********************************************************}
{*                                                        *}
{*  procedure pause                                       *}
{*  ---------------                                       *}
{*  This procedure is used to generate a pause during the *}
{*  execution of the program                              *}
{*                                                        *}
{**********************************************************}

procedure pause;

begin
  writeln;
  writeln('Hit <Enter> to continue ...');
  readln;
end;

{**********************************************************}
{*                                                        *}
{*  procedure clearscreen                                 *}
{*  ---------------------                                 *}
{*  This procedure is used to clear the screen during the *}
{*  execution of the program.                             *}
{*                                                        *}
{**********************************************************}
```

```
procedure clearscreen;

const
  scrnlimit = 25;

var
  int : integer;
begin
  for int := 1 to scrnlimit do
   writeln;
end;

{***********************************************************}
{*                                                         *}
{*  procedure printlines                                   *}
{*  --------------------                                   *}
{*  This procedure is used to generate a specified number *}
{*  of lines which is passed to it as a parameter.        *}
{*                                                         *}
{***********************************************************}

procedure printlines(z: integer);

var
  i : integer;

begin
  for i := 1 to n do
   writeln;
end;

{***********************************************************}
{*                                                         *}
{*  procedure initialize                                   *}
{*  --------------------                                   *}
{*  This procedure is used to initialize all the global   *}
{*  variables.                                             *}
{*                                                         *}
{***********************************************************}

procedure initialize;
var
  a,b,c : integer;
begin
  setentry := false;
  setexit := false;
  entrynodeflag := false;
  exitnodeflag := false;
  for a := 1 to 2  do
   ndary[a] := -1; {initialize entry and exit}
               {node labels to -1}
  selfloopset := false;
  doneonce := false;
  bigchild := 0;
  edge_direction := 1;
end;

{*********************************************************}
{*                                                       *}
{*  procedure read_matrix_values                         *}
{*  ----------------------------                         *}
{*  This procedure is used to read in the adjacency matrix*}
{*  supplied by the user.                                *}
{*                                                       *}
{*********************************************************}

procedure read_matrix_values;
var
  infile : text;
  i,j,sum,a1,a2,a3,a4,a5,cntrl : integer;
  fname : string;

begin
  write('Enter adjacency matrix file name: ');
  readln(fname);
  assign(infile,fname);
  reset(infile);
  write('Enter # of nodes in control flow graph: ');
  readln(n);
  for i := 1 to n do
  begin
    for j := 1 to n do
      read(infile,matrix[i,j]);
    readln(infile);
  end;
  for i := 1 to n do
  begin
    sum := 0;
    for j := 1 to n do
      sum := sum + matrix[i,j];
    if (sum =1) or (sum = 0) then
      matrix[i,n+1] := 0
    else
      matrix[i,n+1] := sum -1;
  end;
  close(infile);
  for a1 := 1 to n do
    begin
      if (matrix[a1,n+1] > bigchild) then
     bigchild := matrix[a1,n+1];
    end;
  bigchild := bigchild + 1; {largest # of children}
                  {of any decision node }
  maxcols := bigchild + 2; {number of columns}
                 {in proof matrix}
  for a2 := 1 to n do
    for a3 := 1 to maxcols do
      proofmatrix[a2,a3] := 0;
  for a2 := 1 to n do {generate EVC for proof matrix}
   if (matrix[a2,n+1] >= 1) then
     proofmatrix[a2,1] := matrix[a2,n+1] + 1 + 3;
  { fill up proof matrix }
  for a4 := 1 to n do
      begin
     cntrl := 2;
     if (matrix[a4,n+1] >= 1) then
       for a5 := 1 to n do
         begin
           if (matrix[a4,a5] = 1) then
          begin
            proofmatrix[a4,cntrl] := a5;
            inc(cntrl);
          end;
         end;
      end;
  total_sum := 0;
  for i := 1 to n do
    total_sum := total_sum + matrix[i,n+1];
    inc(total_sum);
end;

{***********************************************************}
{*                                                         *}
{*  procedure print_matrix                                 *}
{*  ----------------------                                 *}
{*  This procedure prints out the adjacency matrix.        *}
{*                                                         *}
{***********************************************************}

procedure print_matrix(n : integer);
var
  i,j : integer;
begin
  clearscreen;
  write(' ');
  for i := 1 to (((n + 1)* 3) + 6) do
    write('_');
  write(' ');
  writeln;

  write('|');
  for i := 1 to (((n + 1)* 3) + 6) do
    write(' ');
  writeln('|');

  write('|');
  write(' ':4);
  for i := 1 to n+1 do
    write(i:3);
  write('  |');
  writeln;
  write('|');
  for i := 1 to (((n + 1)* 3) + 6) do
    write('_');
  write('|');
  writeln;

  write('|');
  for i := 1 to (((n + 1)* 3) + 6) do
    write(' ');
  writeln('|');

  for i := 1 to n do
  begin
    write('|');
    write(i:2,'|':2);
    for j := 1 to n+1 do
      write(matrix[i,j]:3);
    writeln('  |');
  end;

  write('|');
  for i := 1 to (((n + 1)* 3) + 6) do
    write(' ');
  writeln('|');

  write('\');
  for i := 1 to (((n + 1)* 3) + 6) do
    write('-');
  write('''');
  writeln;
  writeln;
  writeln(' ******* ADJACENCY MATRIX *******');
  writeln;
  pause;
  writeln('   # of links in open chain = ',total_sum-1);
  writeln;
  writeln('   Complexity Measure, C(G) = ',total_sum);
  writeln;
  {writeln('   bigchild is = ',bigchild);}
end;

{**********************************************************}
{*                                                        *}
{*  procedure print_entryexit_nodes                       *}
{*  -----------------------------                         *}
{*  This procedure is used to ascertain and print out the *}
{*  node labels of the unique entry and exit nodes.       *}
{*                                                        *}
{**********************************************************}

procedure print_entryexit_nodes(n : integer);

var
  i,j : integer;
  rowsum,colsum : integer;
  entrynode,exitnode : char;

begin
{WRITELN('AM IN PRINT PATH PROCEDURE');
WRITELN('HIT ENTER TO CONTINUE ...');
READLN;}

 for j := 1 to n do
  begin
   if (not(entrynodeflag)) then
   begin
     {WRITELN('AM IN ENTRY NODE FOR LOOP');
     WRITELN('HIT ENTER TO CONTINUE ...');
     READLN;}
     colsum := 0;
     for i := 1 to n do
      colsum := colsum + matrix[i,j];
     {WRITELN('COLSUM = ', COLSUM);
     WRITELN('ENTRYNODEFLAG IS = ',ENTRYNODEFLAG);}
     if (colsum = 0) and (not(setentry)) then
      begin
       setentry := true;
       entrynodeflag := true;
       if (ndary[1] = -1)  and (entrynodeflag) then
     ndary[1] := j;
      end
     else
      if (colsum = 0) and (setentry) then
       begin
     writeln('   ERROR !! MORE THAN ONE ENTRY NODE !!');
     writeln('   ONLY A UNIQUE ENTRY NODE PERMITTED');
     exit;
       end;
   end;
  end;

 for i := 1 to n do
  begin
   if (not(exitnodeflag)) then
   begin
     {WRITELN('AM IN EXIT NODE FOR LOOP');
     WRITELN('HIT ENTER TO CONTINUE ...');
     READLN;}
     rowsum := 0;
     for j := 1 to n do
      rowsum :=rowsum + matrix[i,j];
     {WRITELN('ROWSUM = ', ROWSUM);
     WRITELN('EXITNODEFLAG IS = ',EXITNODEFLAG);}
     if (rowsum = 0) and (not(setexit)) then
      begin
       setexit := true;
       exitnodeflag := true;
       if (ndary[2] = -1)  and (exitnodeflag) then
      ndary[2] := i;
       end
      else
       if (rowsum = 0) and (setexit) then
        begin
      writeln('   ERROR !! MORE THAN ONE EXIT NODE !!');
      writeln('   ONLY A UNIQUE EXIT NODE PERMITTED');
      exit;
        end;
    end;
  end;

{WRITELN('NDARY[1] = ',NDARY[1]);
WRITELN('NDARY[2] = ',NDARY[2]);}
clearscreen;
if (ndary[1] <> -1) then
 if (ndary[1] <= n) then
  begin
    uen := ndary[1];
    writeln('   Unique Entry Node is = Node #',NDARY[1]);
  end;

if (ndary[2] <> -1) then
 if (ndary[2] <= n) then
  begin
    uxn := ndary[2];
    writeln('   Unique Exit  Node is = Node #',NDARY[2]);
  end;
printlines(12);
end;

{**********************************************************}
{*                                                        *}
{*  procedure read_incidence_matrix                       *}
{*  ------------------------------                        *}
{*  This procedure is used to read in the incidence matrix*}
{*  supplied by the user.                                 *}
{*                                                        *}
{**********************************************************}

procedure read_incidence_matrix;
var
  nextfile : text;
  nexti,nextj,nextsum : integer;
  nextfname : string;

begin
  write('Enter incidence matrix file name: ');
  readln(nextfname);
  assign(nextfile,nextfname);
  reset(nextfile);
  write('Enter # of edges in control flow graph: ');
  readln(totedges);
  for nexti := 1 to n do
  begin
    for nextj := 1 to totedges do
      read(nextfile,incmatrix[nexti,nextj]);
    readln(nextfile);
  end;
  close(nextfile);
end;

{*********************************************************}
{*                                                       *}
{* procedure print_incidence_matrix                      *}
{* ------------------------------                        *}
{* This procedure is used to print out the incidence     *}
{* matrix supplied by the user.                          *}
{*                                                       *}
{*********************************************************}

procedure
print_incidence_matrix(n:integer;totedges:integer);
var
  i,j : integer;
begin
  clearscreen;
  write(' ');
  for i := 1 to (((totedges + 1)* 3) + 4) do
    write('_');
  write(' ');
  writeln;

  write('|');
  for i := 1 to (((totedges + 1)* 3) + 4) do
    write(' ');
  writeln('|');

  write('|');
  write(' ':4);
  for i := 1 to totedges do
    write(i:3);
  write('   |');
  writeln;

  write('|');
  for i := 1 to (((totedges + 1)* 3) + 4) do
    write('_');
  write('|');
  writeln;

  write('|');
  for i := 1 to (((totedges + 1)* 3) + 4) do
    write(' ');
  writeln('|');

  for i := 1 to n do
  begin
```

```
    write('|');
    write(i:2,'|':2);
    for j := 1 to totedges do
      write(incmatrix[i,j]:3);
    writeln('    |');
  end;

  write('|');
  for i := 1 to (((totedges + 1)* 3) + 4) do
    write(' ');
  writeln('|');

  write('\');
  for i := 1 to (((totedges + 1)* 3) + 4) do
    write('-');
  write('''');
  writeln;
  writeln;
  writeln(' ******** INCIDENCE MATRIX *******');
end;

{***********************************************************}
{*                                                         *}
{* procedure print_proof_matrix                            *}
{* ---------------------------                             *}
{* This procedure is used to print out the branch coverage*}
{* matrix.                                                 *}
{*                                                         *}
{***********************************************************}

procedure print_proof_matrix(n : integer;maxcols : integer);
var
  i,j : integer;
begin
  clearscreen;
  write(' ');
  for i := 1 to (((maxcols + 1)* 4) + 4) do
    write('_');
  write(' ');
  writeln;

  write('|');
  for i := 1 to (((maxcols + 1)* 4) + 4) do
    write(' ');
  writeln('|');

  write('|');
  write(' ':4);
  for i := 1 to maxcols do
   begin
    if (i = 1) then
      write('EVC':4)
    else
     if (i = maxcols) then
```

```
        write('RDS':4)
      else
       write((i-1):4);
     end;
    write('    |');
    writeln;

    write('|');
    for i := 1 to (((maxcols + 1)* 4) + 4) do
      write('_');
    write('|');
    writeln;

    write('|');
    for i := 1 to (((maxcols + 1)* 4) + 4) do
      write(' ');
    writeln('|');

    for i := 1 to n do
    begin
      write('|');
      write(i:2,'|':2);
      for j := 1 to maxcols do
        write(proofmatrix[i,j]:4);
      writeln('    |');
    end;

    write('|');
    for i := 1 to (((maxcols + 1)* 4) + 4) do
      write(' ');
    writeln('|');

    write('\');
    for i := 1 to (((maxcols + 1)* 4) + 4) do
      write('-');
    write('''');
    writeln;
    writeln;
    writeln(' ********   COVERAGE MATRIX   *******');
  end;

{***********************************************************}
{*                                                         *}
{* procedure print_child_matrix                            *}
{* --------------------------                              *}
{* This procedure is used to print out the child matrix.   *}
{*                                                         *}
{***********************************************************}

procedure print_child_matrix(n : integer);
var
  i,j : integer;
  children : integer;
begin
```

```
    clearscreen;
    write(' ');
    for i := 1 to (((n + 1)* 3) + 6) do
      write('_');
    write(' ');
    writeln;

    write('|');
    for i := 1 to (((n + 1)* 3) + 6) do
      write(' ');
    writeln('|');

    write('|');
    write(' ':4);
    for i := 1 to n + 1 do
      write(i:3);
    write('  |');
    writeln;

    write('|');
    for i := 1 to (((n + 1)* 3) + 6) do
      write('_');
    write('|');
    writeln;

    write('|');
    for i := 1 to (((n + 1)* 3) + 6) do
      write(' ');
    writeln('|');

    for i := 1 to n do
    begin
      write('|');
      write(i:2,' ':2);
      children := matrix[i,n+1] +1;
      for j := 1 to children  do
        write(childmatrix[i,j]:3);
      writeln('     |');
    end;

    write('|');
    for i := 1 to (((n + 1)* 3) + 6) do
      write(' ');
    writeln('|');

    write('\');
    for i := 1 to (((n + 1)* 3) + 6) do
      write('-');
    write('''');
    writeln;
    writeln;
    writeln('***** CHILD MATRIX *****');
  end;
```

```
{*******************************************************}
{*                                                     *}
{* procedure readin_child_matrix                       *}
{* ---------------------------                         *}
{* This procedure is used to generate the child matrix *}
{* based upon the adjacency matrix supplied by the user. *}
{*                                                     *}
{*******************************************************}

procedure readin_child_matrix;
{this procedure makes a copy of the adjacency matrix and is}
{used in the generation of basic paths}
var
  outfile : text;
  i,j,checksum : integer;
  k :integer;

begin
  assign(outfile,'child.dat');
  rewrite(outfile);
  k := 1;
  for i := 1 to n do
  begin
    k := 1;
    for j := 1 to n do
      begin
     if (matrix[i,j] = 1) then
        begin
           childmatrix[i,k] := j;
           inc(k);
        end
      end;
  end;
  for i := 1 to n do
   begin
     if (i = uxn) then
       write(outfile,0)
     else
     write(outfile,matrix[i,n+1]+1);
     write(outfile,' ');
    for j := 1 to matrix[i,n+1]+1 do
      begin
     if (i = uxn) then
       write(outfile,'0')
     else
       begin
         write(outfile,childmatrix[i,j]-1);
         write(outfile,' ');
       end;
      end;
    writeln(outfile);
   end;
close(outfile);
end;
```

```
{**********************************************************}
{*                                                        *}
{* main program                                           *}
{* ------------                                           *}
{* This is the main program.  It calls all the other      *}
{* procedures.                                            *}
{*                                                        *}
{**********************************************************}

begin
  initialize;
  read_matrix_values;
  writeln;
  {read_incidence_matrix;
  writeln;}
  print_matrix(n);
  pause;
  print_entryexit_nodes(n);
  pause;
  {print_incidence_matrix(n,totedges);
  pause;}
  print_proof_matrix(n,maxcols);
  pause;
  readin_child_matrix;
  {print_child_matrix(n);}
end.
```

APPENDIX D

ADAPTIVE TESTING STRATEGY PROGRAM LISTING

```
/*******************************************************/
/*                                                     */
/*              Adaptive Testing Strategy              */
/*                                                     */
/*******************************************************/
/*                                                     */
/*              Author:  Shankar Narayanaswamy         */
/*              Date:    05/16/91                      */
/*              Class:   COMSC 5000 - Thesis           */
/*              Adviser: Dr. Mansur Samadzadeh         */
/*                                                     */
/*******************************************************/
/*                                                     */
/* Procedures Used:                                    */
/* ----------------                                    */
/* main, process, print, start_another_recursion,      */
/* insertchar.                                         */
/*                                                     */
/* Input for Program:                                  */
/* ------------------                                  */
/* Child matrix for the Control Flow Graph.            */
/* (Each row in the child matrix consists of the number  */
/*  of children each node possesses followed by the node */
/*  labels of the children).                           */
/*                                                     */
/* Output of Program:                                  */
/* ------------------                                  */
/* Prints out the various paths generated according to   */
/* the adaptive testing strategy.                      */
/*                                                     */
/* Program Function                                    */
/* ----------------                                    */
/* The program accepts input in the format specified above*/
/* Given the input in this format the program generates  */
/* the paths in accordance with the Adaptive Testing   */
/* Strategy.                                           */
/*                                                     */
/* Debugging Tools Used:                               */
/* ---------------------                               */
/* Turbo C Debugger                                    */
/*                                                     */
/*******************************************************/


#include <stdio.h>
#include <conio.h>

/* global declarations */

int  Visit [25] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
              0, 0, 0, 0, 0, 0, 0, 0, 0,
              0, 0, 0, 0, 0, };
int  Par_child [20][12];  /* array to store parents and */
                          /* their respective children  */
```

```
int  destination = 0;     /* keeps track of destination */
int  Note_Node = -1, Note_i = -1, check = 0;
int  marknode = 0;  /* node marked or not ? */

/* arrays used to save paths for printing purposes */
int  print_nodes [20], prev_print_nodes[20];

int  nofprint_nodes = 0, prev_printnodes = 0;
int  numberof_common_nodes = 0;


/************************************************************/
/*                                                          */
/* main                                                     */
/* ----                                                     */
/* This is the main program.  It calls all the other        */
/* routines whenever required.                              */
/*                                                          */
/*                                                          */
/*                                                          */
/************************************************************/

main (argc, argv)
int   argc;
char  *argv[];
{
FILE  *fp;
int    number_of_children, i = 0, j = 0;

fp = fopen (argv[1], "r");

clrscr ();

/* read in input from designated file */
while (fscanf (fp, "%d", &number_of_children) != EOF) {
     Par_child [i][j] = number_of_children;
     for (j = 1; j <= number_of_children; j++)
          fscanf (fp, "%d", &Par_child[i][j]);
     j = 0; i++;
     }
destination = i-2;
process (0);

if (marknode != destination) {
 for (i = 1; i <= Par_child[marknode][0]; i++)
  if (Par_child[marknode][i] < marknode) {
   print_nodes[nofprint_nodes++]= Par_child[marknode][i]+1;
     }
     process (marknode);
     }
}
```

```
/***********************************************************/
/*                                                         */
/* process                                                 */
/* -------                                                 */
/* This procedure is used to process the various nodes in */
/* the graph.  This processing is done recursively.  Other*/
/* procedures are called at appropriate locations.         */
/*                                                         */
/***********************************************************/

process (int node) {
     int i = 0, k = 0;
     if (node == destination || check)
          print (node);

     if (node == destination) return(0);


     for (i = 1; i <= Par_child[node][0]; i++) {
      if (Par_child[node][i] < node)
       for(k = 1;k <= Par_child[Par_child[node][i]][0];k++)
               if (Par_child[Par_child[node][i]][k] == node)
{
                    check++;
                    if (check == 2) {
                         check = 0;
                         start_another_recursion (node);
                         return (0);
                         }
                    print (node);
                    return (0);
                    }

          print (node);
          process (Par_child [node][i]);
          }
     }


/***********************************************************/
/*                                                         */
/* print                                                   */
/* -----                                                   */
/* This procedure is used to print out the test paths as  */
/* generated by the application of the adaptive testing   */
/* strategy.  It does so by making insertions into two    */
/* arrays which are meant to be used solely for this.     */
/*                                                         */
/***********************************************************/

print (int node) {
     int i;

     marknode = node;
/*   printf ("%-3d", node+1); */
     print_nodes [nofprint_nodes++] = node+1;
     if (node == destination) {
          printf ("\n");

          if (prev_printnodes != 0) {
            i = 0;
            while (print_nodes[i] != prev_print_nodes[i]) {
            insertchar(prev_print_nodes[i], print_nodes, i);
                    i++;
                    }
               }
          for (i = 0;i < nofprint_nodes; i++) {
               prev_print_nodes[i] = print_nodes[i];
               printf ("%-3d", print_nodes[i]);
               }

          prev_printnodes = i;
          nofprint_nodes = 0;
          }
     }


/***********************************************************/
/*                                                         */
/* start_another_recursion                                 */
/* -----------------------                                 */
/* This procedure is used to start another recursion from */
/* the point at which it is called during the execution   */
/* of the adaptive testing strategy.                       */
/*                                                         */
/***********************************************************/

start_another_recursion (int node) {
 int  k;
 for (k = 1; k <= Par_child[node][0]; k++)
     if (Par_child[node][k] > node) {
          process (Par_child [node][k]);
          }
 }


/***********************************************************/
/*                                                         */
/* insertchar                                              */
/* ----------                                              */
/* This procedure is used to insert node labels during the*/
/* process of printing out the test paths generated, into */
/* the array used for this purpose.  It inserts labels    */
/* into the front end of the array by shifting the        */
/* previous contents of the array to the right.           */
/*                                                         */
/***********************************************************/
```

```
insertchar (ch, aray, pos)
int    ch, *aray,  pos;
{
int i = nofprint_nodes;
do {
   *(aray + i) = *(aray + i - 1);
   i--;
   } while (i != pos);
nofprint_nodes++;
*(aray + pos) = ch;
}
```

APPENDIX E

USER MANUAL

USER MANUAL

Part 1: Complexity Measure Program

1. At the C:\> prompt type

    complexy <Enter>

2. The program will print the following query on the screen.

    Enter adjacency matrix file name:

    Respond with <adjacency matrix filename> <Enter>

3. The following query will then appear on the screen.

    Enter # of nodes in control flow graph:

    Respond with <# of nodes in control flow graph> <Enter>

The program will display the adjacency matrix, the value of C(G), the unique entry node and exit node, and the branch coverage matrix on the screen, in that order. It will also create a file called child.dat which is used by the adaptive testing program.

Part 2: Adaptive Testing Program

At the C:\> prompt type "testing <child.dat> <Enter>".

The file child.dat used here is the one that was created by the complexity measure program in Part 1 above. The program will display the list of test paths on the screen.

2

VITA

Shankar Narayanaswamy

Candidate for the Degree of

Master of Science

Thesis: A GRAPH-THEORETIC MATRIX-BASED APPROACH FOR A MEASURE OF PROGRAM TESTING AND AN ADAPTIVE TESTING STRATEGY

Major Field: Computer Science

Biographical:

Personal Data: Born in Bangalore, Karnataka, India, June 16, 1962, the son of P. Narayana Swamy and Savithri.

Education: Graduated from National College, Bangalore, India, in May 1980; received Bachelor of Engineering in Electronics Engineering from Bangalore University at Bangalore in December 1984; completed requirements for the Master of Science degree at Oklahoma State University in July 1991.

Professional Experience: Teaching Assistant, Department of Electrical Engineering, University Visvesvaraya College of Engineering, Bangalore University at Bangalore, May 1985 to July 1986. Programmer, Oklahoma Resources Integrated General Information Network Systems, Oklahoma State University, August 1989 to January 1991