OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

DEPARTMENT OF COMPUTER SCIENCE
OREGON STATE UNIVERSITY
CORVALLIS, OREGON 97331

A SURVEY OF GRAPH THEORETIC COMPUTER PROGRAM COMPLEXITY MEASURES

Curtis R. Cook
Warren Harrison

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

83-1-2

OSU 2569

DEPARTMENT OF COMPUTER SCIENCE
OREGON STATE UNIVERSITY
CORVALLIS, OREGON 97331

A SURVEY OF GRAPH THEORETIC COMPUTER PROGRAM COMPLEXITY MEASURES

Curtis R. Cook
Warren Harrison

Computer Science Department
Oregon State University
Corvallis, Oregon 97331

ABSTRACT

A computer program complexity measure is a measure of how easy the program is to understand, test, modify, maintain, etc. Many of these measures are derived from the control or flow graph of the program. We describe these measures graph theoretically, indicate what aspect or aspects of the program they measure and compare their strengths and weaknesses.

KEYWORDS : Graph theory, psychological complexity, complexity measures, flow graph, program graph, call graph

INTRODUCTION

Studies of programming have revealed that over the life of a program nearly two-thirds of the cost is spent on maintenance and over one-half of the time is spent testing. Both maintenance and testing are primarily human activities. Hence the complexity or ease of working with or understanding a program is extremely important.

There are two approaches to assessing how difficult a given program will be to work with. We can either

a) actually measure the performance of a programmer working with the program, or
b) we can observe that the program has characteristics similar to those possessed by other programs that we have found to be difficult to work with by following the methodology of a).

Research on software (psychological) complexity has attempted to identify and measure characteristics of a program that determine how difficult or easy it is to work with. Note that for the purposes of this paper, software complexity is associated with the programmer rather than program performance. The term computational complexity refers to the formal mathematical analysis of the efficiency (execution time, memory space used) of the program algorithm, data structures, etc.

A software complexity metric is a mapping from computer programs into the positive integers. The interpretation of the mapping is the larger the integer the more complex the program. Thus for two programs A and B, if the value of A is less than the value of B, then program A is "less complex" than program B.

Software complexity measures have several uses. Probably the most immediate is to provide feedback to the programmer about his or her code. It could indicate that it will be difficult to test or maintain and should be rewritten. It could also be used as a measure of the quality of the program such as an acceptance standard. A software complexity measure could also be used to predict the resources needed to implement and test or the number of errors or the difficulty in maintaining the program.

There is no one measure of software complexity. In fact there is no commonly accepted definition of complexity. It is acknowledged that complexity is a multi-faceted, relative and elusive concept. Everyone has nearly the same general idea of what complexity is, but nearly everyone disagrees on how to measure it, or for that matter, if it can be measured. Often complexity is defined in terms of the proposed measure.

Many of the software complexity measures are derived from a static analysis of a graph associated with the program. These measures are based on program properties such as flow of control within a program or between program modules. In the next section we will define the graphs most commonly associated with computer programs. Of these, the control graph or flow graph is the most common.

Section 3 describes several graph theoretic complexity measures. For each measure we will define it graph theoretically and indicate the program property it measures. We will also give an evaluation of the metric in terms its utility and ease of computing. Section 4 summarizes the graph theoretical measures and indicates directions for future work in the area.

GRAPHS ASSOCIATED WITH PROGRAMS

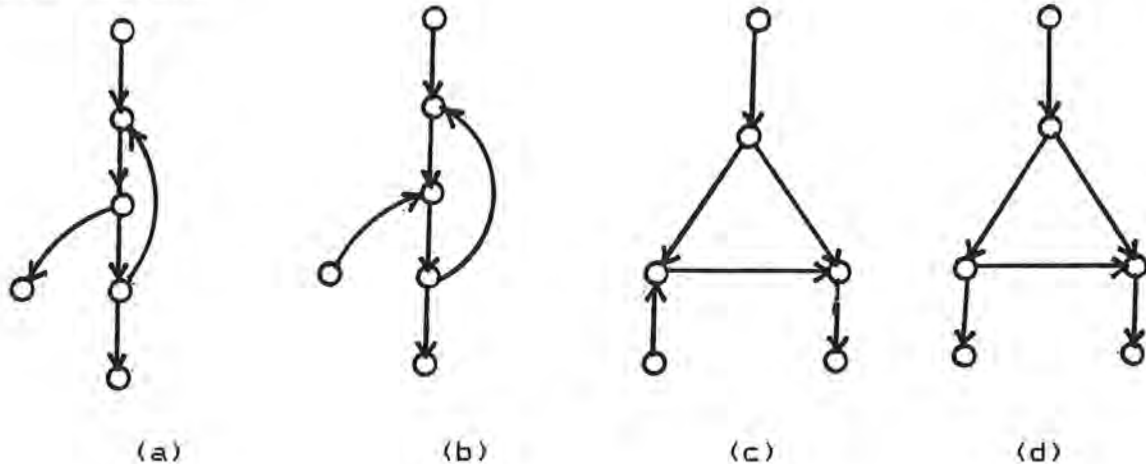
One of the simplest graphs associated with a computer program is the program graph [8]. It has a node for each program statement and an arc (a,b) if statement b may be executed immediately after after statement a. An example of a program graph is given in Figure 1. This definition works well for Fortran, Basic, Cobol or other languages without a grouping statement. However for languages with a grouping statement such as the compound statement in Pascal or Algol, one or more statements may be part of a single statement. A compound statement consists of a BEGIN followed by one or more statements followed by an END. A problem is how to represent this in the graph. Probably the simplest solution is to treat the BEGIN, END and each statement in the compound statement as statements.

One obvious disadvantage of the program graph is its size - one node per statement. Usually straight-line sequences of program statements (set of nodes with indegree 1 and outdegree 1) are considered no more complex than a single statement. In a flow graph or control graph, straight-line sequences are replaced by a single node. A program block (or just block) is a sequence of consecutive program statements that can only be entered at the first statement and exited from the last statement and there is no flow of control to other than the next statement. The nodes of the flow graph (control graph or control flow graph) are the program blocks and the edges correspond to a branch or flow of control between blocks. The flow graph of the Bubble Sort program of Figure 1 is given in Figure 2. Flow graphs are the graphs most frequently used in defining software complexity measures.

A third graph is the call graph. Its nodes are the main program and its subprograms (procedures, subroutines, functions, etc.). An edge (a,b) represents the invocation of subprogram b in subprogram a. See Figure 3. Note that a subprogram may contain several different invocations of the same subprogram. Hence the call graph may have multiple edges between a pair of nodes. Also a non-recursive program will have an acyclic call graph; the call graph of a recursive program will contain one or more cycles.

Graphs have many applications in the programming area besides software complexity measures. One example is structured programming. A program is said to be structured if it is written using only the three building blocks : sequence, if-then-else and do-while. See Figure 4. A structured program is characterized by a "forbidden subgraph" characterization of its flow graph.

Theorem [7]: A program is nonstructured if and only if its flow graph contains a subgraph ismorphic to one of the four graphs given below.



(a) (b) (c) (d)

Corollary: If a program is nonstructured it contains at least two of the forbidden subgraphs.

Note that the four graphs in the Theorem correspond to :

(a) Branching out of a loop.
(b) Branching into a loop.
(c) Branching out of a decision.
(d) Branching into a decision.

SOFTWARE COMPLEXITY MEASURES

In this section we will describe the most common graph theoretic software complexity measures. Our descriptions will include both the graph theory definition and the program property it measures. We will discuss the merits and shortcomings of each measure.

1. McCabe's Cyclomatic Complexity

McCabe [7] defined the complexity of a program as the cyclomatic number of its control graph. The cyclomatic number V(G) of a graph with n nodes, e arcs and p connected components is

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

McCabe's cyclomatic complexity is the most widely used graph theoretic complexity measure. He felt that complexity was best measured by the number of paths through a program. But since this number could be extremely large and hence impractical to compute, he defined complexity in terms of the number of linearly independent basic paths. Every path can be expressed as a combination of basic paths. He was motivated by the following

theorem from Berge [1]:

Theorem: In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits.

Given a program we can associate a flow graph with a unique entry node and a unique exit node. Note that if a flow graph has several exit nodes, we can add arcs from these nodes to a unique exit node. If we assume that each node of the control graph is reachable from the entry node (i.e. every program statement can be reached from the first statement), then by adding an arc from the exit node to the entry node the flow graph becomes strongly connected. Hence the theorem applies.

The cyclomatic number is simple to compute. It equals the number of program predicates (conditional or looping statements) plus one. Thus it is not necessary to construct the flow graph in order to compute the cyclomatic complexity.

Undoubtedly this ease of computation is one of the major reasons for the popularity of McCabe's measure. An early experiment in software complexity measures for large programs by Farr and Zagorsky [4] also concluded that the "IF" statement density was a useful and easily computed measure of logical complexity. Of the 93 candidate measures, their study concluded that the IF statement count had a significant weight in their most accurate formula.

After he analyzed many Fortran programs McCabe concluded that rather than limiting a module to a certain number of statements such as 50, a cyclomatic complexity of 10 was a reasonable upper limit.

The major limitation of the cyclomatic number as a complexity measure is that it is based entirely on one property of the program, the number of condition and looping statements. It totally ignores such things as nesting levels of these statements, program comments, choice of data structure, choice of variable names, etc. Certainly the number of predicates is only one aspect of complexity.

McCabe's results were based on a study of a large number of Fortran programs. It is not clear that the same is true for other programming languages.

2. CHEN'S PROGRAM COMPLEXITY

Chen [2] attempted to analyze programmer productivity as a function of program control complexity. His complexity measure took nesting of statements into account. He defined the maximal intersect number (MIN) on the regions of either the program graph or the flow graph of the program. A single arc whose removal disconnects the weakly connected graph is called a bridge. For each maximal subgraph that does not contain a bridge or is not weakly connected, draw a line from the arc entering its entry node to the arc leaving its exit node. Then the MIN of a strongly connected graph is the maximum number of arcs intersected by a line drawn from the outer region that enters all regions of the graph only once. See Figure 5.

For a weakly connected graph its MIN is equal to the sum of the MIN's of all its strongly connected subparts minus twice the number of subparts plus two. Thus

MIN = sum of MIN's of strongly connected subparts - 2 * number of subparts + 2

For the graph of Figure 6,

MIN = (4 + 5) - 2*(2) + 2 = 7

To show that the MIN takes nesting into account, Figure 7 shows two programs with two conditional statements. When the conditional statements are in sequence and not nested, the MIN is 2; whereas when the conditional statements are nested, the MIN is 3. In general if a program has n conditional statements, its MIN value ranges from n+1 (all conditionals are in a single nested structure) to 2 (the conditionals are in sequence with no nesting)

There is no natural program characteristic that correpsonds to the MIN. In fact, the MIN is not a graph parameter. It is only defined in terms of a graph. Probably the graph parameter that comes closest to the MIN is a type of maximum cut-set.

Chen's empirical data showed that programmer productivity (number of source statements produced) decreased as the complexity (MIN) of the program increased. However the MIN of a program is very tedious to compute.

3. Scope Metric

The Scope Metric [5] is computed from the flow graph. It also attempts to measure the nesting of the blocks of the program by measuring the scope of control of the control structure.

Every node of the graph is classified as either a selection node (outdegree 2 or more) or a receiving node (outdegree 1 or 0). To obtain the scope number create a subgraph G' consisting of all nodes immediately succeeding a given selection node. The subgraph G' will have at least one "lower bound", or node that is reachable via every path out of the subgraph. The greatest lower bound (GLB) is the lower bound of the subgraph that precedes every other lower bound of G'. The number of nodes preceding the GLB and succeeding the selection node plus one is the "adjusted complexity" of the selection node. The procedure is repeated for every selection node in the flow graph. The adjusted complexity of a receiving node is one. The Scope Number is the sum of the adjusted complexities of all nodes in the flow graph. See Figure 8.

It is claimed that the Scope Number reflects the level of nesting and the amount of processing done within the nested structure. A program with n binary decisions will have Scope Number that ranges from 3n+1 (decisions are serial) through 3(n(n+1)/2)+2n+1 (each decision node is nested under the previous decision node.)

Just as for McCabe's cyclomatic complexity, the Scope Number is based on the number of decision statements. It does take statement nesting into account, but does not consider the choice of algorithm, data structure, comments, meaningful variable names, etc. However, the Scope Number assigns complexities to control constructs based on their scope of effect within the

program rather than upon the construct alone. The Scope Number is easier to compute than the MIN, but more difficult than the cyclomatic complexity.

4. Knot Count

The knot count [10] is a measure of the unstructuredness of a program. Programmers frequently draw arrows in the left hand margin of a program listing as an aid in following the logic flow and branching in the program. This is especially true for Fortran and Basic programs. Intuitively, the knot count is the minimum number of intersections of these arrows.

Graph theoretically, the knot count can be defined from the program graph as folows: List the nodes of the program graph in order vertically on a page. Draw all of the arcs on one side of the vertical list of nodes. The knot count is the minimum number of arc crossings over all possible arrangements of the arcs. See Figure 9.

Cook [3] related the knot count to the overlap graph. Assign consecutive integers starting with 1 to the statements of the program. The overlap graph has a node for each transfer of control in the program to other than the natural successor statement, e.g. the next statement. Hence each node corresponds to the open interval of statements between the two statements involved in the transfer of control. There is an undirected edge in the overlap graph between two nodes whenever the corresponding open intervals have more than an endpoint in common and neither is properly contained in the other. Figure 10 gives the overlap graph of the program in Figure 9. The number of edges in the overlap graph is the knot count of the program.

The knot count depends on the order of the program statements. This is in sharp contrast to McCabe's cyclomatic measure which is totally independent of the statement ordering. Clearly the layout of the program impacts its readablility and understandability. The larger the knot count, more difficult it is to follow the program logic.

The knot count does not measure nesting, but it does measure unstructuredness. A program with n properly nested loops has a knot count of zero. A structured program does not have a knot count of zero as the if-then-else construct has one knot. However if we perform the usual reduction (primitive structured programming constructs replaced with a single node and removal of self loops), a structured program reduces to a single node with zero knots. After a program has been reduced as much as possible, the remaining knots are called essential knots. The essential knot count is a measure of unstructuredness.

The knot count is determined from the program text. Its best utilization is as an indictor of program readability. It is simple to compute especially using the overlap graph.

5. Henry and Kafura's Information Flow Complexity

This measure is quite different from the ones we have encountered thus far. It measures the interconnectivity of the program modules. Our previous metrics basically measured the

complexity of a single module and defined the complexity of a large program with subprograms as the sum of the complexities of the subprograms. Henry and Kafura [6] based their measure on the premise that the complexity of the interface between program modules and data flow are critical for program testing and maintenance.

The graph of the program is the call graph with additional nodes for the data structures. Arcs correspond to invocations of modules, data values passed to or between modules, and information deposited or retrieved from a data structure. The fan-in and fan-out are merely the indegree and outdegree respectivley. They defined the complexity of a module as

$$\text{length} * (\text{fan-in} * \text{fan-out})^2$$

where the length is some complexity measure of the module such as McCabe's cyclomatic complexity or the number of statements in the module.

They validated their measure by correlating it with data about changes in the UNIX system. These changes were mostly to correct errors. The modules with a high complexity had most of the changes. The advantage of their measure is that it can be applied at design time as well as to program code. Current techniques in data flow analysis can provide the information for the graphs automatically at compile time. Hence the computation is not that difficult.

Henry and Kafura feel that an information flow complexity measure is an appropriate and practical measure for large real world systems. It provides information that cannot be derived from simple single module based measures.

FURTHER WORK

As we have shown, a popular approach to measuring software complexity involves the analysis of some of the characteristics the program flow graph. Typically, these characteristics are some aspect of the flow of control. All of the metrics we have described concentrate on different characteristics. For example, McCabe's metric is based on the number of decisions in the program; Chen's metric is a function of how the decisions in the program are related to each other (e.g. nesting); and the Scope metric reflects both the number of decisions and how they are related. The knot count is also based on how decisions are related to each other, but rather than nesting it reflects program "unstructuredness". The information flow metric approaches complexity from an entirely different direction by considering of control flow betweem modules.

It is not clear what is the best approach or what most accurately measures the complexity of real programs. However, it appears that limiting the analysis to control flow characteristics disregards other factors that most programmers feel contribute to the ease or difficulty of working with a program. These other factors include commenting, choice of variable names, data structuring, etc. Two programs that are

identical except for comments and variable names have identical flow graphs, but could have quite different complexities. Also it is not clear that that complexity metrics provide a valid comparison between different programs. Are two quite different programs with the same complexity value equally difficult to work with? Is a program with a complexity value twice that of another program twice as difficult to work with? Ideally a metric should be sensitive to more than the number or nesting of decisions and should be fine enough so that rarely will two non-isomorphic flow graphs have the same complexity value. This leads us to conclude that it is impossible to expect a single integer value to accurately reflect the complexity of a program if the measure is based on the flow graph or another graph associated with the program. It seems more likely that a complexity metric measures only one aspect of complexity or the difficulty of performing a particular programming task. Thus, for example, McCabe's measure would seem to be a good indication of the difficulty of testing a program.

Three goals for further research in this area are:

1. Development of more sensitive metrics.

2. Establishment through empirical evaluation of the utility of existing metrics.

3. Relation of existing metrics to particular programming tasks.

CONCLUSIONS

There currently exist many metrics that purport to measure program complexity. A number of these are based on an analysis of the flow graph of the program. This approach has a major weakness in that the program flow graph only reflects the flow of control in the program. It does not reflect characteristics of easy-to-understand programs such as meaningful variable names, comments, indentation, etc.

Our understanding of program complexity is currently in its infancy so we are unable to evaluate the "goodness" of any metric based on what characteristics it measures. However, it is clear that current metrics lack sufficient sensitivity as they are based on only a few (usually one) program characteristics. Future research must determine the relevant parts of current metrics and expand the pool of potential candidate characteristics to be investigated.

REFERENCES

1. Berge, C. Graphs and Hypergraphs, North-Holland, Amsterdam, The Netherlands, 1973.

2. Chen, E., Program complexity and programmer productivity, IEEE-SE-2 No. 2, May 1978, pp. 187-194.

3. Cook, C., A Graph theoretic program complexity measure, Proceedings of First West Coast Conference on Graph Theory and Computing, 1978, pp. 109-124.

4. Farr, L. and Zagorski, H. S., "Quantitative analysis of programming cost factors: a progress report" in Frelink, A. B. (ed) Economics of Automatic Data Processing. ICC Symposium Proceedings 1965 Rome. North-Holland, Amsterdam, 1965.

5. Harrison, W. and Magel, K., A Complexity measure based on nesting level, ACM SIGPLAN Notices, March 1981, pp. 63-74.

6. Henry S. and D. Kafura, Software structure metrics based on information flow, IEEE-SE-7, No. 5, Sept. 1981, pp. 510-518.

7. McCabe, T., A complexity measure, IEEE-SE-2 No. 4, Dec. 1976, pp. 309-321.

8. Paige, M. R., Program graphs, an algebra and their implications for programming, IEEE-SE-1, No. 3, Sept. 1975, pp. 286-291.

9. Ryder, B., Constructing the call graph of a program, IEEE-SE-5 No. 3, May 1979, pp. 216-226.

10. Woodward, M., Hennell, M. and Hedley, D., A Measure of control flow complexity in program text, IEEE-SE-5 No. 1, Jan. 1979, pp. 45-50.

```
1              SUBROUTINE BUBBLE(A,N)
2              DO I = 2,N
3                IF (A(I) .GE. A(I-1)) GOTO 200
4                J = I
5      100       IF (J .LE 1) GOTO 200
6                IF (A(J) .GE. A(J-1)) GOTO 200
7                ITEMP = A(J)
8                A(J) = A(J-1)
9                A(J-1) = ITEMP
10               J = J-1
11               GOTO 100
12     200     CONTINUE
13             RETURN
14             END
```

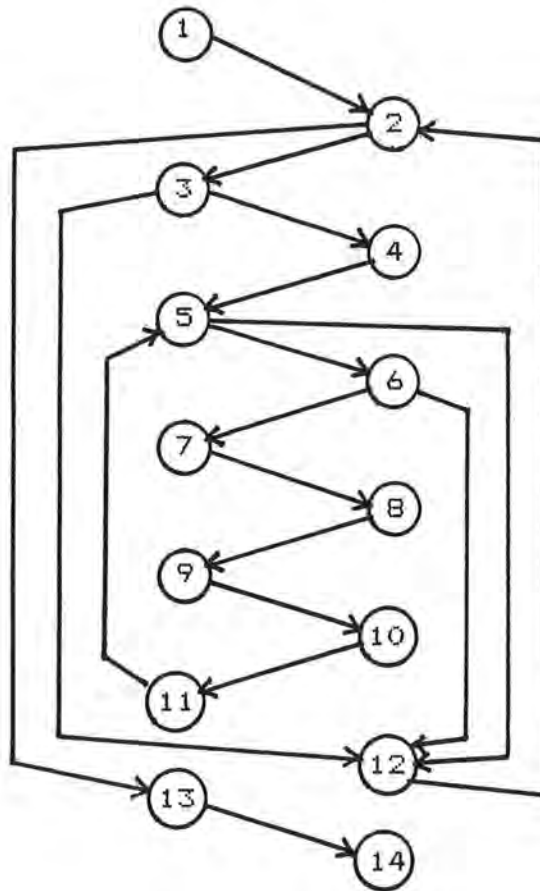


FIGURE 1. Fortran Bubble Sort Program and its Program Graph.

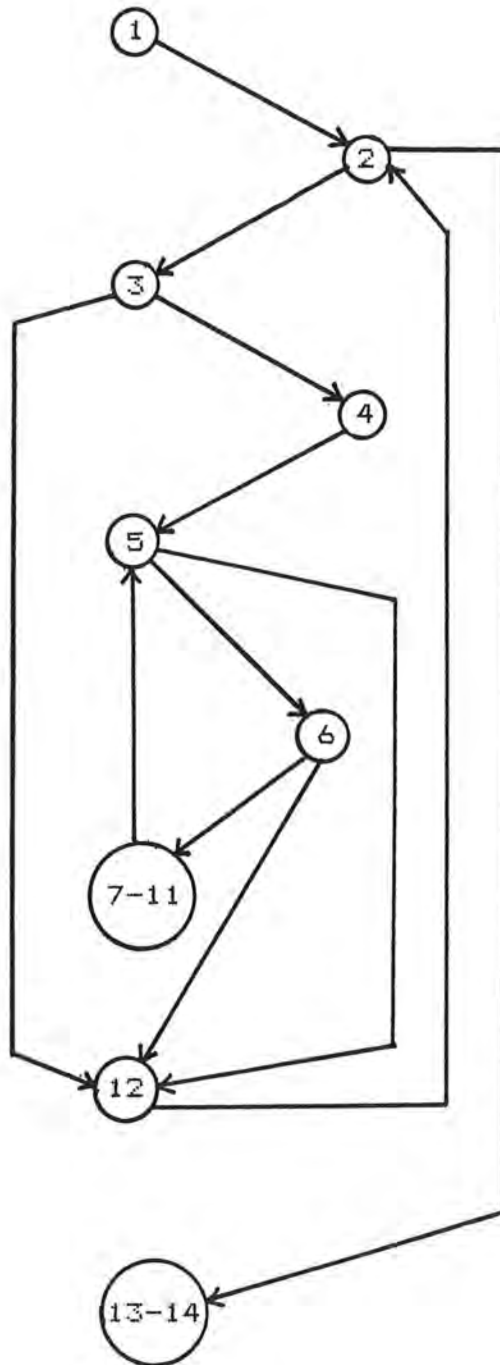


Figure 2. Flow graph of Fortran Program in Figure 1.

```
PROGRAM MAIN
CALL SUB1
CALL SUB2
CALL SUB1
CALL SUB3
END

SUBPROGRAM SUB1
CALL SUB3
CALL SUB2
CALL SUB3
END

SUBPROGRAM SUB2
CALL SUB1
CALL SUB3
END

SUBPROGRAM SUB3
CALL SUB2
CALL SUB4
CALL SUB1
END

SUBPROGRAM SUB4
CALL SUB3
END
```

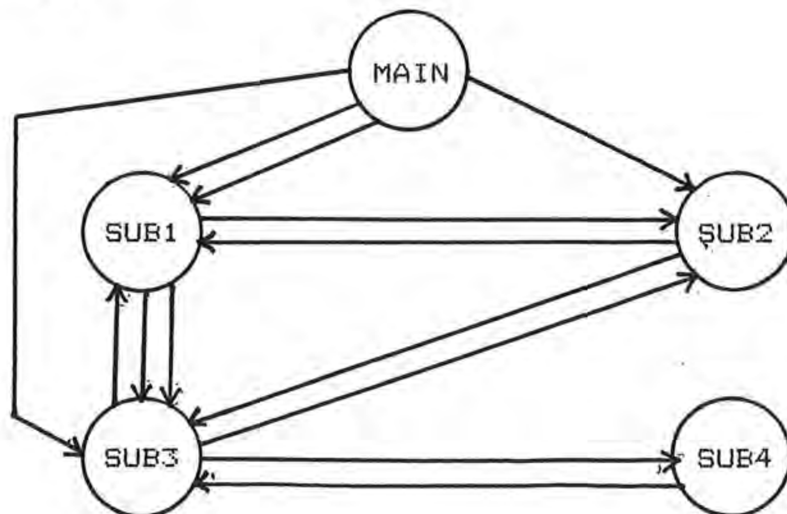
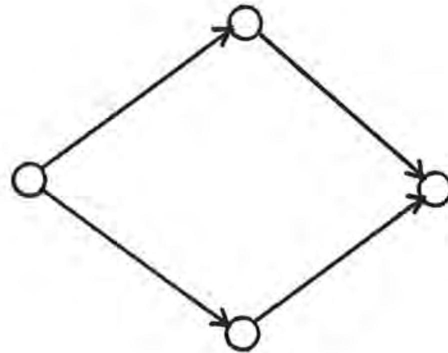


Figure 3. Example program and call graph.

(a) Sequence

(b) if-then-else



(c) do-while

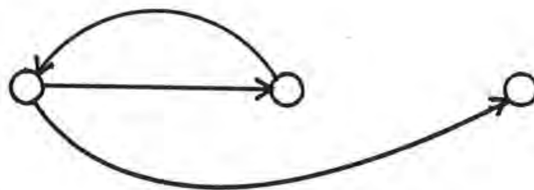
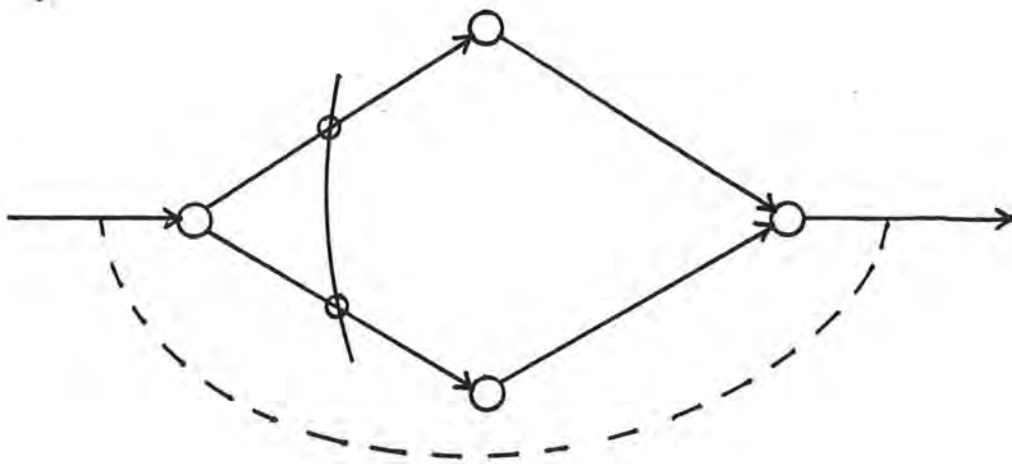
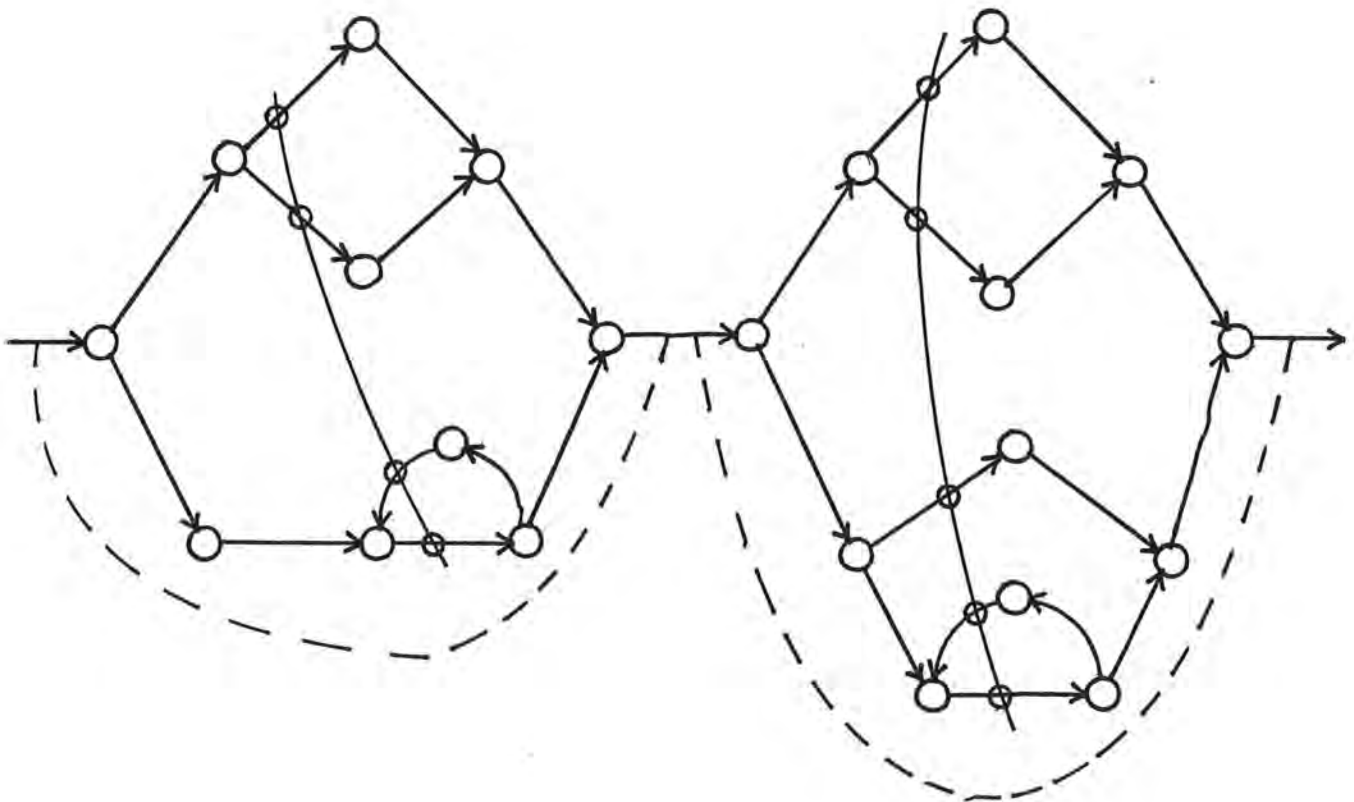


Figure 4. Three structured programming building blocks



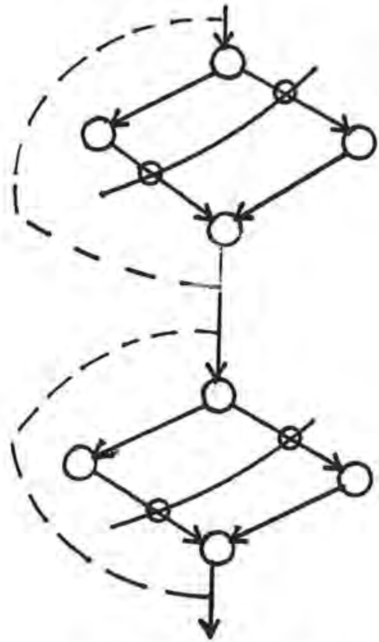
MIN = 2

Figure 5. Example of MIN

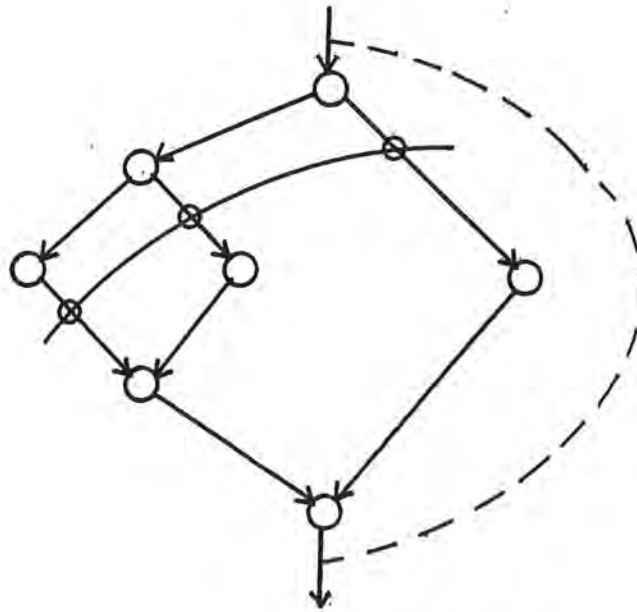


MIN = 4 + 5 - 2 * (2) + 2 = 7

Figure 6. MIN as sum of subparts

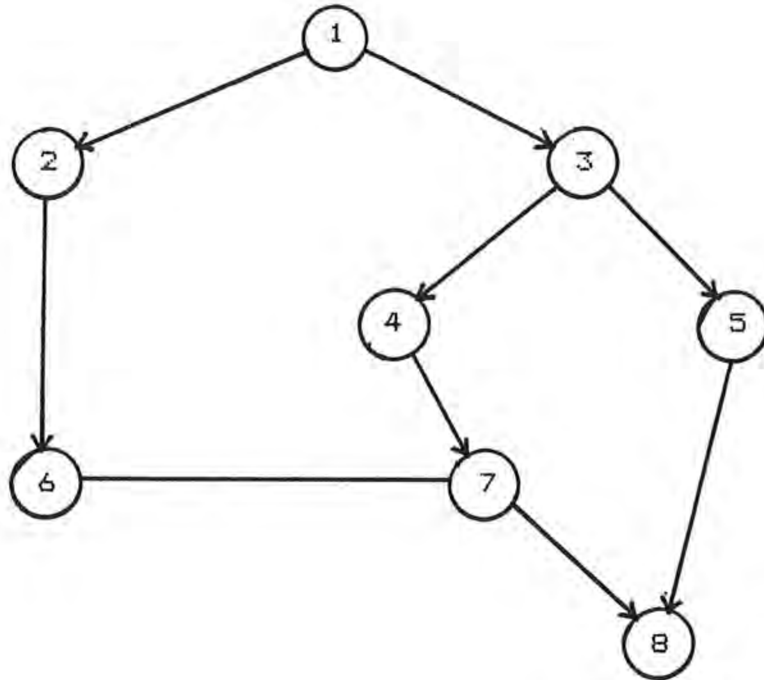


MIN = 4 - 2 * (2) + 2 = 2



MIN = 3 - 2 + 2 = 3

Figure 7. MIN of nested and non-nested conditional statements.



Adjusted Complexity

Node 1	7
Node 2	1
Node 3	4
Node 4	1
Node 5	1
Node 6	1
Node 7	1
Node 8	1

Scope Number = 17

Figure 8. Example of flow graph and its scope number.

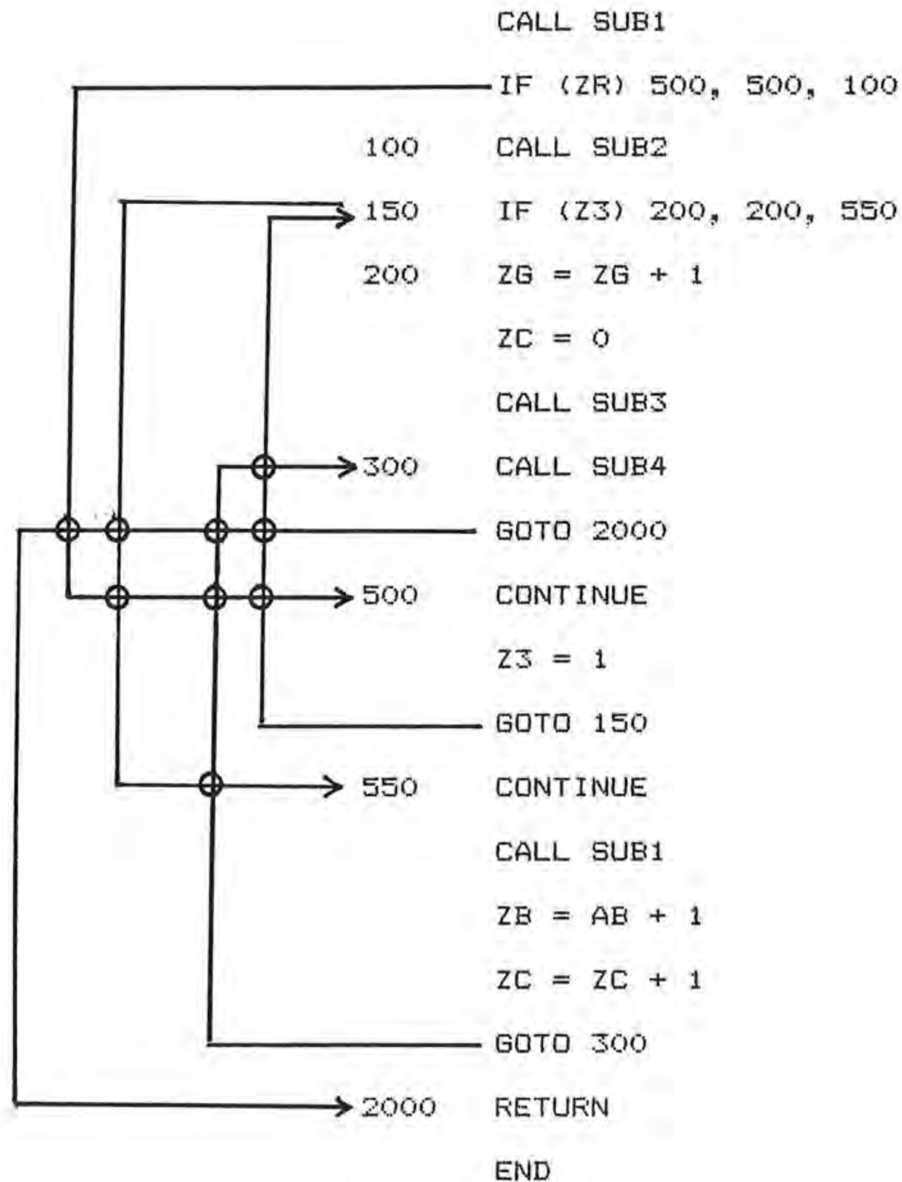


Figure 9. Program with 9 knots

Line Numbers		
1		CALL SUB1
2		IF (ZR) 500, 500, 100
3	100	CALL SUB2
4	150	IF (Z3) 200, 200, 550
5	200	ZG = ZG + 1
6		ZC = 0
7		CALL SUB3
8	300	CALL SUB4
9		GOTO 2000
10	500	CONTINUE
11		Z3 = 1
12		GOTO 150
13	550	CONTINUE
14		CALL SUB1
15		ZB = ZB + 1
16		ZC = ZC + 1
17		GOTO 300
18	2000	RETURN
19		END

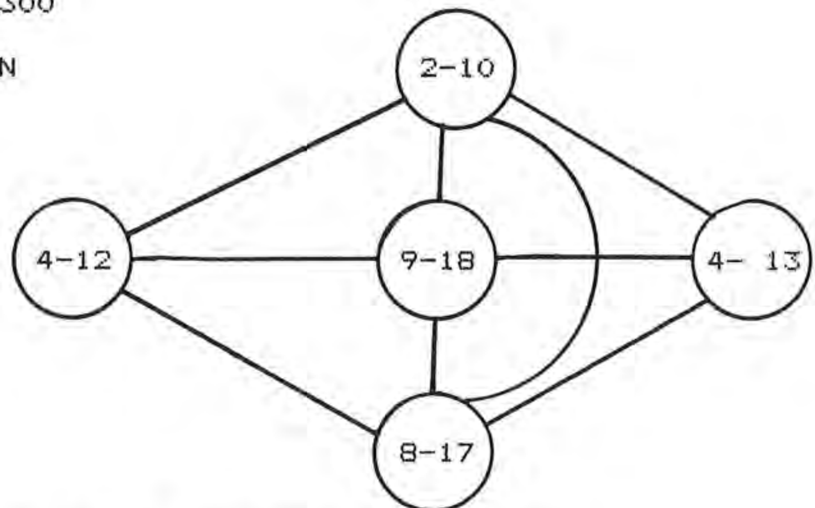


Figure 10. Program and its overlap graph