Cyclomatic complexity

The complexity of algorithms

White Box vs. Black Box Testing

Black Box Testing

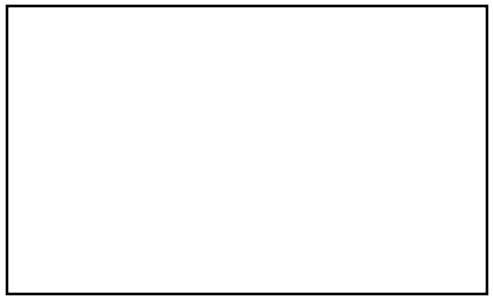
Test Focus: Requirements

Validation Audience: User

Code Coverage: Supplemental

Positive Test: Testing things in your specs that you already know about

White Box Testing



Test Focus: Implementation

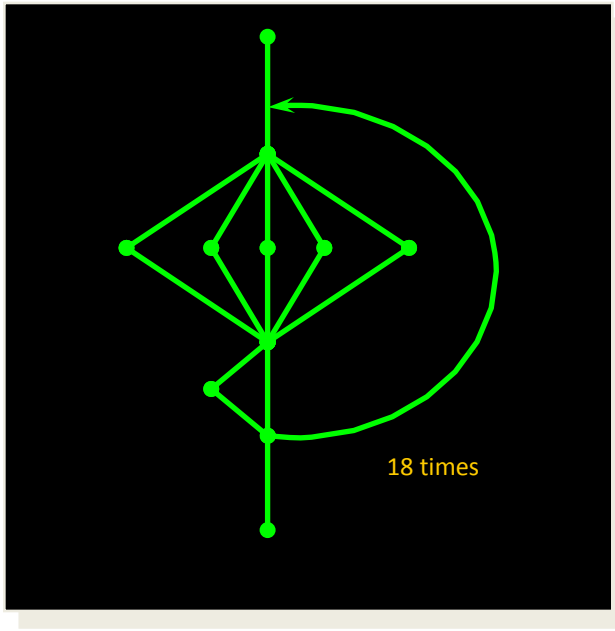
Validation Audience: Code

Code Coverage: Primary

Negative Test: Testing things that may not be in your specs but in the implementation

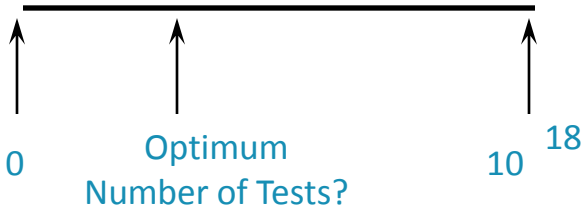
IV. Testing Software for Reliability - Structured Testing

Impossibility of Testing Everything



All Statistical Paths = 10^{18}

- It is IMPOSSIBLE to test All Statistical Paths
- So, from structural coverage view, When Should We Stop Testing?
  - Not care? (planes do, missiles do)
  - All lines?
  - All branches?
  - All cyclomatic paths?

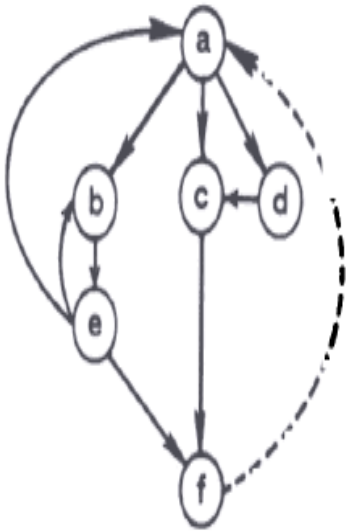


If allow 1 Nanosecond per test, time is :

$$T = \frac{10^{18}}{10^{9} \times 3600 \times 24 \times 365} = 31.7 \text{ Years}$$

A complexity measure

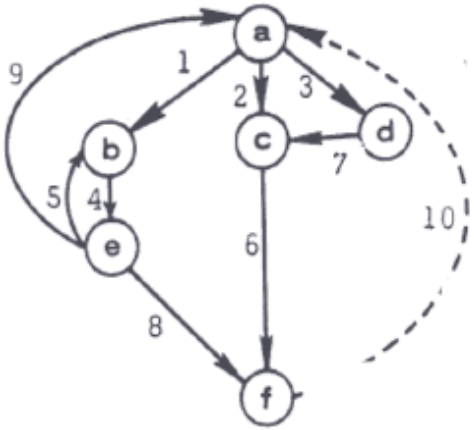
G:



- Definition: The cyclomatic number v(G) of a graph G with e edges, n nodes, and p connected components is e-n+p.

- Theorem: In a strongly connected graph the cyclomatic number is equal to the maximal number of linearly independent circuits.

With this example



- V(G) = 10 – 6 + 1 = 5
- # of regions = 5

Pick paths for the 5 circuits

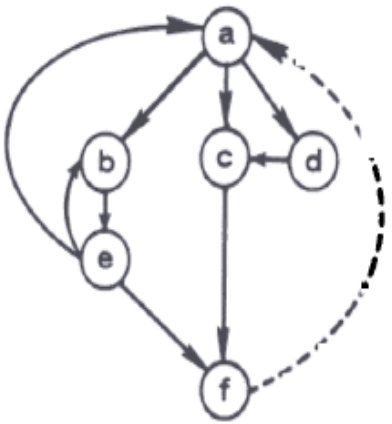
- Using theorem 1 we choose a set of circuits that are paths. The following set B is a basis set of paths.

- B: (abef) , $(a(be)^2f)$, $((abe)^2f)$, (acf) , (adcf)

- Linear combinations of paths in B will generate any path. For example,

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

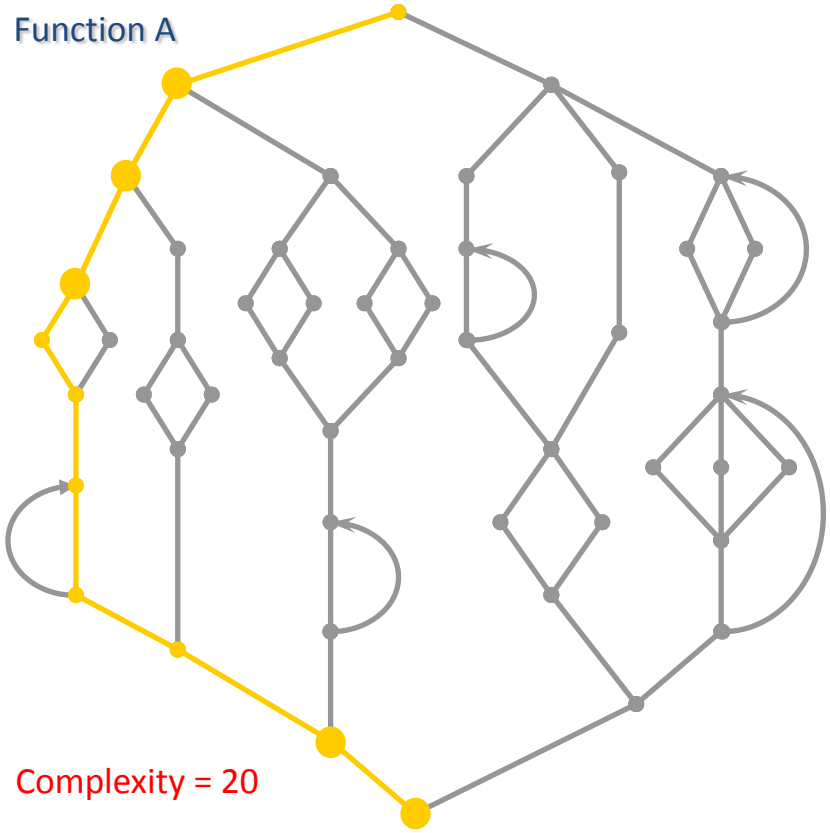
- $(a(be)^2abef) = (a(be)^2f)+((abe)^2f)-(abef)$



Methods of computing v(G)

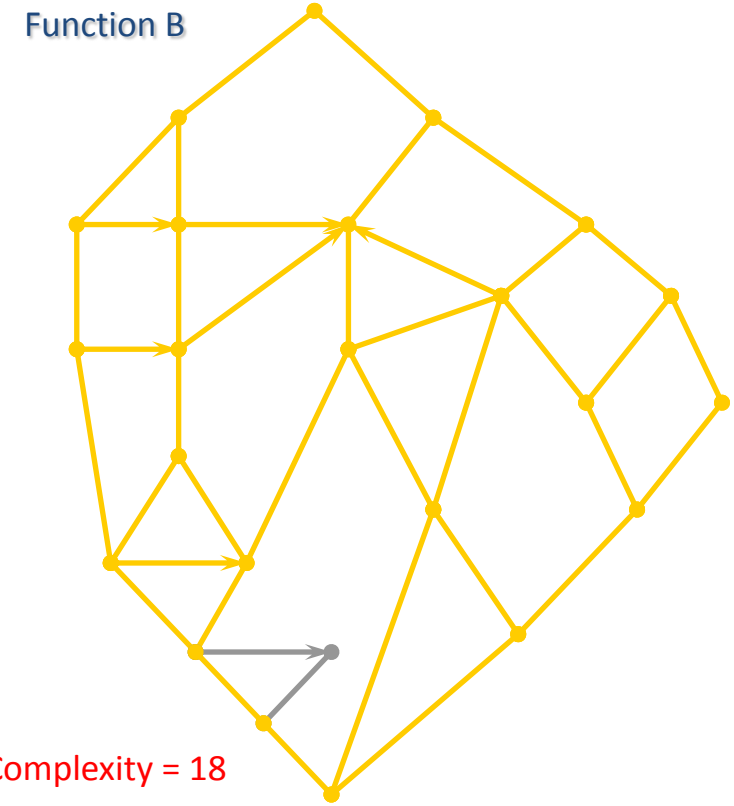
- E-n+2
- # decisions + 1
- # of regions in a planar flow graph

Complexity (decisions) & # lines here are similar, but …



Complexity = 20

Unstructuredness (evg) = 1



Complexity = 18

Unstructuredness (evg) = 17

So B is MUCH harder to maintain!

Applications

Testing

Essential complexity --- reverse engineering

Design Complexity ---- system integration

Data complexity --- Y2K testing

Pareto Distribution of errors --- reverse engineering

Code Breaker ---- Reuse

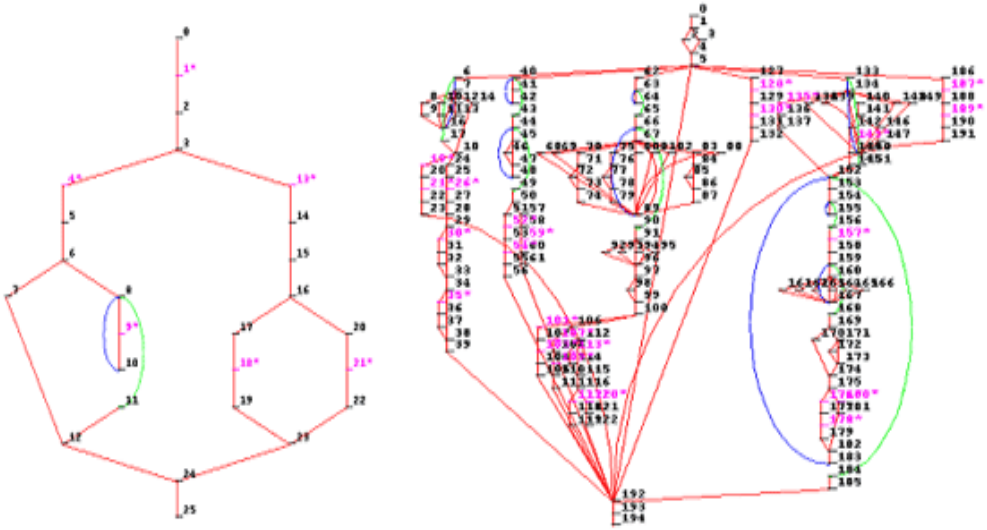
Security threats

Data Slicing ----- Reuse

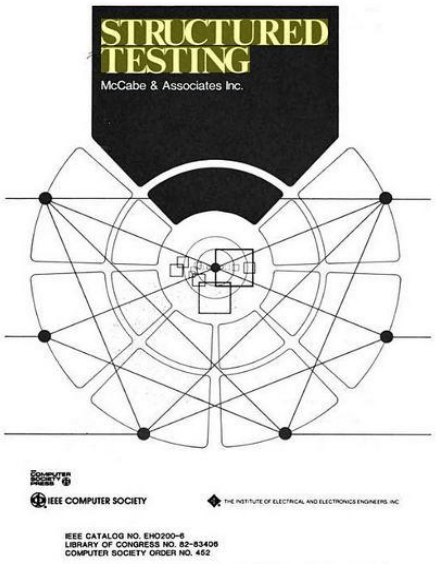
Code walkthroughs ---- validation

Design validation

Requirements Validation



Google book search – McCabe Complexity – 22,000 books



Google web search (McCabe complexity): 456,000 references

In 2009 Tom McCabe's publication on Software Complexity was chosen as one of the retrospective 23 highest impact papers in computer science by the ACM-SIGSOFT

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].

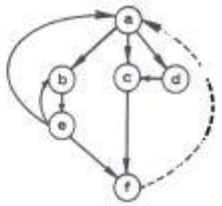
Definition 1: The cyclomatic number V(G) of a graph G with n vertices, e edges, and p connected components is

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

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