A Complexity Measure

THOMAS J. McCABE

Abstract—This paper describes a graph-theoretic complexity measure and illustrates how it can be used to manage and control program complexity. The paper first explains how the graph-theory concepts apply and gives an intuitive explanation of the graph concepts in programming terms. The control graphs of several actual Fortran programs are then presented to illustrate the correlation between intuitive complexity and the graph-theoretic complexity. Several properties of the graph-theoretic complexity are then proved which show, for example, that complexity is independent of physical size (adding or subtracting functional statements leaves complexity unchanged) and complexity depends only on the decision structure of a program.

The issue of using nonstructured control flow is also discussed. A characterization of nonstructured control graphs is given and a method of measuring the "structuredness" of a program is developed. The relationship between structure and reducibility is illustrated with several examples.

The last section of this paper deals with a testing methodology used in conjunction with the complexity measure; a testing strategy is defined that dictates that a program can either admit of a certain minimal testing level or the program can be structurally reduced.

Index Terms—Basis, complexity measure, control flow, decomposition, graph theory, independence, linear, modularization, programming, reduction, software, testing.

I. Introduction

THERE is a critical question facing software engineering today: How to modularize a software system so the resulting modules are both testable and maintainable? That the issues of testability and maintainability are important is borne out by the fact that we often spend half of the development time in testing [2] and can spend most of our dollars maintaining systems [3]. What is needed is a mathematical technique that will provide a quantitative basis for modularization and allow us to identify software modules that will be difficult to test or maintain. This paper reports on an effort to develop such a mathematical technique which is based on program control flow.

One currently used practice that attempts to ensure a reasonable modularization is to limit programs by physical size (e.g., IBM-50 lines, TRW-2 pages). This technique is not adequate, which can be demonstrated by imagining a 50 line program consisting of 25 consecutive "IF THEN" constructs. Such a program could have as many as 33.5 million distinct control paths, only a small percentage of which would probably ever be tested. Many such examples of live Fortran programs that are physically small but untestable have been identified and analyzed by the tools described in this paper.

Manuscript received April 10, 1976.

The author is with the Department of Defense, National Security Agency, Ft. Meade, MD 20755.

II. A Complexity Measure

In this section a mathematical technique for program modularization will be developed. A few definitions and theorems from graph theory will be needed, but several examples will be presented in order to illustrate the applications of the technique.

The complexity measure approach we will take is to measure and control the number of paths through a program. This approach, however, immediately raises the following nasty problem: "Any program with a backward branch potentially has an infinite number of paths." Although it is possible to define a set of algebraic expressions that give the total number of possible paths through a (structured) program,[1] using the total number of paths has been found to be impractical. Because of this the complexity measure developed here is defined in terms of basic paths—that when taken in combination will generate every possible path.

The following mathematical preliminaries will be needed, all of which can be found in Berge [1].

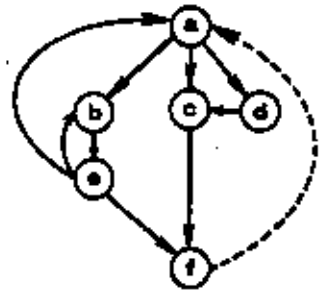
Definition 1: The cyclomatic number V(G) of a graph G with n vertices, e edges, and p connected components is

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

Theorem 1: In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits.

The applications of the above theorem will be made as follows: Given a program we will associate with it a directed graph that has unique entry and exit nodes. Each node in the graph corresponds to a block of code in the program where the flow is sequential and the arcs correspond to branches taken in the program. This graph is classically known as the program control graph (see Ledgard [6]) and it is assumed that each node can be reached by the entry node and each node can reach the exit node. For example, the following is a program control graph with entry node "a" and exit node "f."

G:

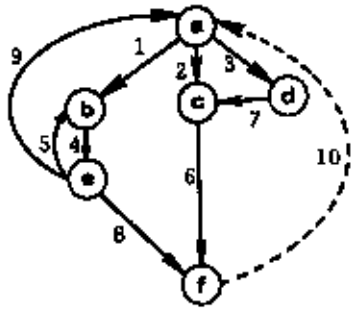


[1] See the Appendix.

Theorem 1 is applied to G in the following way. Imagine that the exit node (f) branches back to the entry node (a). The control graph G is now strongly connected (there is a path joining any pair of arbitrary distinct vertices) so Theorem 1 applies. Therefore, the maximum number of linearly independent circuits in G is 9−6+2. For example, one could choose the following 5 independent circuits in G:

B1: (abefa), (beb), (abea), (acfa), (adcfa).

It follows that B1 forms a basis for the set of all circuits in G and any path through G can be expressed as a linear combination of circuits from B1. For instance, the path (abeabebebef) is expressable as (abea) +2(beb) + (abefa). To see how this works its necessary to number the edges on G as in



Now for each member of the basis B1 associate a vector as follows:

	1	2	3	4	5	6	7	8	9	10
(abefa)	1	0	0	1	0	0	0	1	0	1
(beb)	0	0	0	1	1	0	0	0	0	0
(abea)	1	0	0	1	0	0	0	0	0	0
(acfa)	0	1	0	0	0	1	0	0	0	1
(adcfa)	0	0	1	0	0	1	1	0	0	1

The path (abea(be)^{3}fa) corresponds to the vector 2004200111 and the vector addition of (abefa), 2(beb), and (abea) yields the desired result.

In using Theorem 1 one can choose a basis set of circuits that correspond to paths through the program. The set B2 is a basis of program paths.

B2: (abef), (abeabef), (abebef), (acf), (adcf).

Linear combination of paths in B2 will also generate any path. For example,

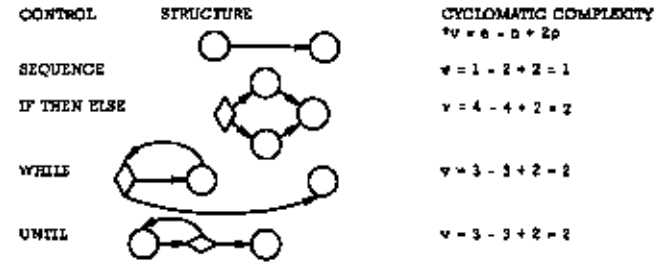
$$(abea(be)^3f) = 2(abebef) - (abef)$$

and

$$(a(be)^2abef) = (a(be)^2f) + (abeabef) - (abef).$$

The overall strategy will be to measure the complexity of a program by computing the number of linearly independent paths v(G), control the "size" of programs by setting an upper limit to v(G) (instead of using just physical size), and use the cyclomatic complexity as the basis for a testing methodology.

A few simple examples may help to illustrate. Below are the control graphs of the usual constructs used in structured programming and their respective complexities.



Notice that the sequence of an arbitrary number of nodes always has unit complexity and that cyclomatic complexity conforms to our intuitive notion of "minimum number of paths." Several properties of cyclomatic complexity are stated below:

1) v(G)≥1.
2) v(G) is the maximum number of linearly 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.

III. Working Experience with the Complexity Measure

In this section a system which automates the complexity measure will be described. The control structures of several PDP-10 Fortran programs and their corresponding complexity measures will be illustrated.

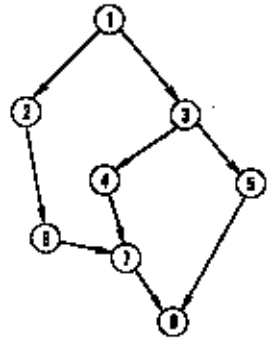
To aid the author's research into control structure complexity a tool was built to run on a PDP-10 that analyzes the structure of Fortran programs. The tool, FLOW, was written in APL to input the source code from Fortran files on disk. FLOW would then break a Fortran job into distinct subroutines and analyze the control structure of each subroutine. It does this by breaking the Fortran subroutines into blocks that are delimited by statements that affect control flow: IF, GOTO, referenced LABELS, DO, etc. The flow between the blocks is then represented in an n by n matrix (where n is the number of blocks), having a 1 in the i-jth position if block i can branch to block j in 1 step. FLOW also produces the "blocked" listing of the original program, computes the cyclomatic complexity, and produces a reachability matrix (there is a 1 in the i-jth position if block i can branch to block j in any number of steps). An example of FLOW's output is shown below.

```
      IMPLICIT INTEGER(A-Z)
      COMMON / ALLOC / MEM(2048),LM,LU,LF,LW,LX,LY,LQ,LWEK,
     1   NCHARS,NWORDS
      DIMENSION MEMORY(2048),INHEAD(4),ITRANS(128)
      TYPE 1
    1 FORMAT(@COMOLKI STRUCTURE FILE NAME?@)
      NAMDML=0
      ACCEPT 2,NAMDML
    2 FORMAT(A5)
      CALL ALCHAN(ICHAN)
      CALL IFILE(ICHAN,'DSK',NAMDML,'DAT',0,0)
      CALL READB(ICHAN,INHEAD,132,NREAD,8980,8990)
      NCHARS=INHEAD(1)
      NWORDS=INHEAD(2)
```

*The role of the variable p will be explained in Section IV. For these examples assume p = 1.

```
      NTOT=(NCHARS+7)*NWORDS
      LTOT=(NCHARS+5)*NWORDS
****** BLOCK NO. 1     ****************************
      IF(LTOT.GT.2048) GO TO 900
******* BLOCK NO. 2     ***************************
      CALL READB(ICHAN,MEMORY,LTOT,NREAD,8990,8990)
      LH=0
      LU=NCHARS*NWORDS+LH
      LV=NWORDS+LU
      LW=NWORDS+LV
      LX=NWORDS+LW
      LY=NWORDS+LX
      LQ=NWORDS+LY
      LNEX=NWORDS+LQ
         BLOCK NO. 3     ***************************
******** DO 700 I=1,NWORDS
      MEMORY(LNEX+I)=(MEMORY(LW+I).OR.(MEMORY(LW+I)*2))
  700 CONTINUE
******** BLOCK NO. 4     ***************************
      CALL EXTENT(ITRANS)
      STOP
******** BLOCK NO. 5     ***************************
  900 TYPE 3,LTOT
    3 FORMAT('STRUCTURE TOO LARGE FOR CORE; ',18,' WORDS' /
        ' SEE COOPER       ' /)
      STOP
******** BLOCK NO. 6     ***************************
  990 TYPE 8
    8 FORMAT(' READ ERROR, OR STRUCTURE FILE ERROR; ' /
        ' SEE COOPER       ' /)
      STOP
      END
```

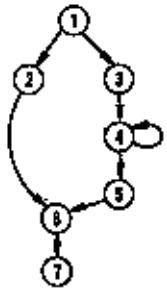
V(G) = 2

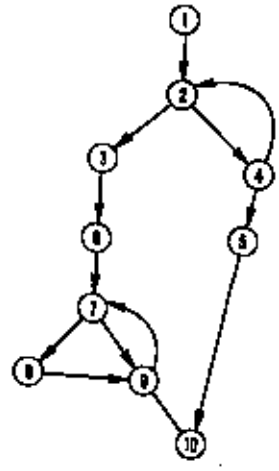


V(G) = 3

CONNECTIVITY MATRIX

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





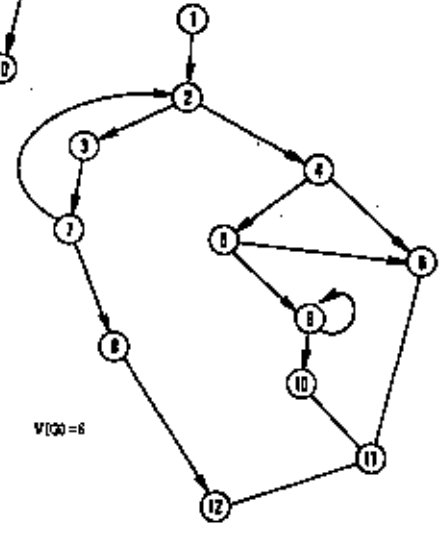
V(G) = 5

,DL,DL,DL,DL,DL,DL,DL,DL,DL,DL,DL,DL,DL,DL CYCLOMATIC COMPLEXITY = 3

CLOSURE OF CONNECTIVITY MATRIX

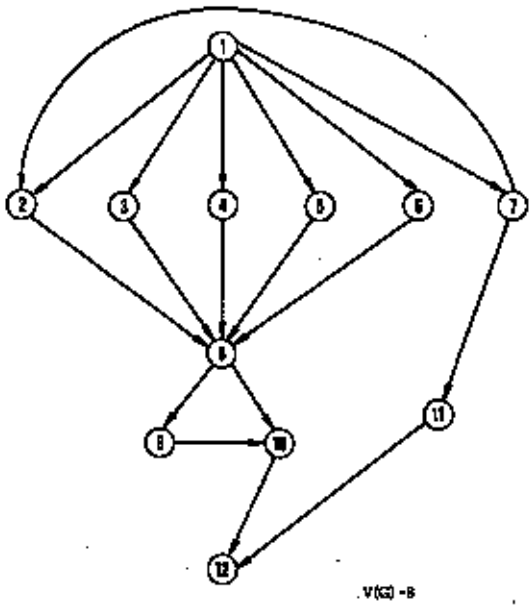
	1	2	3	4	5	6	7
1	0	1	1	1	1	1	1
2	0	0	0	0	0	1	1
3	0	0	0	1	1	1	1
4	0	0	0	1	1	1	1
5	0	0	0	0	0	1	1
6	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0

,END

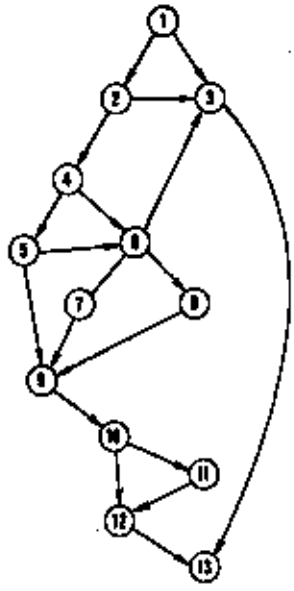


V(G) = 6

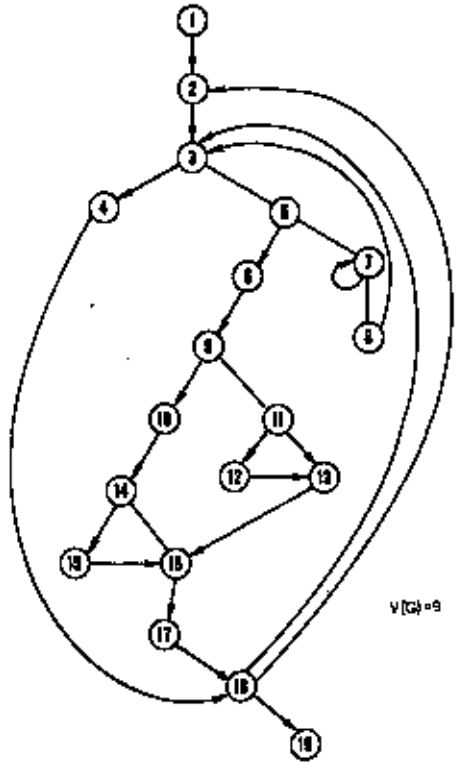
At this point a few of the control graphs that were found in live programs will be presented. The actual control graphs from FLOW appear on a DATA DISK CRT but they are hand drawn here for purposes of illustration. The graphs are presented in increasing order of complexity in order to suggest the correlation between the complexity numbers and our intuitive notion of control flow complexity.


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

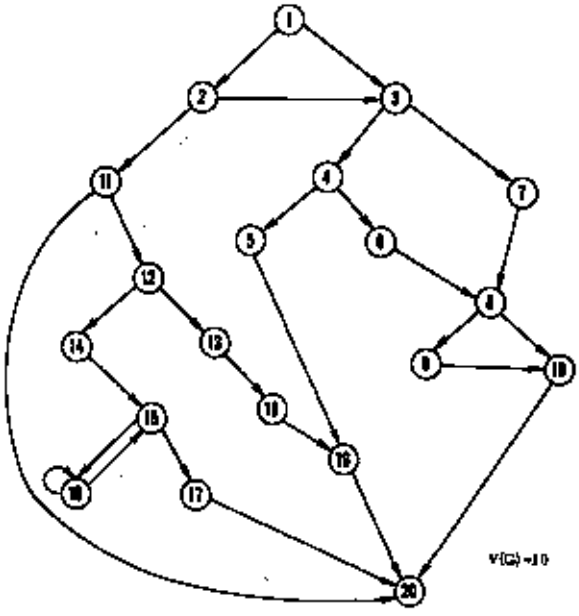
V(G) = 8

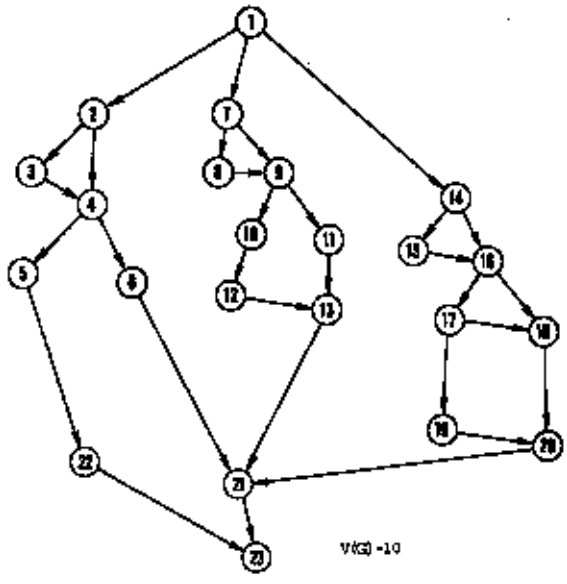

1
2
3
4
5
6
7
8
9
10
11
12
13

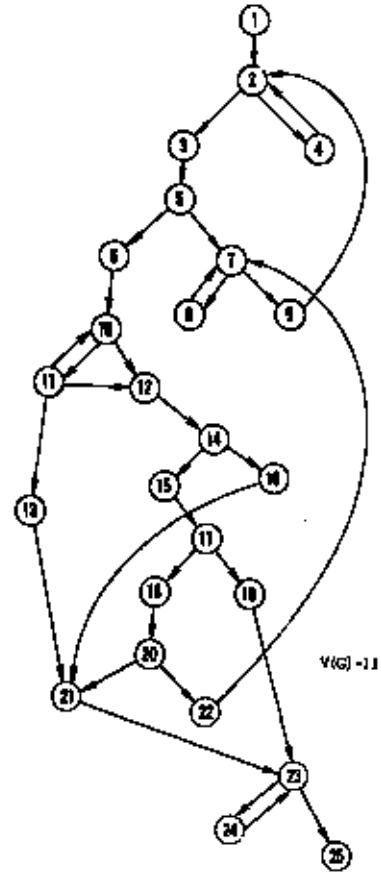
V(G) = 8


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

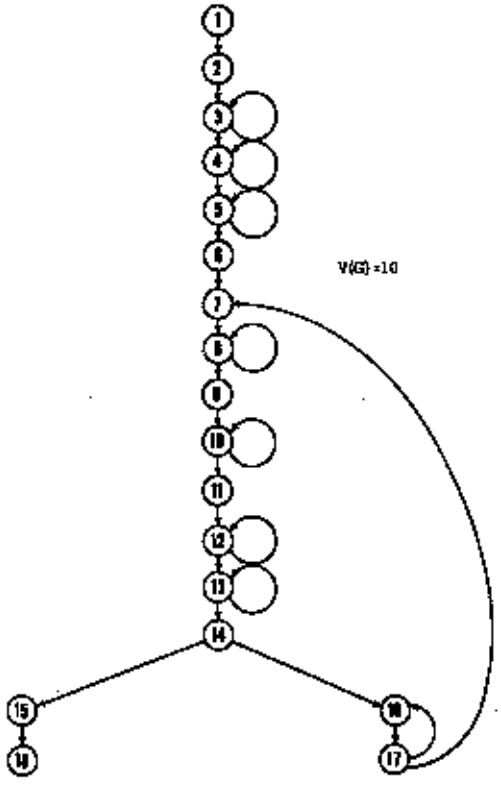
V(G) = 9

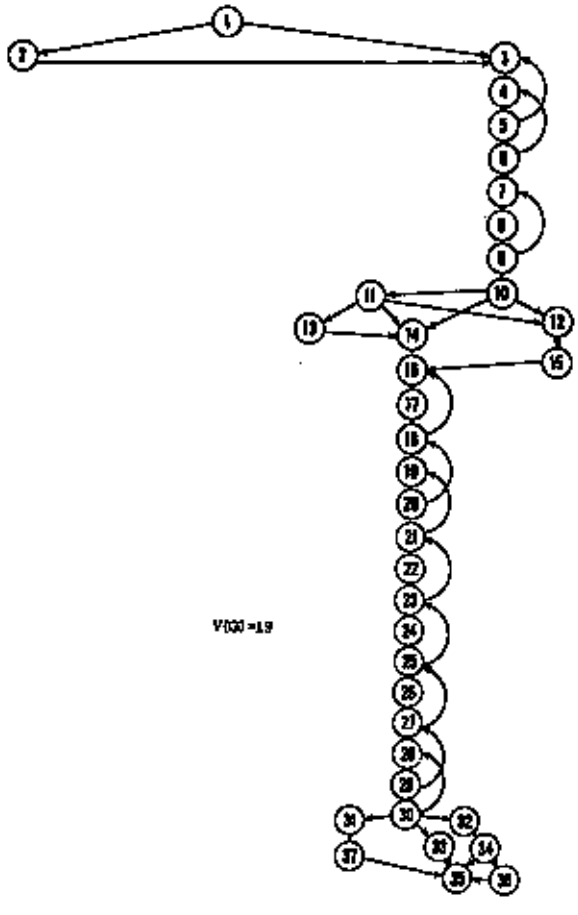

V(G) =10


V(G) =10


V(G) =11

One of the more interesting aspects of the automatic approach is that although FLOW could be implemented much more efficiently in a compiler level language, it is still possible to go through a year's worth of a programmer's Fortran code in about 20 min. After seeing several of a programmer's control graphs on a CRT one can often recognize "style" by noting similar patterns in the graphs. For example, one programmer had an affinity for sequencing numerous simple loops as in





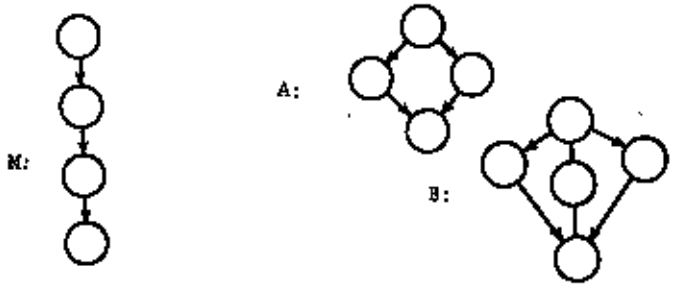
It was later revealed that these programs were eventually to run on a CDC6600 and the "tight" loops were designed to stay within the hardware stack.

These results have been used in an operational environment by advising project members to limit their software modules by cyclomatic complexity instead of physical size. The particular upper bound that has been used for cyclomatic complexity is 10 which seems like a reasonable, but not magical, upper limit. Programmers have been required to calculate complexity as they create software modules. When the complexity exceeded 10 they had to either recognize and modularize subfunctions or redo the software. The intention was to keep the "size" of the modules manageable and allow for testing all the independent paths (which will be elaborated upon in Section VII.) The only situation in which this limit has seemed unreasonable is when a large number of independent cases followed a selection function (a large case statement), which was allowed.

It has been interesting to note how individual programmer's style relates to the complexity measure. The author has been delighted to find several programmers who never had formal training in structured programming but consistently write code in the 3 to 7 complexity range which is quite well structured. On the other hand, FLOW has found several programmers who frequently wrote code in the 40 to 50 complexity range (and who claimed there was no other way to do it). On one occasion the author was given a DEC tape of 24 Fortran subroutines that were part of a large real-time graphics system. It was rather disquieting to find, in a system where reliability is critical, subroutines of the following complexity: 16, 17, 24, 24, 32, 34, 41, 54, 56, and 64. After confronting the project members with these results the author was told that the subroutines on the DEC tape were chosen because they were troublesome and indeed a close correlation was found between the ranking of subroutines by complexity and a ranking by reliability (performed by the project members).

IV. Decomposition

The role of p in the complexity calculation $v = e - n + 2p$ will now be explained. Recall in Definition 1 that p is the number of connected components. The way we defined a program control graph (unique entry and exit nodes, all nodes reachable from the entry, and the exit reachable from all nodes) would result in all control graphs having only one connected component. One could, however, imagine a main program M and two called subroutines A and B having a control structure shown below:



Let us denote the total graph above with 3 connected components[2] as MUAUB. Now, since $p = 3$ we calculate complexity as

$$v(M \cup A \cup B) = e - n + 2p = 13 - 13 + 2 \times 3 = 6.$$

This method with $p \neq 1$ can be used to calculate the complexity of a collection of programs, particularly a hierarchial nest of subroutines as shown above.

Notice that $v(M \cup A \cup B) = v(M) + v(A) + v(B) = 6$. In general, the complexity of a collection C of control graphs with k connected components is equal to the summation of their complexities. To see this let C_i, $1 \leq i \leq k$ denote the k distinct connected components, and let e_i and n_i be the number of edges and nodes in the ith connected component. Then

$$v(C) = e - n + 2p = \sum_{i}^{k} e_i - \sum_{i}^{k} n_i + 2k$$

$$= \sum_{i}^{k} (e_i - n_i + 2) = \sum_{i}^{k} v(C_i).$$

V. Simplification

Since the calculation $v = e - n + 2p$ can be quite tedious for a programmer an effort has been made to simplify the complexity calculations (for single-component graphs). There are two results presented in this section—the first allows the complexity calculations to be done in terms of program syntactic constructs, the second permits an easier calculation from the graph form.

In [7] Mills proves the following: if the number of function, predicate, and collecting nodes in a structured program is θ, π, and γ, respectively, and e is the number of edges, then

$$e = 1 + \theta + 3\pi.$$

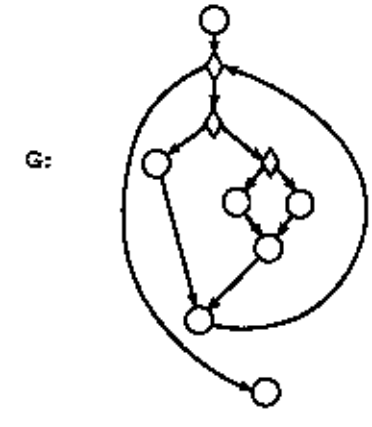
Since for every predicate node there is exactly one collecting node and there are unique entry and exit nodes it follows that

$$n = \theta + 2\pi + 2.$$

Assuming $p = 1$ and substituting in $v = e - n + 2$ we get

$$v = (1 + \theta + 3\pi) - (\theta + 2\pi + 2) + 2 = \pi + 1.$$

This proves that the cyclomatic complexity of a structured program equals the number of predicates plus one, for example in



[2] A graph is connected if for every pair of vertices there is a chain going from one to the other. Given a vertex a, the set of vertices that can be connected to a, together with a itself is a connected component.

complexity $v(G) = \pi + 1 = 3 + 1 = 4$. Notice how in this case complexity can be computed by simply counting the number of predicates in the code and not having to deal with the control graph.

In practice compound predicates such as IF "C1 AND C2" THEN are treated as contributing two to complexity since without the connective AND we would have

IF C1 THEN IF C2 THEN

which has two predicates. For this reason and for testing purposes it has been found to be more convenient to count conditions instead of predicates when calculating complexity.[3]

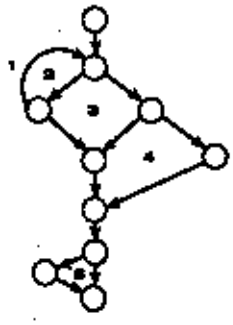
It has been proved that in general the complexity of any (unstructured) program is $\pi + 1$.

The second simplification of the calculation of $e - n + 2p$ reduces the calculation of visual inspection of the control graph. We need Euler's formula which is as follows. If G is a connected plane graph with n vertices, e edges, and r regions, then

$$n - e + r = 2.$$

Just changing the order of the terms we get $r = e - n + 2$ so the number of regions is equal to the cyclomatic complexity. Given a program with a plane control graph one can therefore calculate v by counting regions, as in

G:



v(G) = 5

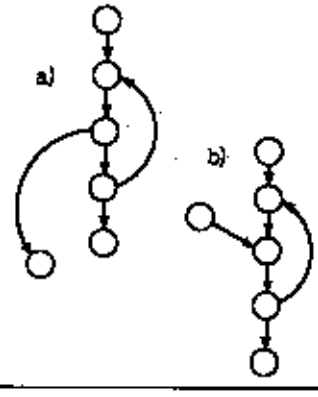
VI. Nonstructured Programming

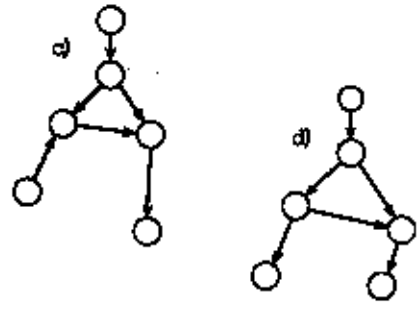
The main thrust in the recent popularization of structured programming is to make programmers aware of a few syntactic constructs[4] and tell them that a structured program is one written with only these constructs. One of the difficulties with this approach is it does not define for programmers what constructs they should not use, i.e., it does not tell them what a structured program is not. If the programming population had a notion of what constructs to avoid and they could see the inherent difficulty in these constructs, perhaps the notion of structuring programming would be more psychologically palatable. A clear definition of the constructs that structured programming excludes would also sensitize programmers to their use while programs are being created, which (if we believe in structured programming) would have a desirable effect.

One of the reasons that the author thinks this is important is that as Knuth [4] points out—there is a time and a place when an unstructured goto is needed. The author has had a similar experience structuring Fortran jobs—there are a few very specific conditions when an unstructured construct works best. If it is the case that unstructured constructs should only be allowed under special circumstances, one need then to distinguish between the programmer that makes judicious use of a few unstructured goto's as compared to the programmer that is addicted to them. What would help is first the definition of the unstructured components and second a measure of the structureness of a program as well as the complexity of a program.

Rao Kasaraju [5] has a result which is related—a flow graph is reducible[5] to a structured program if and only if it does not contain a loop with two or more exits. This is a deep result but not, however, what we need since many programs that are reducible to structured programs are not structured programs. In order to have programmers explicitly identify and avoid unstructured code we need a theorem that is analogous to a theorem like Kuratowski's theorem in graph theory. Kuratowski's theorem states that any nonplanar graph must contain at least one of two specific nonplanar graphs that he describes. The proof of nonplanarity of a graph is then reducible to locating two specific subgraphs whereas showing nonplanarity without Kuratowski's result is, in general, much more difficult.

The following four control structures were found to generate all nonstructured programs.



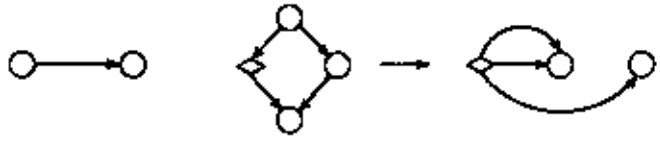


A number of theorems and results will be stated below without proof.

Result 1: A necessary and sufficient condition that a program[6] is nonstructured (one that is not written with just

[3] For the CASE construct with N cases use N-1 for the number of conditions. Notice, once again, that a simulation of case with IF's will have N-1 conditions.

[4] The usual ones used (sometimes called D-structures) are



[5] Reducibility here means the same function is computed with the same actions and predicates although the control structure may differ.

[6] Assuming the program does not contain unconditional GOTO's.

D-structures) is that it contains as a subgraph either a), b), or c).

The reason why graph d) was slighted in Result 1 is that any 3 of the 4 graphs will generate all the unstructured programs—this will be illustrated later. It is convenient to verbalize the graphs a)–d), respectively, as follows:

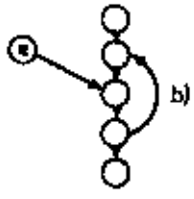
a) branching out of a loop;
b) branching into a loop;
c) branching into a decision; and
d) branching out of a decision.

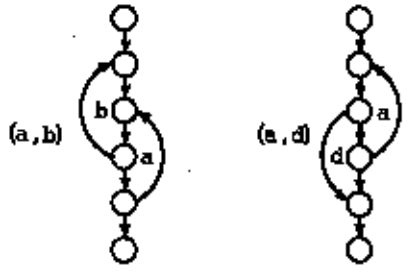
The following version of Result 2 may seem more intuitively appealing.

A structured program can be written by not "branching out of or into a loop, or out of or into a decision."

The following result gives insight into why a nonstructured program's logic often becomes convoluted and entwined.

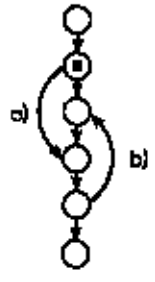
Result 2: A nonstructured program cannot be just a little nonstructured. That is any nonstructured program must contain at least 2 of the graphs a)–d). Part of the proof of Result 2 will be shown here because it helps to illustrate how the control flow in a nonstructured program becomes entangled. We show, for an example, how graph b) cannot occur alone. Assuming we have graph b):





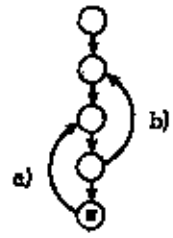
the entry node E occurs either before, after, or from a node independent of the loop. Each of these three cases will be treated separately.

Case 1: E is "before" the loop E is on a path from entry to the loop so the program must have a graph as follows:



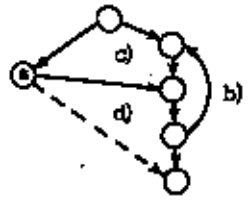
Notice how E is a split node at the beginning of a decision that is branched into. So in this case we have a c) graph along with the original b) graph.

Case 2: E is "after" the loop. The control graph would appear as follows:



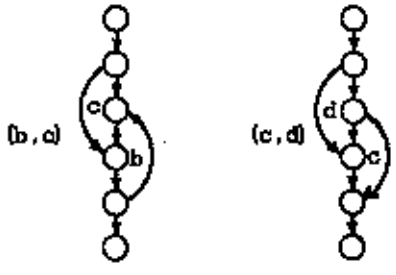
Notice how a type a) graph must appear.

Case 3: E is independent of the loop. The control graph would look as follows:



The graph c) must now be present with b). If there is another path that can go to a node after the loop from E then a type d) graph is also generated. Things are often this bad, and in fact much worse.

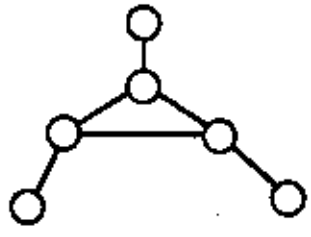
Similar arguments can be made for each of the other nonstructured graphs to show that a)–d) cannot occur alone. If one generates all the possible pairs from a)–d) it is interesting to note that they all reduce to 4 basic types:



which leads us the following result.

Result 3: A necessary and sufficient condition for a program to be nonstructured is that it contains at least one of: (a, b), (a, d), (b, c), (c, d). Result 4 is now obvious.

Result 4: The cyclomatic complexity if a nonstructured program is at least 3. It is interesting to notice that when the orientation is taken off the edges each of the 4 basic graphs a)–d) are isomorphic to the following nondirected graph.



Also if the graphs (a, b) through (c, d) have their directions taken off they are all isomorphic to:

By examining the graphs (a, b) through (c, d) one can formulate a more elegant nonstructured characterization:

Result 5: A structured program can be written by not branching out of loops or into decisions—a) and d) provide a basis.

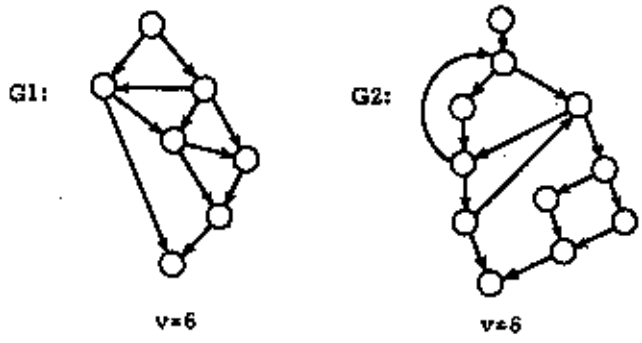
Result 6: A structured program can be written by not branching into loops or out of decisions—b) and d) provide a basis.

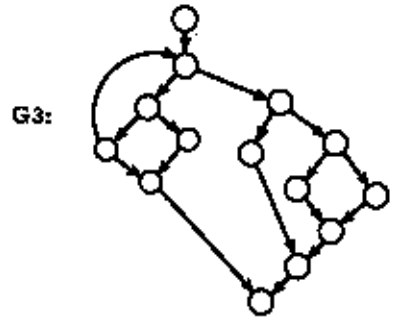
A way to measure the lack of structure in a program or flow graph will be briefly commented upon. One of the difficulties with the nonstructured graphs mentioned above is that there is no way they can be broken down into subgraphs with one entry and one exit. This is a severe limitation since one way in which program complexity can be controlled is to recognize when the cyclomatic complexity becomes too large—and then identify and remove subgraphs with unique entry and exit nodes.

Result 7: A structured program is reducible[7] to a program of unit complexity.

The following example illustrates how a structured program can be reduced.

Notice in the nonstructured graphs below, however, that such a reduction process is not possible.

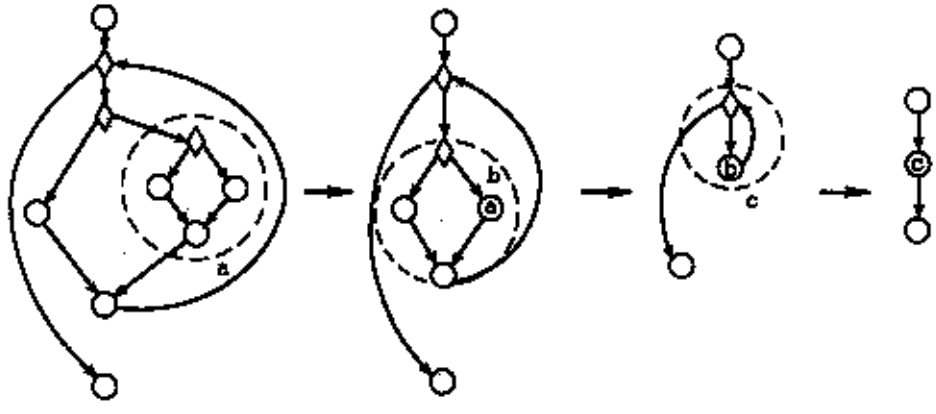




Let m be the number of proper subgraphs with unique entry and exit nodes. Notice in G1, G2, and G3 m is equal to 0, 1, and 2, respectively. The following definition of essential complexity ev is used to reflect the lack of structure.

Definition: ev = v - m.

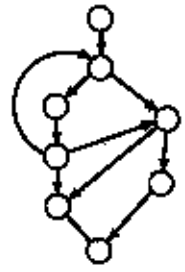
For the above graphs we have ev(G1) = 6, ev(G2) = 5, and ev(G3) = 4. Notice how the essential complexity indicates the



[7]Reduction is the process of removing subgraphs (subroutines) with unique entry and exit nodes.

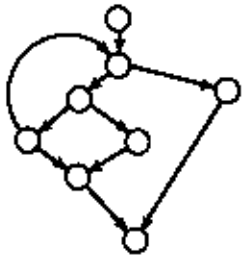
extent to which a graph can be reduced—G1 cannot be reduced at all since its complexity is equal to its essential complexity. G2 and G3, however, can be reduced as follows:

G2':



v(G2')=5

G3':



v(G3')=4

This last result is stated for completeness.

Result 8: The essential complexity of a structured program is one.

VII. A Testing Methodology

The complexity measure v is designed to conform to our intuitive notion of complexity and since we often spend as much as 50 percent of our time in test and debug mode the measure should correlate closely with the amount of work required to test a program. In this section the relationship between testing and cyclomatic complexity will be defined and a testing methodology will be developed.

Let us assume that a program P has been written, its complexity v has been calculated, and the number of paths tested is ac (actual complexity). If ac is less than v then one of the following conditions must be true:

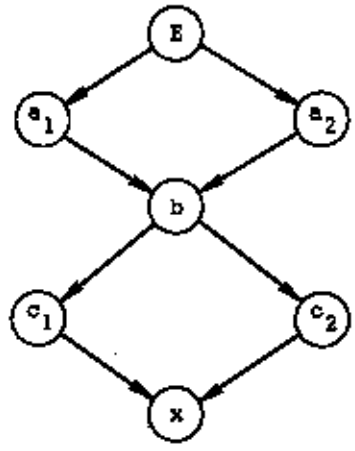
1) there is more testing to be done (more paths to be tested);

2) the program flow graph can be reduced in complexity by v-ac (v-ac decisions can be taken out); and

3) portions of the program can be reduced to in line code (complexity has increased to conserve space).

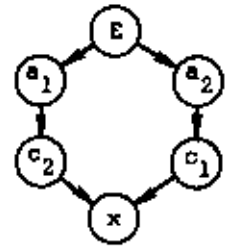
Up to this point the complexity issue has been considered purely in terms of the structure of the control flow. This testing issue, however, is closely related to the data flow because it is the data behavior that either precludes or makes realizable the execution of any particular control path. A few simple examples may help to illustrate. Assume we start with the following flow graph:

G:



Suppose that ac = 2 and the two tested paths are [E, a_1, b, c_2, x] and [E, a_2, b, c_1, x]. Then given that paths [E, a_1, b, c_1, x] and [E, a_2, b, c_2, x] cannot be executed we have ac<v so case 2 holds and G can be reduced by removing decision b as in

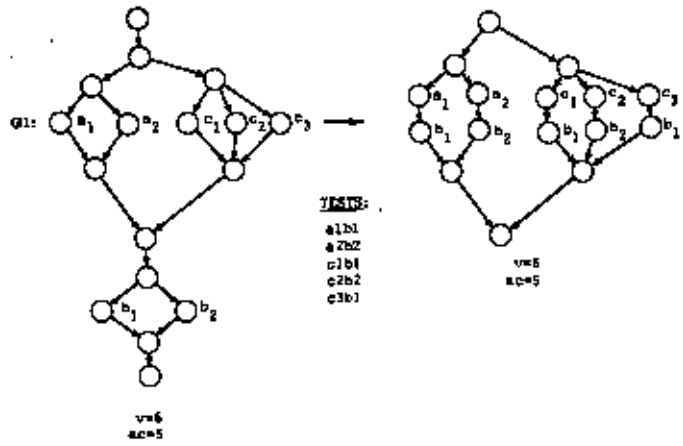
G1:

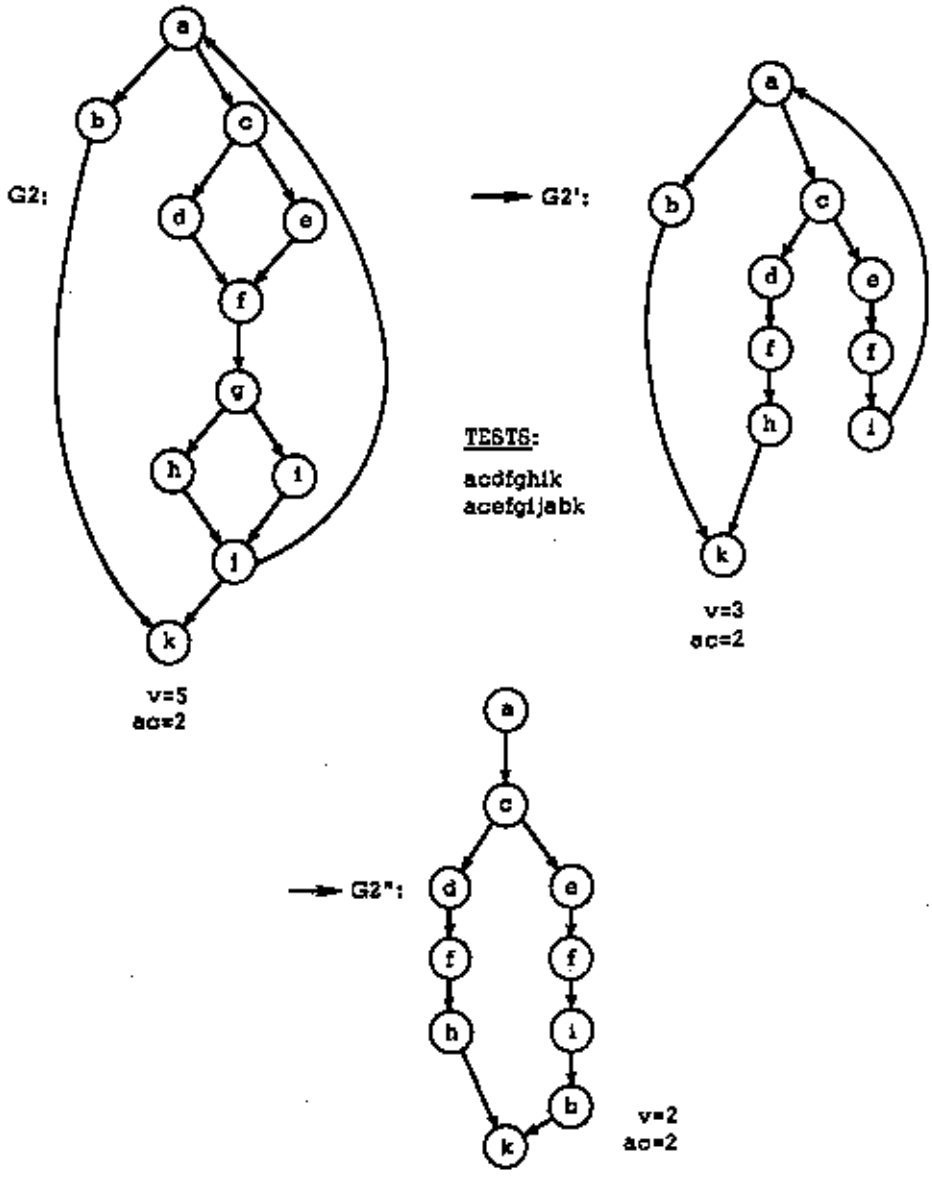


Notice how in G v = ac and the complexity of G1 is less than the complexity of G.

In experience this approach is most helpful when programmers are required to document their flow graph and complexity and show explicitly the different paths tested. It is often the case when the actual number of paths tested is compared with the cyclomatic complexity that several additional paths are discovered that would normally be overlooked. It should be noted that v is only the minimal number of independent paths that should be tested. There are often additional paths to test. It should also be noted that this procedure (like any other testing method) will by no means guarantee or prove the software—all it can do is surface more bugs and improve the quality of the software.

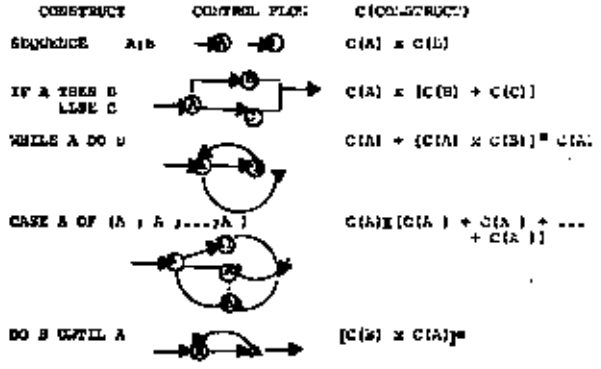
Two more examples are presented without comment.





APPENDIX

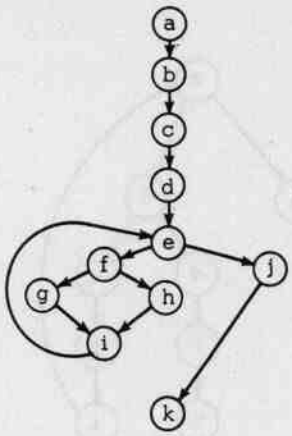
A method of computing the number of possible paths in a structured program will be briefly outlined. This method associates an algebraic expression C with each of the structured constructs and assumes that the complexity of a basic functional or replacement statement is one. The various syntactic constructs used in structured programming and their control flow and complexity expressions are shown below. The symbol α stands for the number of iterations in a loop.



The program SEARCH below is used to illustrate. SEARCH performs a binary search for input parameter ITEM on a table T of length N. SEARCH sets F to 1 and J to ITEM's index within T if the search is successful—otherwise F is set to 0 indicating that ITEM is not in T.

```
PROCEDURE SEARCH (ITEM)  INTEGER ITEM
  BEGIN
    INTEGER  L, H;
    F←0;
    L←0;
    H←N;

    While  H > L   and   F = 0  Do
      If  T[J < (H+L)  DIV 2] = item
        THEN
          If item   <   T[J]

              THEN

                H ← J - 1

              ELSE

                L ← J + 1

        ELSE

          F ←1
  END
```

The flow graph for SEARCH is



The algebraic complexity C would be computed as

$$C(\text{SEARCH}) = 1\ 1\ 1\ \ \{1 + (1\ [1[1 + 1] + 1])^{\alpha}\ 1\}$$
$$= \{1 + (3)^{\alpha}\}.$$

Assuming I to be at least 4, the lower bound for the expression $\{1 + 3^{\alpha}\}$ is 4 which indicates there are at least 4 paths to be tested. The first test would be from the immediate exit from the WHILE loop which could be tested by choosing H less than L initially. The next three tests (the three ways through the body of the loop) correspond to cases where ITEM = T[J], ITEM < T[1], and ITEM > T[J].

References

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

[2] B. W. Boehm, "Software and its impact: A quantitative assessment," *Datamation*, vol. 19, pp. 48-59, May 1973.

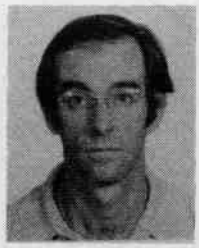
[3] W. B. Cammack and H. J. Rogers, "Improving the programming process," IBM Tech. Rep. TR 00.2483, Oct. 1973.

[4] D. E. Knuth, "Structured programming with GOTO statements," *Computing Surveys*, vol. 6, pp. 261-301, Dec. 1974.

[5] R. Kosaraju, "Analysis of structured programs," *J. Comput. Syst. Sci.*, vol. 9, pp. 232-255, Dec. 1974; also, Dep. Elec. Eng., The Johns Hopkins Univ., Baltimore, MD, Tech. Rep. 72-11, 1972.

[6] H. Legard and M. Marcotty, "A generalogy of control structures," *Commun. Assoc. Comput. Mach.*, vol. 18, pp. 629-639, Nov. 1975.

[7] H. D. Mills, "Mathematical foundations for structured programming," Federal System Division, IBM Corp., Gaithersburg, MD, FSC 72-6012, 1972.



Thomas J. McCabe was born in Central Falls, RI, on November 28, 1941. He received the A.B. degree in mathematics from Providence College, Providence, RI and the M.S. degree in mathematics from the University of Connecticut, Storrs, in 1964 and 1966, respectively.

He has been employed since 1966 by the Department of Defense, National Security Agency, Ft. Meade, MD in various systems programming and programming management positions. He also, during a military leave, served as a Captain in the Army Security Agency engaged in large-scale compiler implementation and optimization. He has recently been active in software engineering and has developed and taught various software related courses for the Institute for Advanced Technology, the University of California, and Massachusetts State College System.

Mr. McCabe is a member of the American Mathematical Association.