

A MEASURE OF SOFTWARE COMPLEXITY

by

LLOYD DAVID BORCHERT

B.S., Oglethorpe University, Atlanta, Georgia, 1972

---

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

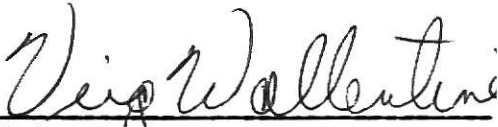
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY  
Manhattan, Kansas

1981

Approved by:

  
Major Professor

SPEC  
COLL  
LD  
2668  
R4  
1981  
B67  
c.2

A11200 068994

## TABLE OF CONTENTS

Chapter 1. Introduction	1
A. Overview of Report	1
B. Importance of complexity measurement	1
C. Definition of program complexity	2
D. Characteristics of measurement criteria	3
Chapter 2. Some current methods for measuring program complexity	5
A. McCabe (Graph theory cyclomatic number)	5
B. Chapin (Set theory index of module complexity)	5
C. McClure (Control variable and module complexity)	12
D. Halstead (Measure of program effort)	24
Chapter 3. Basis of proposed measure	29
A. Characteristics measured	29
B. Computation of measure	31
C. Example of computation	31
Chapter 4. Other examples and comparative analyses	33
Chapter 5. Conclusions	39
Chapter 6. Future work	40
Appendix A	41
References.	42

**THIS BOOK  
CONTAINS  
NUMEROUS PAGES  
WITH DIAGRAMS  
THAT ARE CROOKED  
COMPARED TO THE  
REST OF THE  
INFORMATION ON  
THE PAGE.**

**THIS IS AS  
RECEIVED FROM  
CUSTOMER.**

## Chapter 1

### Introduction

#### A. Overview of Report.

This report describes several complexity measures and illustrates how each can be used to quantify the complexity of a sample program. The purpose of the report is to present several different and unique approaches to program complexity measurement, each using a different program characteristic, and then to present an alternative approach to complexity measurement. The report is structured to first give the reader an introduction to program complexity, complexity measurement, and the types of characteristics used in measuring program complexity. Next, the reader is presented several current methods for measuring program complexity with the basis of each measure, the method of computation, and an example of computation, each using the same sample program. Then, a new complexity metric is presented with the basis of the proposed measure, the computation of the measure, and the example of computation. This is followed by additional examples, a comparative analysis of all the metrics presented, some conclusions, and the future work that is required on this proposal.

#### B. Importance of complexity measurement.

Anyone who has studied software engineering has had the

opportunity to recognize that there are many different ways to design and code a program that will accomplish the same basic function. The programming style used by most programmers is individualistic and generally based upon their own experiences and education. It is these diverse approaches to design and programming style that give us our traditional problems in developing, testing, and maintaining computer software. As demonstrated by many authors on this subject, often over half of the development time is spent in testing. Additionally, the total cost of a large software system is dominated by program maintenance. The increased cost of software development and software maintenance and the decreased cost of hardware has caused a shift in emphasis from concerns of machine efficiency to the production of clear and structured programs that can be easily maintained.

#### C. Definition of Program Complexity

Program complexity is an objective measure of how simple or how difficult a program is to understand when the coded instructions for that program are examined by an individual who is proficient in the language in which the coded instructions are written. Although related, the measure of program complexity is separate from the inherent complexity of the total problem. The inherent complexity of a problem is determined by the nature of the problem. Program complexity is a function of the process used to solve the problem. In the past, individuals have generally relied on intuition and common sense to measure what appeared to be simple and what appeared

to be complex; however, program complexity must be independent of individual style and based upon measurement methods which will always perform satisfactory, which have no built-in prejudices, which can be assigned quantitative values, and which are convenient to use.

#### D. Characteristics of measurement criteria

Software that is functionally equivalent, that is different programs that accomplish the same functions, can have many major differences. These differences are derived from the software characteristics of each program. It is the quantitative measurement of these characteristics and their comparison that will make it possible to determine which programming techniques produce software that will reduce the current problems of understandability, maintainability, and reliability. It is important to make a distinction between the types of characteristics used in measuring complexity. Subjective characteristics such as testable, understandable, and maintainable are difficult to standardize and are dependent on programming language. Zolnowski [1] identifies four main categories of measurable characteristics:

##### GENERAL TYPES OF VARIABLES

CATEGORY	MEASURED
1. Instruction mix	-Program size data -Numbers and types of instructions -Specific attributes of instructions
2. Data reference	-Numbers and types of variables -Number of references to and span of each

- variable
- Parameter nesting data
- 3. Interaction/
  - Interconnection
    - Numbers and types of subprograms referenced
    - Nesting levels of subprograms
    - Number of entry points
    - Nesting levels of parameters
    - Types and numbers of parameters
    - How data is connected between modules
- 4. Structured/
  - Control flow
    - Counts attributes of branching instructions
    - Flow graph of program
    - Basic flow graph variables such as numbers and sizes of basic blocks and intervals
    - Detailed loop analysis
    - Detailed strongly connected region data

There have been complexity measuring techniques advanced for all of these characteristics and others not mentioned here. The most popular measurement techniques are based on structured characteristics such as the number of independent circuits in a program, the number of source statements in a program, the number of decision nodes or the number of operands in a program. Structured characteristics are used because they are more measurable, less dependent on programming language, and easier to use in the development of a complexity measure.

## Chapter 2

### Some Current Methods for Measuring Program Complexity

#### A. McCabe (Graph theory cyclomatic number)

##### BASIS OF MEASURE

M McCabe's complexity metric [2] is based on the decision structure of a program and the number of linearly independent control paths comprising the program. McCabe's metric counts the number of basic control path segments which, when combined, will generate every possible path through the program; he presented this as a measure of program complexity.

##### COMPUTATION OF THE MEASURE

M McCabe's  $V(g)$  metric, the cyclomatic number is defined as:

$$V(g) = \text{NUMBER OF EDGES} - \text{NUMBER OF NODES} + 2(\text{NUMBER OF CONNECTED COMPONENTS}).$$
 McCabe presents two simpler methods calculating  $V(g)$ . McCabe's  $V(g)$  can be computed as the number of predicate nodes plus 1, or as the number of regions in a planar graph of the control flow.

##### EXAMPLE OF COMPUTATION

Using the sample procedure in appendix A and its associated directed graph (figure 1)  $V(g)$  is computed as follows:

NUMBER OF EDGES -  $E = 11$

NUMBER OF NODES -  $N = 9$

NUMBER OF CONNECTED COMPONENTS -  $P = 1$

$V(g) = E - N + 2*P = 4$

#### B. Chapin (Set theory index of module complexity)



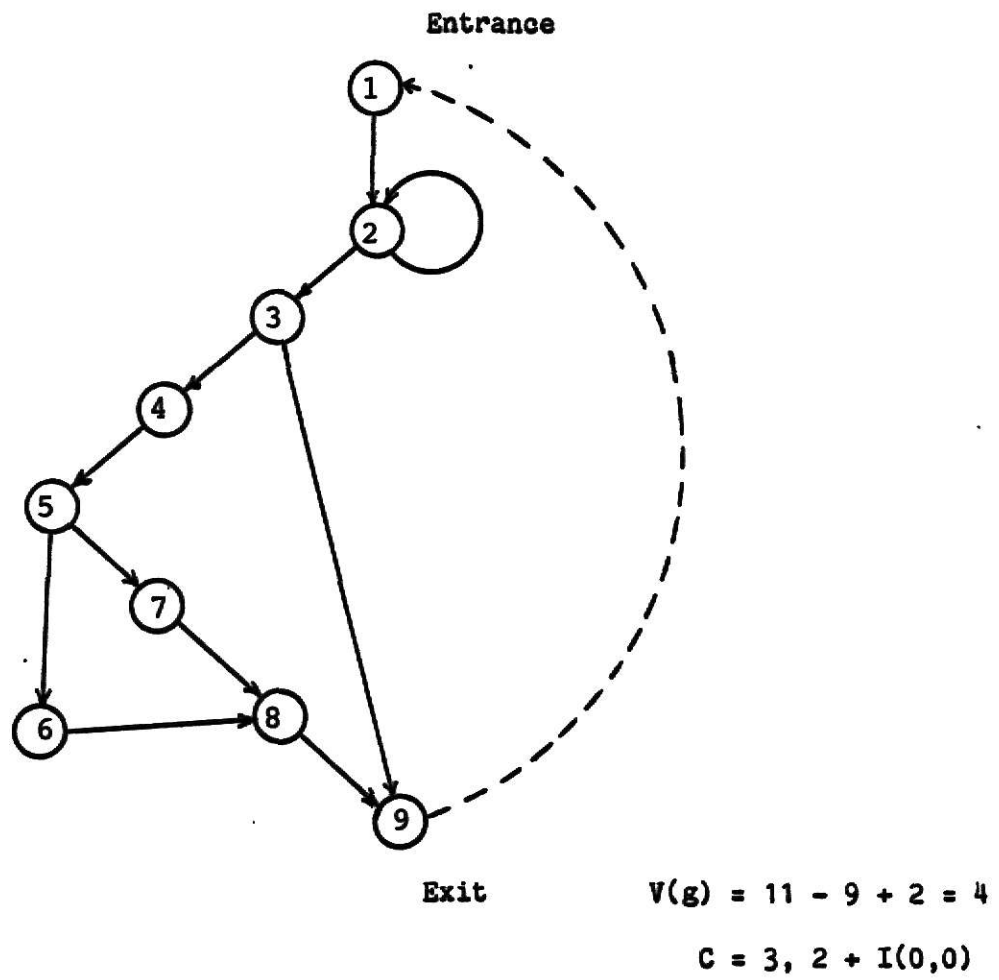


Figure 1

## **BASIS OF MEASURE**

Chapin's measure of software complexity [3], which he labels 'Q', is a quantitative value for the difficulty people have in understanding the function of the software. His measure is quantitative, it shows reasonable validity, and it is easily computed when his system of program documentation is used.

Chapin's basis for his Q measure springs from the set theoretic definition of a function. His function is a correlation between different sets of input data of a specified domain and different sets of output data of a specified range. He expresses a difference in functions as a difference in the input or in the output or both.

There are two forms of differences - one difference is in the membership in the sets and the other difference is in the domains and ranges. Since a change in domain of the input data or in the range of the output data is usually of smaller impact on the function of a program than the actual changes in the input or output, his measure of software complexity ignores domains and ranges and concentrates on the input and output.

Chapin further classifies data by the role that the data plays in the program. Input data that is needed for processing and the production of output data is called 'P' data. The data that is created, or modified in value or identity by the execution of the program function is called 'M' data. The data used to control which functions are executed is called 'C' data. The data which passes through a function unchanged in value or identity, that is just passed from one part of the software to another, is called 'T' data.

The control data contributes the most to program complexity. The data which plays a modification role is also a major contributor to program complexity. The data which plays a processing role contribute some complexity. The data that only passes through a function contributes the least to software complexity.

The modules of the software may communicate data among themselves; data may start as 'M' data in one module, becoming 'T' data as they are communicated through other modules, and then terminating as 'C' or 'P' data in a using module. The more modules that are involved in communicating any item of data, the higher the complexity of the software.

The communication of data among modules can be further complicated by the presence of iteration. Iteration control is the most difficult aspect of complexity in modularized programs and systems. While modularization can reduce the total complexity it will not reduce the complexity of iteration when more than one module is involved. If the extent of iteration is determined by control data which comes from modules other than the loop exit module, then the complexity of the software increases, but not in a linear manner.

#### COMPUTATION OF THE MEASURE

Chapin states that the high reliability of his measure comes from the simple computational procedure used on the documentation for the program or system. A measure is reliable when different people consistently come to the same result when they use the same computational procedure.

The ten steps in the computational procedure for the measure 'Q' are:

1. For each module, count the number of data items shown in 'C', 'P', or 'T' roles as input, and in 'M' or 'T' roles as output. When one data item appears in multiple roles or has multiple sources or destinations, each is to be counted. Data serving as program wide or system wide constants or literals are not counted.
2. Multiply for each module the total count for each role by the appropriate weighting factor 'W', as follows: 3 for 'C', 2 for 'M', 1 for 'P', and .5 for 'T'.
3. Sum the weighted counts by module.
4. Assign an initial 'E' value of 0 to all modules. Then examine the documentation to determine which modules are to contain the exit tests for iteration where subordinate modules are part of the iteratively invoked loop body. The loops or iterations are ignored when they are to be performed entirely within a module with no subordinate modules iteratively invoked.
5. For each iteration-exit module identified in step 4, examine the 'C' items to determine which are to serve in the exit test for the iteration of the subordinate modules that comprise the loop body. Determine where these 'C' data come from. If they come from within the subordinate loop body, add 1 to 'E' for each such 'C' data item. If they come from outside of the loop body, add 2 to 'E' for each such 'C' data item.
6. Convert 'E' for each module into a repetition factor 'R' by adding 1 to the square of one-third of 'E'.

7. Multiply the sum of the weighted counts from step 3 by the module's respective 'R' values.
8. Find the square root of the products from step 7. This is 'Q', the index of module complexity. The computation in this step is a computation of the geometric mean of the total weighted counts and the inter-module iteration control.
9. Calculate the 'Q' of the program by finding the arithmetic mean (average) of the component modules.
10. Calculate the 'Q' of the system by finding the arithmetic mean of the component modules within the component programs, or by weighting the programs 'Q' from step 9 by the relative sizes (in terms of modules) of the programs.

The practical lower bound on module complexity is 1.0. No upper bound exists for 'Q', but values beyond 11.0 are uncommon with structured programming and structured design.

#### EXAMPLE OF COMPUTATION

Using the data in the sample procedure in appendix A and constructing the input-output table as shown in figure 2 the index of module complexity 'Q' would be computed as follows:

Step 1 Raw-counts  $C = 5$ ,  $P = 3$ ,  $M = 5$ ,  $T = 3$

Step 2 Weighted-counts  $3 * C = 15$ ,  $1 * P = 3$ ,  $2 * M = 10$

$.5 * T = 1.5$

Step 3 Weighted-total  $15 + 3 + 10 + 1.5 = 29.5$

Step 4  $E = 0$  (no subordinate modules)

Step 5 Not applicable in this case

Step 6  $R = 1 + (.333 * E) Sq = 1 + 0 = 1$

**Input-Output Table**

<b>Input</b>	<b>Type</b>	<b>From</b>	<b>Output</b>	<b>Type</b>	<b>To</b>
<b>Procedure Test</b>					
<b>Check</b>	<b>CPT</b>	<b>Current-node</b>	<b>Check</b>	<b>M</b>	<b>Proc-Test</b>
<b>I</b>	<b>C</b>	<b>Proc-Test</b>	<b>none</b>	<b>M</b>	<b>Proc-Test</b>
<b>Fair</b>	<b>C</b>	<b>Proc-Test</b>	<b>none</b>	<b>M</b>	<b>Proc-Test</b>
<b>A</b>	<b>CPT</b>	<b>Proc-Test</b>	<b>List-Pointer</b>	<b>M</b>	<b>Proc-Test</b>
<b>Last</b>	<b>CPT</b>	<b>Proc-Test</b>	<b>List-Pointer</b>	<b>M</b>	<b>Proc-Test</b>

**Figure 2**

Step 7 Prod =  $29.5 * 1 = 29.5$

Step 8 SqRt 29.5 = 5.431

### C. McClure (Control variable and module complexity)

#### BASIS OF MEASURE

McClure [4], [5] states that program complexity is, "a function of the number of possible execution paths in the program and the difficulty of determining the path for an arbitrary set of input data." Thus, a program in which there are no alternative paths has no complexity.

McClure presents a method of quantitatively measuring the complexity of interaction between modules. It includes two measures; the measure of control variable complexity and module complexity.

Control variable complexity is a measure of the effect a control variable has on the interaction between modules. It quantifies the difficulty of understanding how a control variable will affect the module and how that control variable is used in the interaction between modules.

The complexity of a control variable is a function of the set of all the modules which use the control variable and the calling relationship among these modules. The more modules which use the same control variable the more difficult it is to understand the interactions of each module using the control variable.

A module uses a control variable if the value of the control variable is modified and/or referenced in the module. Let the set of modules in a program that uses control variable V be denoted by AV.

Then, AV can be divided into two disjoint sets:

1. The set of modules in which the value of control variable V is modified, denoted by MV.
2. The set of modules in which the value of control variable V is referenced but not modified, denoted by RV.

Another set of modules which must be considered when measuring the complexity of a control variable is the set of modules whose invocation is dependent upon the value of the control variable, denoted by EV. Modules in the set EV may or may not use control variable V.

The complexity of control variable V is determined by looking at the membership of the sets AV, RV, MV, and EV. The complexity of control variable V is divided into two factors; The degree of ownership for the control variable V and the complexity of the interaction of the control variable V among the members of the sets MV, RV, and EV.

The degree of ownership for a control variable is a measure of the complexity of the relationship between the module that owns the control variable and the modules that modify the value of this control variable.

It is defined to be one of the following four cases:

- DEGREE 1: The control variable is modified exclusively by its owner or never modified in the program.
- DEGREE 2: The control variable is modified by its owner and at least one descendant of its owner.
- DEGREE 3: The control variable is only referenced by its owner and modified by at least one descendant of its owner.



DEGREE 4: The control variable is not referenced by its owner and is modified by at least one descendant of its owner.

As the degree of a control variable increases, its use in the program becomes more difficult to understand.

Module complexity is a measure of the invocation of one module relative to the other modules in the program. It evaluates the difficulty of determining how program control is passed among each module. Module complexity measures the difficulty of understanding how program control is passed among modules in a structured program.

The analysis necessary to calculate module complexity includes an evaluation of the control structures and the complexity of the control variables referenced in the calling of a module or a branch to a routine which will terminate the program. The four possible sources for module complexity in a structured program are:

1. The first source for module complexity is the complexity of the control variables upon whose values module calling is dependent.
2. The second source for module complexity is the control structures that direct module calling. Module complexity increases if a module is called within a iteration structure. In the case of iteration the value of the control variables used to direct iterative module calling may be modified by the module that is being called. This example of iteration will increase the module complexity value since the called module, as well as the calling module, must be examined to understand loop termination

conditions.

3. The third source for module complexity is when a module has multiple calling modules. It is more difficult to understand in this case when and how the module is invoked. Also, when a change is made to a common module, the possible effect of this change on each calling module must be carefully examined.
4. The fourth source for module complexity is when a branch from the module is made to a routine which aborts the program. In this case, the calling-called invocation relationship is overridden.

Module complexity then, depends upon the complexity of the control variables used to invoke a module and select the abort branch, the control structures used for calling modules and the number of other modules to which a module may branch.

#### COMPUTATION OF MEASURE

The formula for calculating the control variable complexity, denoted by  $C(v)$ , is used to calculate the complexity value of control variable  $V$  in a well-structured program.  $C(v)$  is defined as follows:

$$C(v) = (Dv * Iv) / N$$

Where  $N$  is the number of different modules in the program,  $Dv$  is the degree of ownership for  $V$ , and  $Iv$  is the interaction complexity for  $V$ .

$Dv$  is assigned an integral value of 1 to 4 according to the following:

$Dv = 1$  if control variable  $V$  has degree 1

Dv = 2 if control variable V has degree 2

Dv = 3 if control variable V has degree 3

Dv = 4 if control variable V has degree 4

Interaction complexity, denoted by Iv, indicates the complexity of the interaction between modules using control variable V. The formula for calculating Iv is:

$$Iv = IM1 + IM2 + IM3 + IM4$$

IM1 measures the complexity of interaction between modules which use explicitly the control variable V. To calculate the value of IM1, perform the following steps:

1. Determine the members of the set Mv (The set of modules in which the value of control variable V is modified).
2. Determine the members of the set EV (The set of modules whose calling is dependent upon the value of the control variable V).
3. Count the members in the set intersection of MV and EV and assign this value to IM1.

IM1 = number of modules in which the value of the control variable is both modified and whose calling is dependent upon the value of the control variable.

IM2, IM3, and IM4 measure the complexity of interaction between modules which have implicit use of the control variable V. To calculate IM2, perform the following steps:

1. Determine the members of the set RV (The set of modules where the control variable is referenced but not modified).
2. Determine the members of the set MV (The set of modules in which the value of V is modified).

3. Determine the relationship (i.e., calling-called or non-related) of each member of RV paired with each member of MV.
4. Add one to IM2 for each calling-called pair in which the calling module is a member of RV and the called module is a member of MV.

IM2 = number of calling-called modules pairs that use control variable V such that the calling module of the pair is a member of the set of modules that references V and the called module of the pair is a member of the set of modules that modifies V.

To calculate IM3 perform the following steps:

1. Determine the members of the set AV (The set of modules in a program that use control variable V).
2. Determine the members of the set MV (The set of modules which modify V).
3. List all pairs of modules in AV that are not related.
4. For each pair listed in step 3 such that one module is the pair is a member of MV and would be above the other in a structure chart, add 1 to IM3.

IM3 = number of module pairs that use control variable V such that one module of the pair is a member of the set of modules that modify V and is above the other module of the pair in the structure chart.

IM4 measures how intermittently control variable V is used in the program. To calculate IM4 perform the following steps:

1. Determine the members of the set AV (The set of modules that use V).
2. For each module in AV, other than the owner of control

variable V, determine if all its calling modules are members of AV. If not, add 1 to IM<sup>4</sup>.

IM<sup>4</sup> = number of modules which use the control variable V which are called by modules which do not use the control variable.

The control variables of a program can be compared by means of their complexity values. Control variables with greater complexity values contribute more complexity to the design than control variables with lesser complexity values. When computed the complexity value of a control variable must be in the range of 0 - 8.

Module complexity is a measure of the difficulty of understanding how the control in a program is passed from one module to another in a well-structured program. It is the sum of three complexity terms:

T1 = The complexity of passing control to module M.

T2 = The complexity of passing control from module M to other modules that are called by module M.

T3 = The complexity of branching from module M to a routine that will terminate the program.

M(m) denotes the complexity of module m and is defined as:

$$M(m) = (T1 + T2 + T3)$$

Each of the three complexity terms has two factors:

1. The number of modules involved in either a calling or a called role.
2. The complexity of the control variables referenced and whether iteration structures are used or not.

For example, the complexity of passing control to module m depends upon the number of modules from which m is called, if iteration structures are used, and the complexity of the control variables

whose values are referenced to direct the calling of  $m$ . The three module complexity terms are defined as:

$$T1 = Fm * Xm$$

Where  $Fm$  is the number of calling modules of module  $m$  and  $Xm$  is the complexity of the control variables and the iteration structures referenced in calling module  $m$ .

$$T2 = Sm * Ym$$

Where  $Sm$  is the number of called modules of module  $m$  and  $Ym$  is the complexity of the control variables and the iteration structures referenced by module  $m$  in calling the called modules.

$$T3 = Bm * Zm$$

Where  $Bm$  is the number of routines which will terminate the program to which module  $m$  may branch and  $Zm$  is the complexity of the control variable referenced by module  $m$  to direct branches from  $m$  to these routines.

The following two definitions explain how to determine the values of  $Xm$ ,  $Ym$ , and  $Zm$ , which are the complexities of the control variables in the three module complexity terms  $T1$ ,  $T2$ , and  $T3$ :

1. A calling control variable set is the set of control variables upon whose values a particular calling of a module depends.
2. A branch control variable set is the set of control variables upon whose values a branch may be made to a routine that will terminate the program.

The complexity contribution of control variables in each module complexity term is calculated by finding the mean (average) complexity value of the appropriate control variable sets.  $Xm$ ,  $Ym$ ,

and  $Z_m$  are all calculated in a similar manner.  $X_m$  is calculated in three steps:

1. Determine all the calling control variable sets referenced by the calling modules of module  $m$  in the calling of module  $m$ . If module  $m$  is the root module or if module  $m$  is always unconditionally invoked by the calling module(s), set  $X_m$  to 0; otherwise perform steps 2 and 3.
2. For each such calling control variable set, total the control variable complexity value  $C(v)$  of each control variable in the set. If this module calling occurs within a iteration structure, multiply this total by 2.
3. Next, sum the complexity values of all the calling control variable sets calculated in step 2 above. Find the average of this sum by dividing it by the number of calling control variable sets used in the calling of module  $m$ .

To calculate  $Y_m$ , perform the following three steps:

1. Determine the calling control variable sets used by module  $m$  to invoke its called modules. If there are no such sets, either because  $m$  has no called modules or because all the invocations within  $m$  are unconditional, set  $Y_m$  to 0; otherwise, perform steps 2 and 3 below.
2. For each such calling control variable set, sum the control variable complexity value of each control variable in the set. If any control variable in the calling control variable set is used within a iteration structure, multiply this sum by 2.
3. Next, sum the complexity values for all the calling control

variable sets calculated in step 2 above. Find the average of this sum by dividing by the number of calling control variable sets that are used in module  $m$  to invoke the called modules.

$Z_m$  is calculated in the following manner:

1. Determine all the branch calling control variable sets that module  $m$  references to direct branches out of this module to a routine that terminates the program. If there are no such sets, set  $Z_m$  to 0; otherwise perform steps 2 and 3 below.
2. For each such branch control variable set, sum the complexity value of each control variable in this set.
3. Next, sum the complexity values for all the branch control variable sets calculated in step 2 above. Take the average of this sum by dividing it by the number of branch control variable sets that are used in module  $m$ .

The module complexity formula, denoted by  $M(m)$ , for module  $m$  in a well-structured program is defined as follows:

$$\begin{aligned} M(m) &= (T_1 + T_2 + T_3) \\ &= (F_m * X_m) + (S_m * Y_m) + (B_m * Z_m) \end{aligned}$$

If module  $m$  is the root module,  $T_1$  will have a value of zero; if module  $m$  has no called modules,  $T_2$  will have a value of zero; and, if module  $m$  contains no branches out of the module to a termination routine,  $T_3$  will have a value of zero.

The range of values for  $M(m)$  is  $0 - 16 * S * N$ , where  $S$  is the total number of control variables and  $N$  is the number of unique modules defined in the design. The greater the value of  $M(m)$ , the



more difficult it is to understand how program control is passed to and from module m. For example, a module that is unconditionally invoked, has no called modules, and does not branch to a termination routine has zero module complexity.

Module complexity can be divided into four cases based upon module type as indicated below:

Case 1: A one-module program

Case 2: A root module that calls other modules

Case 3: A module that calls no other modules

Case 4: A module that has called modules

Module complexity, just as control variable complexity, can be used as a criterion in comparing programs. Given a set of programs that perform the same task, the program selected as best may be the program with the lowest program complexity value, where the program complexity value is the sum of the module complexity values of each unique program module.

#### EXAMPLE OF COMPUTATION

Using the example in appendix A and construction procedure test as shown in figure 3, the complexity of procedure test is computed as follows:

Control Variable Complexity ( C(v) )

$$C(v) = (Dv * Iv) / N$$

$$Iv = IM1 + IM2 + IM3 + IM4$$

Control Variable	IM1	IM2	IM3	IM4	Iv	Dv	N	C(v)
Check (I)	0	0	0	0	0	1	3	0
Fair	1	2	1	0	4	1	3	1
List-pointer	1	0	0	0	1	1	3	.25

```

Procedure Test
  Main Routine
    Begin
      Integer A, I
      Boolean Fair
      Fair := True
      I := 1
      Perform Module A
      Perform Module B
    End
  Stop
Module A
  While ((Check(I) not equal "P") and (Fair = True)) Do
    Begin
      Fair := Match(I)
      I := I + 1
    End
Module B
  If Fair = True then
    Begin
      A := allocate 1
      Perform Module C
      Last := A
      set LIST1(Last, null)
      set LISTA(Last, LIST2(Current-node + 1))
    End
Module C
  Begin
    If List-pointer = null then
      List-Pointer := A
    else set LIST1(Last, A)
  End

```

Figure 3

#### Module Complexity $M(m)$

$$M(m) = (T1 + T2 + T3)$$

$$M(m) = (Fm * Xm) + (Sm * Ym) + (Bm * Zm)$$

Module	$Fm * Xm + Sm * Ym + Bm * Zm = T1$						T2	T3	$M(m)$	
Main	0	0	2	0	0	0	0	0	0	0
A	1	1	0	1	0	0	1	0	0	1
B	1	1	1	1	0	0	1	1	0	2
C	2	.25	0	.25	0	0	.5	0	0	.5

Total program complexity =  $0 + 1 + 2 + .5 = 3.5$

#### D. HALSTEAD (MEASURE OF PROGRAM EFFORT)

##### BASIS OF MEASURE

According to Halstead [7], [8], [9] and Gordon [6] several metrics have been developed which attempt to provide a quantitative measurement of the different factors which contribute to the quality of software. One of the important goals in software research is to measure programming style. This is important because significant improvement in programming style will decrease the effort required to understand and maintain that software. Gordon states that, "To accurately assess programming style we must be able to accurately assess the amount of mental effort expended to understand the code."

All of Halstead's metrics are functions of the number and frequency of the operators and operands occurring in the program. The mental efforts calculated from these represents the amount of mental work required to understand the function of the program. This measure of clarity will not reflect the programmers fluency or

familiarity with the program problem area. However, Halstead and Gordon assume that the programmer is fluent in the programming language which is used. There are several factors which influence how easy or difficult a program is to understand. These factors may be categorized into three broad areas: programmer ability based on language fluency and experience; program form, based on commenting, the placement of declarations, indentation, and the assignment of variable names; and program structure which is based on executable statements, the complexity of the control flow graph, the depth of statement nesting, clustering of the data references, and the locality of operations. Gordon's measure of program clarity is a simple function of the program's structure.

#### COMPUTATION OF THE MEASURE

Halstead specifies the following easily obtained properties of a program:

$n1$  = The number of distinct operators

$n2$  = The number of distinct operands

$N1$  = The total number of operators

$N2$  = The total number of operands

The developed theories will relate these values to program properties such as vocabulary, length, implementation level, and volume. These properties are defined to develop an algorithm based on the four basic parameters specified above. The definition of program vocabulary  $n$  and the program length  $N$  is:

$n = n1 + n2$  The number of distinct operators plus the number of distinct operands.

$N = N1 + N2$  The total number of operators plus the total

number of operands.

The program volume  $V$  is defined in terms of program vocabulary and program length as the number of the total usages of operators and operands in the program times the number of bits which would be required to provide a unique designator for each of the  $n$  different items composing the program vocabulary. The program volume then has the units of bits.

$$V = N \log_2 n$$

The program length  $N$  may be closely approximated as a function of the number of unique operators and operands used in the program. The function is defined as:

$$\text{approx } N = n_1 (\log_2 n_1) + n_2 (\log_2 n_2)$$

Program level  $L$  is a measure of the conciseness of an implementation of an algorithm. The highest level at which an algorithm may be represented is in the form of a called program. The volume of this representation is called the potential volume. When the actual volume equals the potential volume,  $L = 1$ . When more lengthy representation are used, those involving several operators and the repeated use of operands, their level is less than one. Since the logical input and output parameters for a program are difficult to determine the following estimator for determining the value of the implementation level is used:

$$\text{approx } L = 2 * n_2 / n_1 * N_2$$

Gordon states that the difficulty of programming increases as the volume of the program increases and decreases as the program level increases. Program style and differences in program structure, change the measured level and length of the implementation in an

inconsistent manner. To minimize the influence of style on the measure of programming effort, the measure for program effort is defined by Gordon as:

$$E_p = \text{approx } N (\log_2 n) / \text{approx } L$$

#### EXAMPLE OF COMPUTATION

Using the example in appendix A and figure 4, the measure of program clarity is computed as follows:

$$n_1 = \text{Number of distinct operators} = 9$$

$$n_2 = \text{Number of distinct operands} = 14$$

$$N_1 = \text{Total number of operators} = 33$$

$$N_2 = \text{Total number of operands} = 28$$

$$n = \text{Vocabulary} = n_1 + n_2 = 23$$

$$N = \text{Length} = N_1 + N_2 = 61$$

$$V = \text{Volume} = N \log_2 n = 255.964$$

$$\text{approx } N = n_1 \log_2 n_1 + n_2 \log_2 n_2 = 81.833$$

$$\text{approx } L = \text{Level} = 2n_2 / n_1 N_2 = .1111$$

$$E_p = \text{Measure of program clarity} = \text{approx } N \log_2 n / \text{approx } L$$

$$E_p = 3332.247$$

Operators	Frequency	Operands	Frequency
Do-While	1	Check(I)	1
While	1	"p"	1
Not Equal	1	MATCH(I)	1
:= (assignment)	7	Fair	4
= (equality)	3	True	3
+ (plus)	2	A	3
;	15	allocate 1	1
If-then	2	List-Pointer	2
else	1	null	2
		set LIST1(Last,A)	2
n1 = 9	N1 = 33	I	3
		Last	4
		set LISTA(Last,LIST2	
		(Current-node + 1))	1
		<hr/> n2 = 14	<hr/> N2 = 28

Figure 4

## Chapter 3

### Basis of Proposed Measure

#### A. Characteristics Measured

The goal of this project is to identify and present a program complexity measure that has the following properties:

**Reliable** - The measure can always be expected to perform its intended functions satisfactorily and each time the metric is applied to the same program it will return the same results.

**Objective** - There is no prejudice or preconceived bias built in to the measure.

**Quantitative** - Characteristics that can be assigned quantitative values.

**Easy-to-use** - Fulfills its purpose without wasting the user's time and energy or degrading his morale.

**Language-Independent** - Can be consistently and purposefully applied to programs written in different programming languages.

In addition to these properties, a good complexity measure should be a predictor of error occurrences and possible maintenance costs.

My starting hypothesis was that understanding of a program depended upon the complexity of the algorithm and the clarity of the coding. Based upon previous work by other authors, outlined in Chapter 2, and upon other program complexity examples, I determined



that complexity was generally derived from three basic program properties:

1. Program control structure
2. Program control flow
3. Program data usage

When the different complexity measures, which are derived from the three basic program properties above, are evaluated against the identified complexity measure properties a clear division of the three basic program properties was found. Those complexity measures which were derived from the use of program control structure and program data usage I found to be difficult to use and difficult to understand. For that reason my efforts were limited to characteristics of program control flow.

Program control flow is characterized by measurements taken from a control flow graph of a program. A control flow graph is a directed graph with nodes corresponding to the straight-line section of code and arcs indicating the sequence of control. The graph is connected and unreachable nodes are ignored. The graph has a single entry node and a single exit node (e.g., figure 1).

The control flow graph is used to develop the program complexity measure. Three simple graph measures, which are sensitive to the configuration of the graph, are applied to the graph. The first measure is the count of the possible linear paths through the graph. I define a linear path through the graph to be any connected sequence of nodes, from the entrance node to the exit node ignoring iteration. The second measure is the count of the maximum number of simple predicates encountered while traversing any of the possible linear

paths. The maximum number of simple predicates is defined as the sum of all the simple predicates in each of the decision nodes encountered in a possible path. Examples of decision nodes are If, Do While, and the iterative Do statements. A N-way case statement is counted as  $N - 1$  simple predicates. The third measure is the count of the possible linear paths and the maximum number of simple predicates encountered in a possible path as a result of each different iteration.

#### B. Computation of Measure

The proposed measure of program complexity  $C$  is expressed as the tuple of the possible linear paths through a control flow graph ( $P$ ) and the maximum number of simple predicates ( $D$ ) encountered while traversing any possible path, plus the interval of the effects of each different iteration ( $\text{In}(P, D)$ ).  $P$  and  $D$  are for the path within the iteration.

$$C = (P, D) + \text{In}(P, D)$$

#### C. Example of Computation

Using the sample procedures in appendix A and its associated directed graph (figure 1),  $C$  is computed as follows:

$P$  = The count of the possible linear paths through the graph = 4

$D$  = The maximum number of simple predicates encountered while traversing any of these paths = 3

$\text{In}(P, D)$  = The count based on the effect of iteration =  $I(4, 3)$

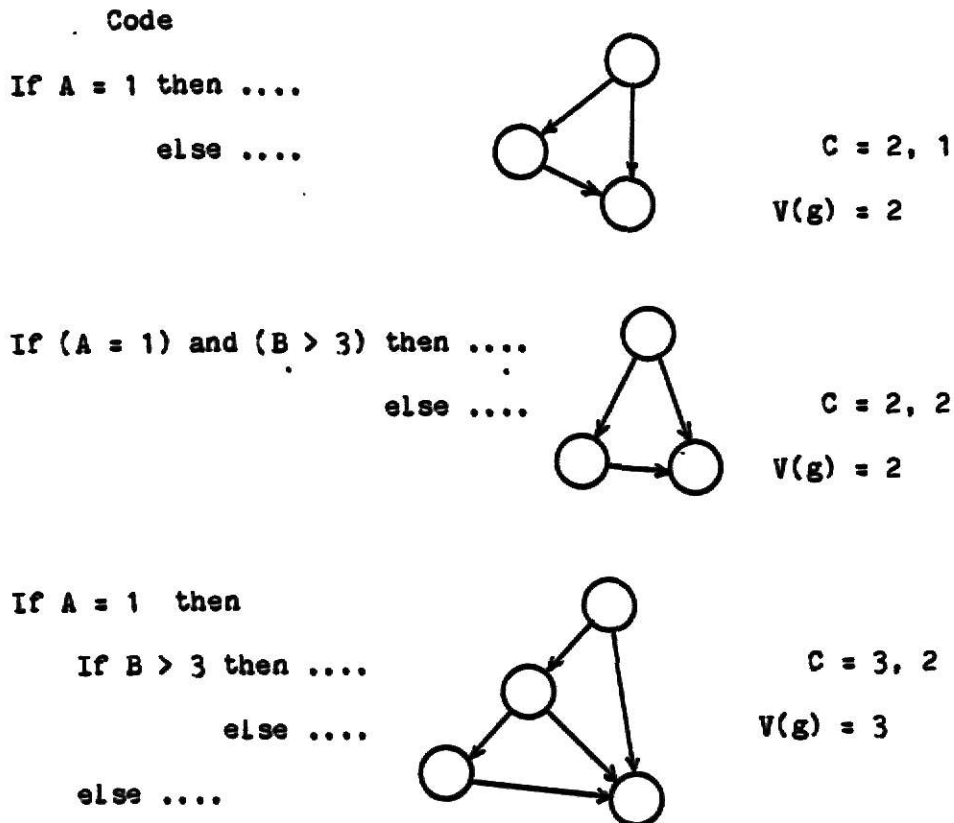
(this is because there is only one iteration and it affects all the possible paths and decision nodes encountered)

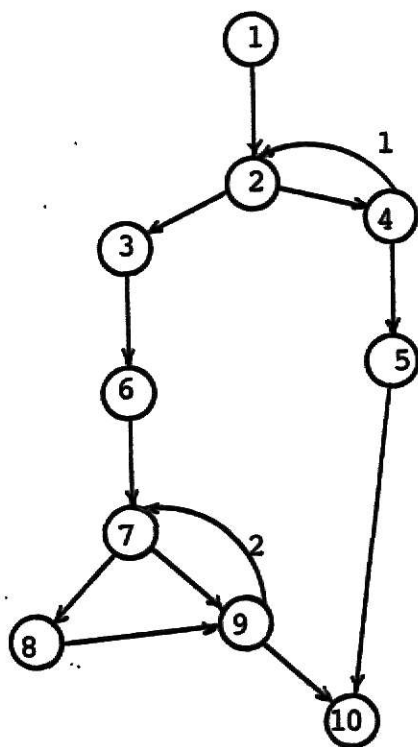
$$C = 4, 3 + I(4, 3)$$

## Other Examples and Comparative Analyses

The examples in figure 5 demonstrate the computation of the proposed complexity measure  $C$  when there is iteration that affects different segments of the graph and compares the results with the McCabe metric,  $V(g)$ :

The example in figure 6 demonstrates the computation of the proposed complexity measure when there is a Case statement and compares the results with the McCabe metric. Another difference between the proposed measure  $C$  and the McCabe metric is in the measurement of simple predicates versus the decision node itself. This is best described by Myers [10] in the following example:





(a)

$$P = 3$$

$$D = 2$$

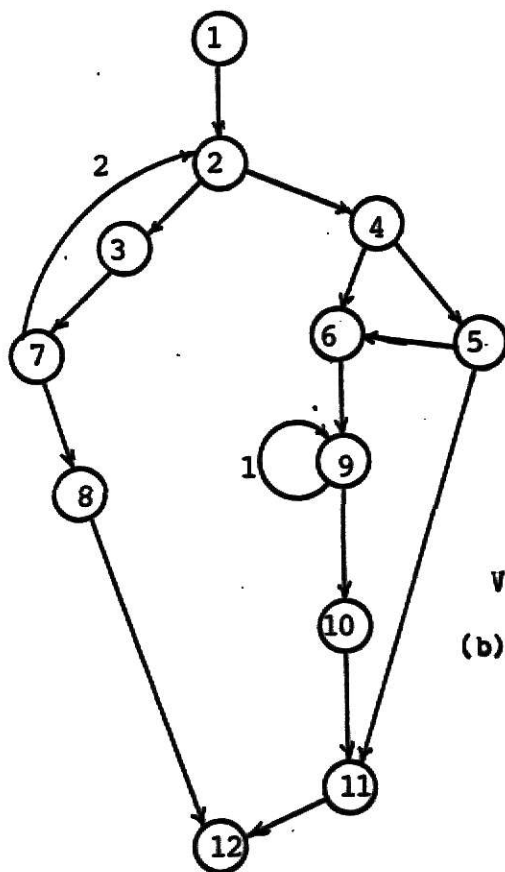
$$I1; P = 3, D = 2$$

$$I2; P = 2, D = 1$$

$$C = 3, 2 + I1(3, 2) + I2(2, 1)$$

$$V(g) = 5$$

Figure 5



(b)

$$P = 4$$

$$D = 3$$

$$I1; P = 0, D = 0$$

$$I2; P = 4, D = 3$$

$$C = 4, 3 + I1(0, 0) + I2(4, 3)$$

$$V(g) = 6$$

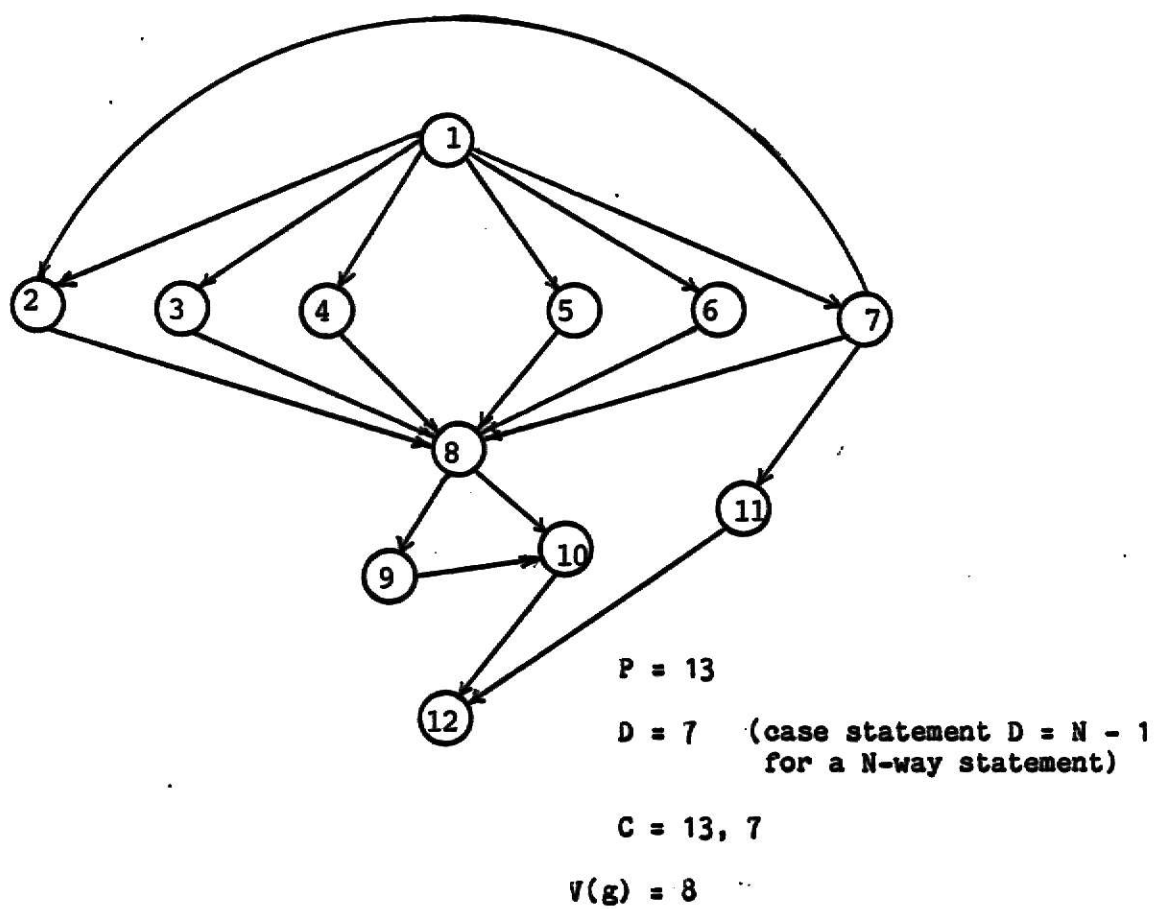


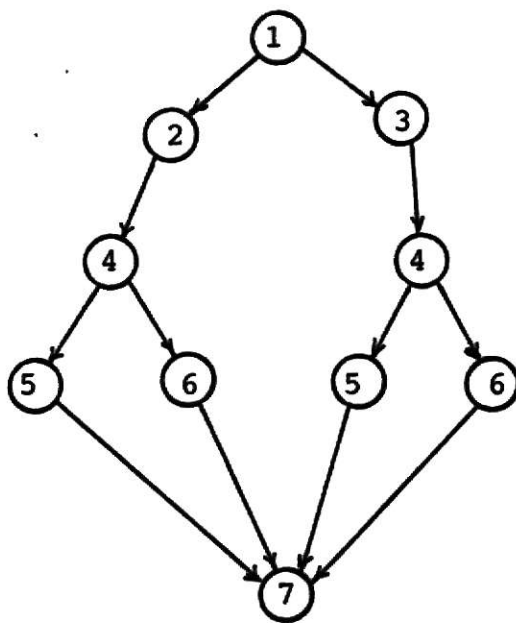
Figure 6

Another difference arises from the repeating of code segments and the changes in complexity when identical segments are used. This is described in the examples in figure 7. Figure 7 demonstrates that under the proposed measure repeating the same code will not increase the complexity. When this condition is examined from the testing and maintenance point of view, this is what would be expected.

The example in figure 8 demonstrates the computation of the proposed complexity measure when there is nested iteration. Each iteration is treated separately and only adds the amount of complexity which is effected by that iteration.

The overall comparison of each complexity measure as it relates to the sample program is appendix A is as follows:

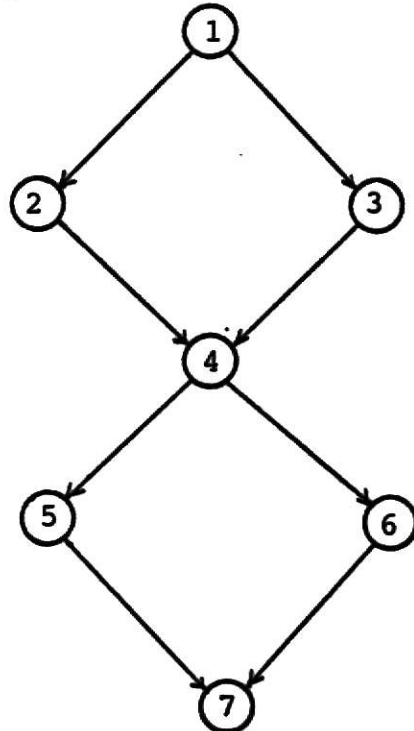
	McCabe	Chapin	McClure	Halstead	Proposed
Measured					
Complexity	4	5.43	3.5	3332.247	$4, 3 + I(4, 3)$



$$C = 4, 2$$

$$V(g) = 4$$

If we remove the duplicate segments we have:



$$C = 4, 2$$

$$V(g) = 3$$

Figure 7



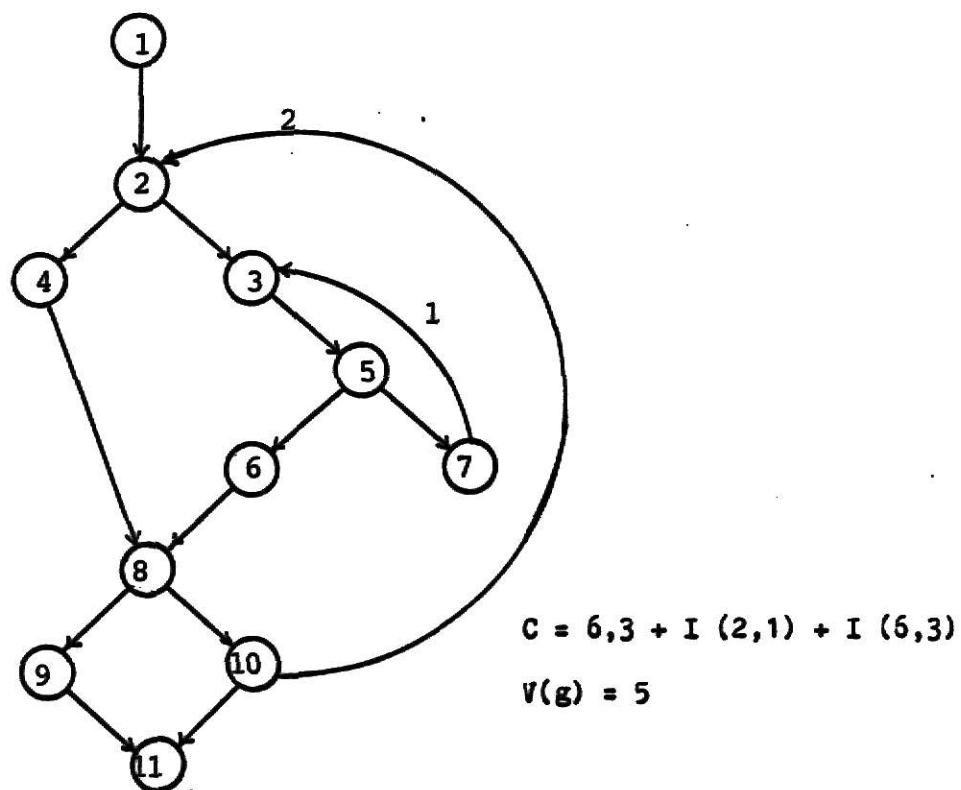


Figure 8

## Chapter 5

### Conclusions

Several complexity measures have been presented with demonstrations of their application to a sample program. Each of the measures presented derives complexity from one of three basic program properties; control structure, control flow, and data usage. I found that those complexity measures which are derived from the control structure and data usage properties were difficult to understand and difficult to apply.

Using program control flow as the property best suited for deriving program complexity a proposed measure was presented and compared to the cyclomatic number advocated by McCabe. Several examples were presented which demonstrated the ability of the proposed measure to overcome the shortcomings of McCabe's cyclomatic number and better rank programs according to their complexity.

## Chapter 6

### Future Work

The future work that still needs to be accomplished is :

1. Implement the proposed measure and test more sample programs collecting a broader base of data from which to judge how well the proposed measure actually predicted error occurrences and maintenance difficulties.
2. Identifying a mathematical representation of the proposed measure which would yield a single numerical quantity.
3. The determination of a more comprehensive method for evaluating complexity measures in general.

## Appendix A

### PROCEDURE TEST

comment Test all conditions for member identified by Current-node;

comment if all conditions are true add member to linked list;

Begin

Integer A, I;

Boolean Fair;

Fair := True;

I := 1;

While ((Check(I) not equal "P") and (Fair = True)) Do

Begin

Fair := Match(I);

I := I+1;

End;

End Do While;

If Fair = True then

Begin

A := Allocate 1;

If List-Pointer = null then

List-Pointer := A

else set LIST1(last, A);

Last := A;

set LIST1(last, null);

set LISTA(last, LIST2(Current-node + 1));

End;

End Test;

## REFERENCES

- [1] Zolnowski, Jean C., and D. B Simmons, "Measuring program complexity," Proceedings of the 1977 Fall CompCon, IEEE, Long Beach CA, 1977, pp. 336 - 340.
- [2] McCabe, Thomas J., "A complexity measure," IEEE Trans. Software Eng., vol. SE-2, pp. 308 - 320, Dec. 1976.
- [3] Chapin, Ned, "A measure of software complexity," Proceedings of the National Computer Conference 1979, InfoSci Inc., Menlo Park, CA, 1979, pp. 995 - 1002.
- [4] McClure, C. L., "Reducing COBOL Complexity through structured programming," Van Nostrand Reinhold, New York, 1978.
- [5] McClure, C. L., "A Model for program complexity analysis," Proceedings of 3rd Int Conf of S.E., pp 149 - 157.
- [6] Gordon, Ronald D., "A measure of mental effort related to program clarity," Ph.D. dissertation, Comput. Sci. Dep., Purdue University, 1977.
- [7] Halstead, M. H., "Towards a theoretical basis for estimating programming effort," Proceedings of ACM 1975, ACM New York, pp 222-224.
- [8] Halstead, M. H., "Advances in software science," Department of Computer Science, Purdue University, Lafayette, Indiana.
- [9] Halstead, M.H., "Elements of Software Science," Elsevier North-Holland, New York, 1977.
- [10] Myers, G.J., "An extension to the cyclomatic measure of program complexity," SIGPLAN Notices, Vol 12, No. 10, Oct. 1977. pp. 61 - 64.

A MEASURE OF PROGRAM COMPLEXITY

by

LLOYD DAVID BORCHERT

B.S., Oglethorpe University, 1972

---

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the  
requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY  
Manhattan, Kansas

1981

## ABSTRACT

This paper describes a program complexity measure and describes how it can be used. The paper first defines program complexity and discusses possible measurable program characteristics. The paper next reviews four different authors and their approach to providing a measurement of program complexity. The paper next explains the basis for the proposed complexity measure, describes how it is computed, and gives examples of computation. The next section of the paper gives a comparative analysis of the proposed complexity measure and the other measures reviewed. The last section of the paper outlines areas for possible future work in this area.