AN EXPERIMENT IN THE IMPLEMENTATION AND APPLICATION OF SOFTWARE COMPLEXITY MEASURES

by

RANDALL ROBERT MEALS

B. S., Iowa State University, 1977

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

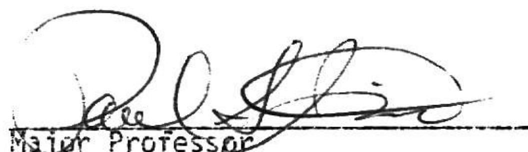
MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1981

Approved by:



Major Professor

SPEC
COLL
LD
2668
.R4
1981
M42
c. 2

TABLE OF CONTENTS

PAGE

CHAPTER 1
Introduction....................1
1.1 Purpose of Report....................2
1.2 Content of Report....................2

CHAPTER 2
Software Complexity Measures....................3
2.1 McCabe's Cyclomatic Complexity Measure....................6
2.2 Halstead's Metrics....................12

CHAPTER 3
Tools to Measure Software Complexity....................16
3.1 McCabe's Complexity Measure Evaluation Tool....................18
3.2 Halstead's Metrics Evaluation Tool....................21
3.3 Design Effort For McClure's Measure....................24

CHAPTER 4
Experience Applying Software Complexity Measures....................25
4.1 Results of Applying McCabe's Measure....................27
4.2 Results of Applying Halstead's Metrics....................36
4.3 Interpreting Results....................39

CHAPTER 5
Summary of Experiment....................41
5.1 Future Improvements and Directions....................42
5.2 Conclusion....................44

BIBLIOGRAPHY

APPENDICES

APPENDIX A
F-Spec for McCabe's Tool
1.0 Introduction....................A-1
1.1 Scope....................A-1
1.2 General Description....................A-1
1.3 References....................A-1
2.0 Environment....................A-2
3.0 Compatability Requirements....................A-2
4.0 Performance Requirements....................A-2
5.0 Interfaces....................A-3
6.0 Functional Description....................A-4
6.1 Functional Limitations....................A-4
6.2 Description of Input....................A-5
6.3 Description of Output....................A-5
6.3.1 Listing of Output....................A-5
6.3.2 Messages....................A-6
6.3.2.1 Insufficient Space in SIRE-FILE....................A-6
6.3.2.2 Insufficient Space in NAME-FILE....................A-6

6.2.3.2 Search For Non-existent Condition-Name....A-7
6.2.3.4 Error Encountered During Search of SIRE-FILE....A-7
6.2.3.5 Error Encountered During Read of NAME-FILE Record....A-7
7.0 Reliability and Maintainability....A-8
7.1 Reliability....A-8
7.2 Maintainability....A-8

APPENDIX B
F-Spec For Halstead's Tool
1.0 Introduction....B-1
1.1 Scope....B-1
1.2 General Description....B-1
1.3 References....B-1
2.0 Environment....B-3
3.0 Compatability Requirements....B-3
4.0 Performance Requirements....B-3
5.0 Interfaces....B-4
6.0 Functional Description....B-5
6.1 Functional Limitations....B-6
6.2 Description of Input....B-6
6.3 Description of Output....B-7
6.3.1 Listing of Output....B-7
6.3.2 Messages....B-8
6.3.2.1 Error Encountered During Search of LINK-FILE....B-8
6.3.2.2 Error Encountered During Search of OPERAND-FILE....B-8
6.3.2.3 Insufficient Space in LINK-FILE....B-9
6.3.2.4 Insufficient Space in OPERAND-FILE....B-9
6.3.2.5 Error Encountered During Rewrite of LINK-FILE Record....B-9
6.3.2.6 Error Encountered During Rewrite of OPERAND-FILE Record....B-10
7.0 Reliability and Maintainability....B-11
7.1 Reliability....B-11
7.2 Maintainability....B-11

APPENDIX C
I-Spec For McCabe's Tool
1.0 Overview....C-1
1.1 Scope....C-1
1.2 General Description....C-1
1.3 References....C-1
2.0 Environment....C-2
3.0 Design Summary....C-2
3.1 System Flow....C-2
3.2 Logic Flow....C-3
3.3 Program Flow....C-4
4.0 Component Description....C-6
4.1 Local Data....C-6
4.2 Routines....C-10

5.0 Development and Maintenance....C-13
5.1 Development Systems....C-13
5.2 Maintenance Considerations....C-13

APPENDIX D
I-Spec For Halstead's Tool
1.0 Overview....D-1
1.1 Scope....D-1
1.2 General Description....D-1
1.3 References....D-1
2.0 Environment....D-3
3.0 Design Summary....D-4
3.1 System Flow....D-5
3.2 Logic Flow....D-5
3.3 Program Flow....D-6
4.0 Component Description....D-8
4.1 Local Data....D-8
4.2 Routines....D-13
5.0 Development and Maintenance....D-16
5.1 Development Systems....D-16
5.2 Maintenance Considerations....D-16

APPENDIX E
McCabe's Tool Source Code

APPENDIX F
Halstead's Tool Source Code

APPENDIX G
McCabe's Measure - Tables of Results

APPENDIX H
Halstead's Metrics - Program A - Version 1

APPENDIX I
Halstead's Metrics - Program A - Version 2

APPENDIX J
Halstead's Metrics - Program B - Version 1

APPENDIX K
Halstead's Metrics - Program B - Version 2

APPENDIX L
Halstead's Metrics - Program C - Version 1

APPENDIX M
Halstead's Metrics - Program C - Version 2

APPENDIX N
Halstead's Metrics - Rank Order Frequency of Operators

LIST OF FIGURES

PAGE

Figure 2.1
Proposed Complexity Measures..4

Figure 2.2
Example of Flow Graph where V(G) = 4.................................7

Figure 2.3
Example of Flow Graph where V(G) = 7.................................9

Figure 2.4
Example of Flow Graph where V(G) = 5.................................11

Figure 4.1
Frequency of Occurrence of Cyclomatic Complexity Values for the First and Second Versions of Program A.........................30

Figure 4.2
Frequency of Occurrence of Cyclomatic Complexity Values for the First and Second Versions of Program B.........................31

Figure 4.3
Frequency of Occurrence of Cyclomatic Complexity Values for the First and Second Versions of Program C.........................32

Figure 4.4
Frequency of Occurrence of Cyclomatic Complexity Values for the First Versions of the Measured Programs........................33

Figure 4.5
Frequency of Occurrence of Cyclomatic Complexity Values for the Second Versions of the Measured Programs.......................34

Figure 4.6
Summary of Results of Applying McCabe's Cyclomatic Complexity Measure...35

Figure 4.7
Comparison of Actual Error Count of Estimated Error Count............38

Figure C.1
General Program Flow Through McCabe's Complexity Measure Evaluation Tool...C-5

Figure D.1
General Program Flow Through Halstead's Metrics Evaluation Tool......D-7

Figure G.1
Table of Cyclomatic Complexity Values Found in the First Version of Program A...G-1

Figure G.2
Table of Cyclomatic Complexity Values Found in the Second Version of Program A...G-2

Figure G.3
Table of Cyclomatic Complexity Values Found in the First Version
of Program B..G-3

Figure G.4
Table of Cyclomatic Complexity Values Found in the Second Version
of Program B..G-4

Figure G.5
Table of Cyclomatic Complexity Values Found in the First Version
of Program C..G-5

Figure G.6
Table of Cyclomatic Complexity Values Found in the Second Version
of Program C..G-6

Figure N.1
Graph of Logarithmic Plot of Rank Order Frequency of Operators
for the First Version of Program A............................N-1

Figure N.2
Graph of Logarithmic Plot of Rank Order Frequency of Operators
for the Second Version of Program A...........................N-2

Figure N.3
Graph of Logarithmic Plot of Rank Order Frequency of Operators
for the First Version of Program B............................N-3

Figure N.4
Graph of Logarithmic Plot of Rank Order Frequency of Operators
for the Second Version of Program B...........................N-4

Figure N.5
Graph of Logarithmic Plot of Rank Order Frequency of Operators
for the First Version of Program C............................N-5

Figure N.6
Graph of Logarithmic Plot of Rank Order Frequency of Operators
for the Second Version of Program C...........................N-6

ACKNOWLEDGEMENTS

I would like to thank my major professor, Dr. Dave Gustafson, for his guidance and encouragement, to thank NCR Corporation for the availability of its resources, and to especially thank my wife, Linda, for her patience, understanding, support, and love.

CHAPTER 1

Introduction

This report is a summary of a project that focused on gaining experience in the implementation and application of software complexity measures. Much of the emphasis in adopting structured programming techniques is due to the perceived need to provide guidelines through which to reduce the complexity and increase the understandability of modules of software. A major factor motivating this emphasis is the high level of maintenance costs often experienced during the life-cycle of a piece of software. Intuitively, the complexity of a software module is somehow related to the difficulty experienced in developing, understanding, testing, modifying, and consequently maintaining that piece of software. With maintenance costs consuming the vast majority of the D.P. budgets of many organizations, there is a very real and pressing need to have some means by which to quantitatively measure software to see if it conforms to programming style guidelines which enforce and reinforce adherence to whatever structured programming techniques an organization selects. Software complexity measures are an attempt to fill this need.

The two complexity measures selected for this project are based on work done by Thomas J. McCabe [McCabe, 1976] and the late Maurice H. Halstead [Halstead, 1977].

1.1 Purpose of Report

This report is to serve as the documentation of the experiment with software complexity measures. It is by no means complete in the pursuit of this purpose. The goal of the project was to evaluate a set of software, for which the error history is known, against several different complexity measures. Many authors have presented what they view as a suitable means of measuring software. Some of these measures have even achieved a degree of universal acceptance, such as the ones selected for this project. An important aspect of this project is to determine if software which is indicated to be of high complexity in terms of a specific complexity measure proves to be problematic in terms of error history.

1.2 Content of Report

Presented in this report is the various aspects and issues involved in the experiment in the utilization of software complexity measures. Examined, to various degrees, are the selection of appropriate complexity measures, the history, determination, and implications of the selected complexity measures, and the experience of implementing and using tools to apply the selected software measures. There is an appendix containing functional and implementation specifications for the selected complexity measures, and listings of the tools used to apply the two measures used in this project, as well as listings and tables of the results of the software complexity measures applied to the set of software examined.

CHAPTER 2

Software Complexity Measures

Prior to the evolution and acceptance of Halstead's metrics and McCabe's cyclomatic measure, attempts at quantifying and limiting the complexity of a piece of software consisted of arbitrary guidelines which recognize that there is a rough correlation between module sizes and the understandability, reliability, and maintainability of those modules. The arbitrary guidelines took such forms as IBM limiting the size of modules to 50 source lines and TRW setting an upper bound of 2 pages of source code [McCabe, 1976]. While these limitations are no doubt useful to some extent, they are far from adequate when seeking a rigorous means by which to measure such aspects of software as clarity, testability, expected error rates, and goodness of the modularity scheme.

In 1972 Halstead published his theory of Software Physics [Halstead, 1972], which has since been renamed Software Science. Since then, there has been an avalanche of proposed software complexity measures and enhancements to proposed complexity measures. Although it is certainly not complete, in [Zolnowski and Simmons, 1980] the authors make a reasonable attempt at presenting a table listing, categorizing, and describing many of the software complexity measures proposed up to the point of publication of their report. The table is reproduced in Figure 2.1 for the reader's edification. Of these proposed measures, consideration for automation was given to Halstead's, McCabe's, McClure's, and Zolnowski and Simons', among others.

At this point, it is important to clarify the difference between measures of computational complexity of software and measures of the

Orientation of Measure	Description	Author
Control Flow	Cyclomatic Number	McCabe [1976]
	Count of Program Paths	Sullivan [1973]
	Enhancement of Cyclomatic Number (includes a count of logical conditions)	Meyers [1977]
	Measurement by the Pair (Cyclomatic Number, Operator Count)	Hansen [1978]
	Number of Multiple Entry Loops	Peterson [1973]
	Number of Knots	Woodward, et al. [1979]
	Cyclomatic Complexity Interval Plus # lines into/out of line of code	Cobb [1978]
Module Interaction	Number of Modules or Subsystems	Gilb [1977]
	$R\left(\frac{\text{Number of Module linkages}}{\text{Number of Modules}}\right)$	Gilb [1977]
	Measure based on control structures and control variables	McClure [1978]
Data Reference	Measure of difficulty in understanding software's function based on components of sets on input and output	Chapin [1979]
Program Control	Minimal Intersection Number	Chen [1978]
Logical Complexity	$R\left(\frac{\text{Number of non-normal exits from a decision statement}}{\text{Total number of instructions}}\right)$	Gilb [1977]
Software Science	Metrics of software science predict complexity of a program	Halstead [1977]
	Approach complexity via statistical (natural) language	Shooman & Laemmel [1977]
Composite Measure of Complexity	Index of Complexity based on Structure/Interaction/Instruction Mix/Data Reference Program Characteristics	Zolnoswski & Simmons [1977]
	Interface complexity/Computational complexity/I/O complexity/Readability	Thayer [1976]

Figure 2.1 Proposed Complexity Measures

psychological complexity of software. Computational complexity is understood as "the quantitative aspects of the solutions to computational problems" [Rabin, 1977]. An example of usage of computational complexity measures would be in comparing the efficiency of alternate algorithmic solutions [Curtis, et al., 1979]. Psychological complexity measures, of the type considered and utilized in this project, assess human performance in programming activities. Although there are certainly correlations between the two concepts, there is not expected to be any simple relationship between computational and psychological complexity measures.

The measures selected, those of T. McCabe and M. Halstead, are psychological software complexity measures. These two measures were selected out of the many choices due to their relative simplicity in what parameters are examined in the software being evaluated. This simplicity of parameters leads to a low level of computational complexity. When compared to psychological complexity measures proposed by McClure and Zolnowski and Simons, for instance, the cyclomatic complexity measure of McCabe and the Software Science metrics of Halstead are relatively easy to automate, yet the results of these measures satisfy many aspects of our intuitive notion of software complexity.

2.1 McCabe's Cyclomatic Complexity Measure

The definition of software complexity proposed by T. McCabe is based on the decision or control flow structures present in a piece of software. The actual metric used is the classical graph theory cyclomatic number indicating the number of regions in a graph. The graph used is the one classically known as the program control graph [Legard and Marcotty, 1975], where each node in the graph corresponds to a block of code where the flow is sequential and each of the arcs of the graph correspond to conditional branches in the blocks of code. The cyclomatic number of the graph is equal to the number of linearly independent control paths comprising a program [Curtis, et al., 1979]. Fundamentally, the measure then is the minimal number of independent paths in a module that has a single entry and a single exit.

The following examples will present three different, but equivalent methods by which McCabe's cyclomatic complexity measure may be computed. The first formula that can be used is:

$$V(G) = e - n - 2p$$

where $V(G)$ = cyclomatic complexity of graph,

e = number of edges,

n = number of nodes, and

p = number of connected components in the graph ($p = 1$ for single entry/single exit modules).

In Figure 2.2, the graph given there has 9 edges and 7 nodes. Thus

$$V(G) = e - n + 2$$

$$V(G) = 9 - 7 + 2$$

$$V(G) = 4$$

This means that there are four independent paths in the graph in

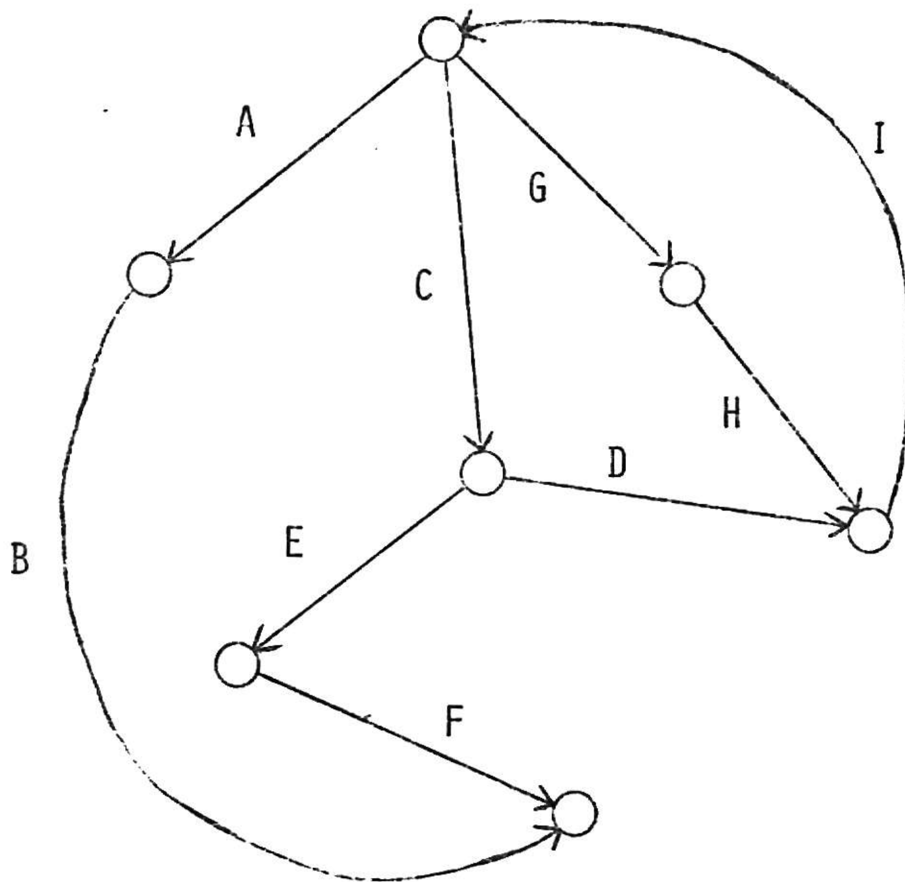


Figure 2.2 Example of Flow Graph where V(G) = 4

Figure 2.2 from which all paths can be made. These four independent paths are:

(1) AB

(2) GHI

(3) CDI

(4) CEF

A second formula for determining McCabe's cyclomatic complexity measure is:

$$V(G) = \pi + 1$$

where $V(G)$ = cyclomatic complexity of graph, and

π = number of predicate nodes (nodes with multi-exit paths).

In this formula, each binary exit node contributes 1 to the cyclomatic complexity, while each case-type node contributes one less than the number of exit-paths. In Figure 2.3, the graph given has three binary nodes and one case node. Thus

$$V(G) = \pi + 1$$

$$V(G) = [3 + (4 - 1)] + 1$$

$$V(G) = 7$$

Note that in this second example, it is not actually necessary to look at the flow graph. Instead, all that is necessary to compute the complexity of a module is to simply count the number of predicates (i.e. conditionals) in the code.

A third method for calculating McCabe's cyclomatic complexity measure is to use Euler's formula. If the flow graph is a planar graph, then:

$$V(G) = R$$

where R = the number of bounded regions on the graph, with the

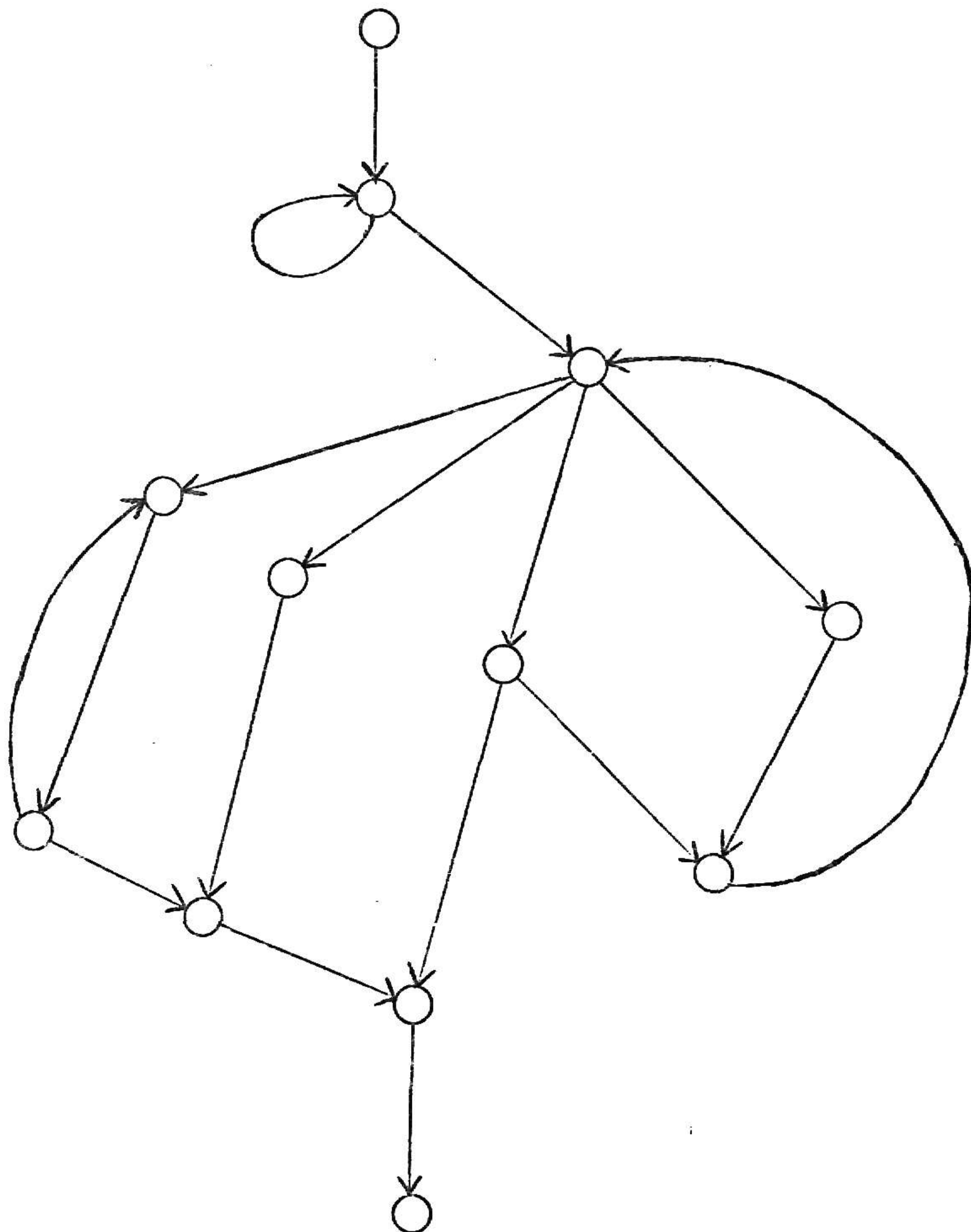


Figure 2.3 Example of Flow Graph where V(G) = 7

region external to the graph also counted as one. In Figure 2.4, the graph has the five bounded regions identified.

By utilizing the three methods given for each of the graphs in Figure 2.2, Figure 2.3, and Figure 2.4, it can be seen that the methods are equivalent. A proof of the equivalence can be found in [McCabe, 1976].

There are several properties of McCabe's cyclomatic complexity measure, which are restated below from [McCabe, 1976]:

1) $V(G) \geq 1$.
2) $V(G)$ is the maximum number of linear independent paths in G; it is the size of a basis set.
3) Inserting or deleting functional statements to G does not affect $V(G)$.
4) G has only one path if and only if $V(G) = 1$.
5) Inserting a new edge in G increases $V(G)$ by unity.
6) $V(G)$ depends only on the decision structure of G.

Although the concept was not used in this experiment, essential complexity will be briefly introduced here. Using the work of Rao Kasaraju [Kasaraju, 1974], McCabe expands on the concept of reducibility of structured and unstructured software, and defines essential complexity. The essential complexity, ev, of a module is the complexity that the flow graph can be reduced to when appropriate single entry/single exit subgraphs are successively replaced by structured single nodes. The minimal essential complexity of an unstructured program is 3, whereas a structured program is reducible to a program whose complexity is 1.

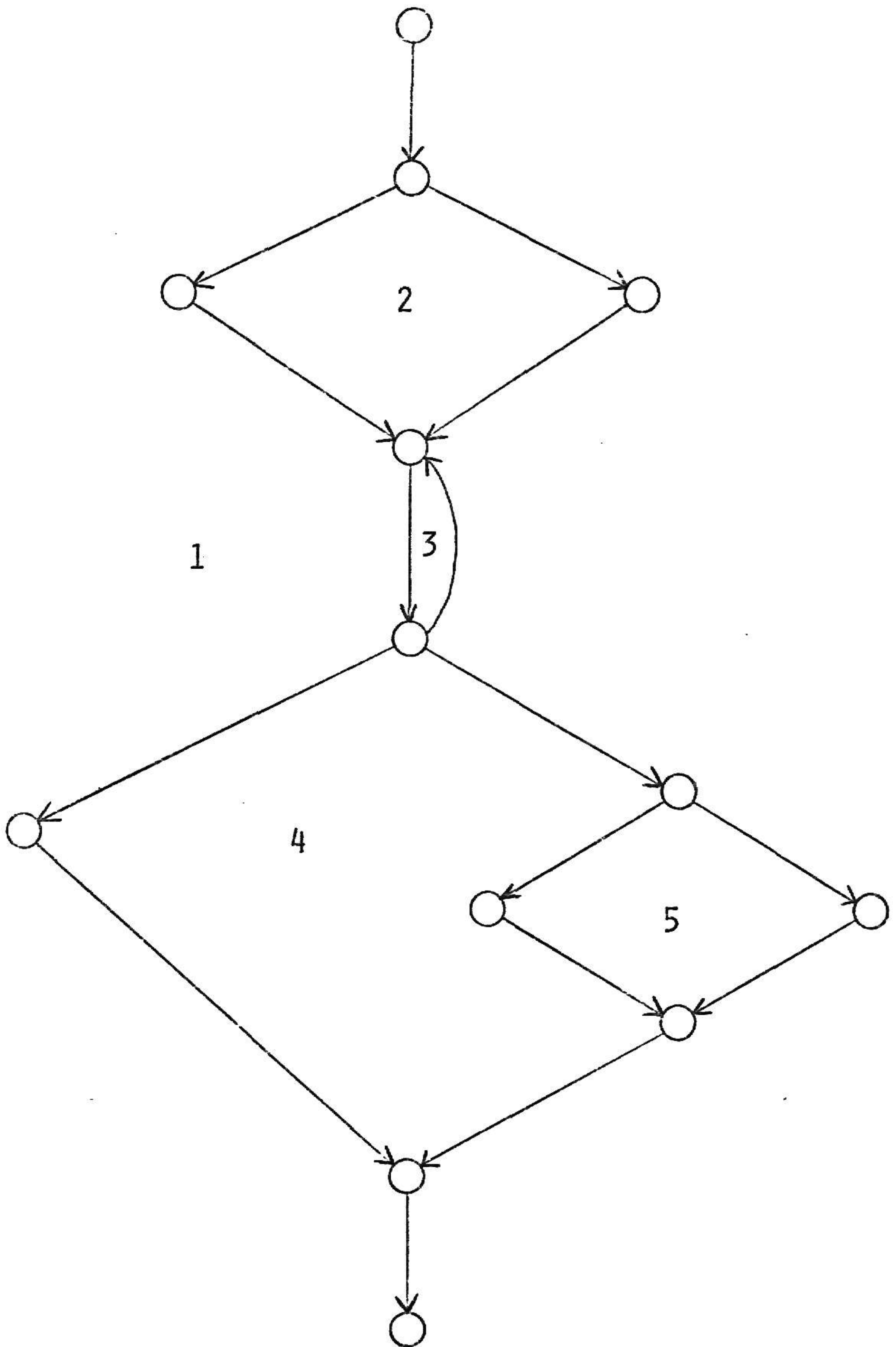


Figure 2.4 Example of Flow Graph where V(G) = 5

2.2 Halstead's Metrics

The metrics of Halstead's Software Science span more than just the simple measurement of the complexity of computer programs. The "software" of Software Science is any communication that appears in symbolic form in conformance with the grammatical rules of any language [Halstead, 1979]. The properties of a program, as defined by Halstead, are based upon four measured parameters:

1) η_1 - the number of unique operators,
2) η_2 - the number of unique operands,
3) N_1 - the total frequency of operators, and
4) N_2 - the total frequency of operands.

From these measured parameters, there are several properties that the theories of Software Science define as useful measures of a piece of software. The computed measures used in this project inlcude:

1) Program Vocabulary - $\eta = \eta_1 + \eta_2$
2) Program Length - $N = N_1 + N_2$
3) Estimated Program Length - $\hat{N} = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$
4) Program Volume - $V = (N_1 + N_2)\log_2(\eta_1 + \eta_2)$
5) Implementation Level - $L = V^*/V$
6) Estimated Implementation Level - $\hat{L} = (2/\eta_1)(\eta_2/N_2)$
7) Estimated Potential Volume - $\hat{V}^* = \hat{L}V$
8) Effort Metric - $E_c = V/L$
9) Programming Time - $\hat{T} = E_c/S$, where S is Stroud's number, 18
10) Language Level - $\lambda = (L)^2 V$

Although a brief discussion of each of these computed measures will be given here, the reader is referred to [Halstead, 1979] for a more complete description of the fundamentals and implications of each metric.

The first two metrics in the list, Program Vocabulary, η, and Program Length, N, should almost appear obvious. The count of items in a program's vocabulary is simply the sum of the number of unique operators and the number of unique operands. The total number of occurrences of operators and operands is defined as the length of the program.

The equation for the estimation of Program Length, $\hat{N}$, is a result of an unexpected, and initially counterintuitive finding in the field of Software Science, that the lengths of programs are determined almost exclusively by the sizes of the vocabulary components η_1 and η_2. The equation estimating Program Length is thus an expression of the vocabulary-length relationship. There have been studies [Elshoff, 1978] where correlation coefficients between N and $\hat{N}$ of over 0.98 have been reported for large groups of programs. There have been identified at least six "Impurity Cases" which can be the cause of a program not conforming to the vocabulary-length relation [Halstead, 1979]:

(1) Complementary Operations, such as adding and immediately subtracting one operand from another,

(2) Ambiguous Operands, the use of one operand name to serve more than one purpose, most likely in different parts of a program,

(3) Synonymous Operands, the use of two different variables whose values are always the same,

(4) Common Subexpressions, the failure to assign a name to the result of a frequently used calculation,

(5) Unwarranted Assignment, the assignment of an operand name to

the result of a calculation used only one, and

(6) Unfactored Expression, the failure to factor a factorable expression.

It is suggested that the inclusion of Impurity Cases in a piece of software makes that software harder to implement and understand.

The equation of Program Volume, V, is independent of the actual number of letters or characters in the identifiers found in a piece of software. The Program Volume is the fewest number of binary digits, or bits, with which a particular piece of software (program) can be represented.

The Implementation Level, L, of a piece of software is a measure of the brevity and compactness of an implementation of an algorithm or function. The Implementation Level may differ for functionally equivalent programs, depending on the power or level of language in which they are expressed. The equation estimating Implementation Level, $\hat{L}$, has been found to be close enough to be used interchangably with L.

It is obvious that the volume of a program varies with the language in which the program is written. Translation of a piece of software into another language may result in an increase or a decrease in volume. Assuming that programs are translatable into ever more powerful languages, there would exist a potentially optimal language, and a potential volume, where the potential volume is the function of the program expressed as a procedure call in the potentially optimal language. Thus, the computed Potential Volume, V^*, represents the minimum possible volume associated with the function of a piece of software, and an absolute value against which others can be compared. For the purposes of this project, as L can be reasonably estimated, using the definition of L, V^* can be

estimated by substitution.

The Effort Metric, E_c, is a ratio of the Program Volume to the Implementation Level, and is an estimation of the mental effort required to create a program. The Program Volume metric serves as an indicator of the number of mental comparisons required to generate a program. The Implementation Level metric is the inverse of difficulty, as it is an indicator of the number of elementary mental discriminations (e. m. d.) required to make one average mental comparison. Thus, the difficulty of creating a program increases as the Program Volume increases, and decreases as the Implementation Level decreases, and the Program Volume multiplied by the inverse of the Implementation Level results in the total number of elementary mental discriminations required to generate the complete program.

By knowing the rate at which elementary mental discriminations occur, the amount of time required to generate the completed program can be computed. Fortunately, psychology provides such a rate, which is nearly constant, and does not vary significantly with intelligence. The number is named in honor of John Stroud, who did work in the examination of psychological time [Stroud, 1966], and who first reported this rate, which, for time in seconds, in taken as 18 [Halstead, 1979].

The equation for determining the Language Level, λ, of a program is equivalent to the Implementation Level multiplied by the Potential Volume. It has been found experimentally that as Potential Volume is increased, Implementation Level will decrease proportionately [Halstead, 1979]. Thus, the average Language Level for programs written in the same language tends to remain constant over a wide range of program sizes. For English, the average Language Level is 2.16.

Chapter 3

Tools to Measure Software Complexity

The automation of the measures of Halstead and McCabe was accomplished by writing separate programs for each measure in COBOL. The machines on which the measures were developed are members of NCR's I-Family of computing systems. The I-Family consists of members of the NCR 8000 series of computers that operate in the interactive direct processing or "I" mode. The I-Family of systems is oriented to conversational, online processing where there is continual interaction between machine and operator. The members of the I-Family range from desk-top micros to minis to mainframes. The major development system used in this project was an I8270 with 1024K bytes of memory, and the IMOS V operating system.

The COBOL language was selected due to its commonality among the I-Family systems. This commonality extends to a common COBOL virtual machine into which COBOL programs are compiled, using a common COBOL compiler. The COBOL virtual machine is emulated via various hardware/firmware/software mixtures in the individual systems of the I-Family. The common COBOL compiler and virtual machine thus allow the complexity measures to be transportable in both source and object form between the members of the I-Family. An additional advantage of using COBOL as the implementation language is that, with minor modifications to remove I-Family specific screen-formatting instructions, the source for the complexity measure tools is transportable to NCR's virtual, or "V" mode systems. The "V" mode systems use the VRX operating system, and have

a separate COBOL machine for the VRX environment. The VRX environment is implemented on small through medium through large mainframes and is primarily batch oriented.

3.1 McCabe's Complexity Measure Evaluation Tool

The tool to evaluate a COBOL program in terms of McCabe's cyclomatic complexity measure was the first of the two complexity measure tools written. A functional specification for the tool can be found in Appendix A, an implementation specification for the tool can be found in Appendix C, and a source listing of the completed COBOL program implementation can be found in Appendix E.

The initial problem to be solved in the design of the McCabe's Complexity Measure Evaluation Tool was the evolution of a technique to isolate tokens from the source program being evaluated. The tool first had to handle the fact that the COBOL source input could be in either fixed- or variable-length record format. The code in the Declaratives section of the program does this by handling fixed-length records as the standard situation and variable-length records as the exception case. The token isolation routines, when presented with a line of code, will sequentially scan each character of the line, and isolate source words based on known delimiters. If a literal is encountered, a special routine which can accomodate embedded quotation marks and allow literals up to the standard maximum of 255 characters is invoked.

In the calculation of McCabe's cyclomatic complexity measure, the methodology was to count all the decision points in the Procedure Division of a program, summing the count for each paragraph and section in the program, and for the whole program, and calculating the average complexity of the paragraphs in the entire program. In all cases, a paragraph is assumed to be equivalent to a module and thus have an initial complexity value of one. In a complex conditional expression, each simple conditional expression was assigned a value of one. In COBOL, a conditional

expression may consist of a condition-name or a combination of condition-names, as well as explicit conditional expressions. These condition-names can be user-defined, such as switches and 88-level items, or part of the language definition, such as "ALPHABETIC", "NUMERIC", etc. In order to recognize user-defined condition-names that may occur in conditional expressions, and thus include their implicit complexity values in the computations, the *Environment Division* is scanned for any 88-level items. In the case of an 88-level item, a single value is counted as adding a value of one and a range, such as "A THRU Z", is counted as adding a value of two. This is due to the assumption that the single value can be tested using an equality conditional and the range can be tested using greater-than-or-equals and less-than-or-equals conditionals.

In order to retain the information associated with condition-names, specifically switches and 88-level items, two scratch work files are used. The reason for this is that the language that the tool is written in, COBOL, does not have dynamic data structures, such as the heap in PASCAL, and the work files are the only reasonable means by which to handle information of indeterminate size. The side effect of this is, given that the source program is coded as efficiently as possible, the majority of the execution time is spent in disk accesses, mostly due to obtaining the next source line, but sometimes due to searching for a particular condition-name.

In the implementation phase, the largest problem encountered was smoothing out the token isolation and advancement procedures, especially the parsing of specific COBOL *syntactic structures* in the Procedure Division. In order to facilitate this debugging process,