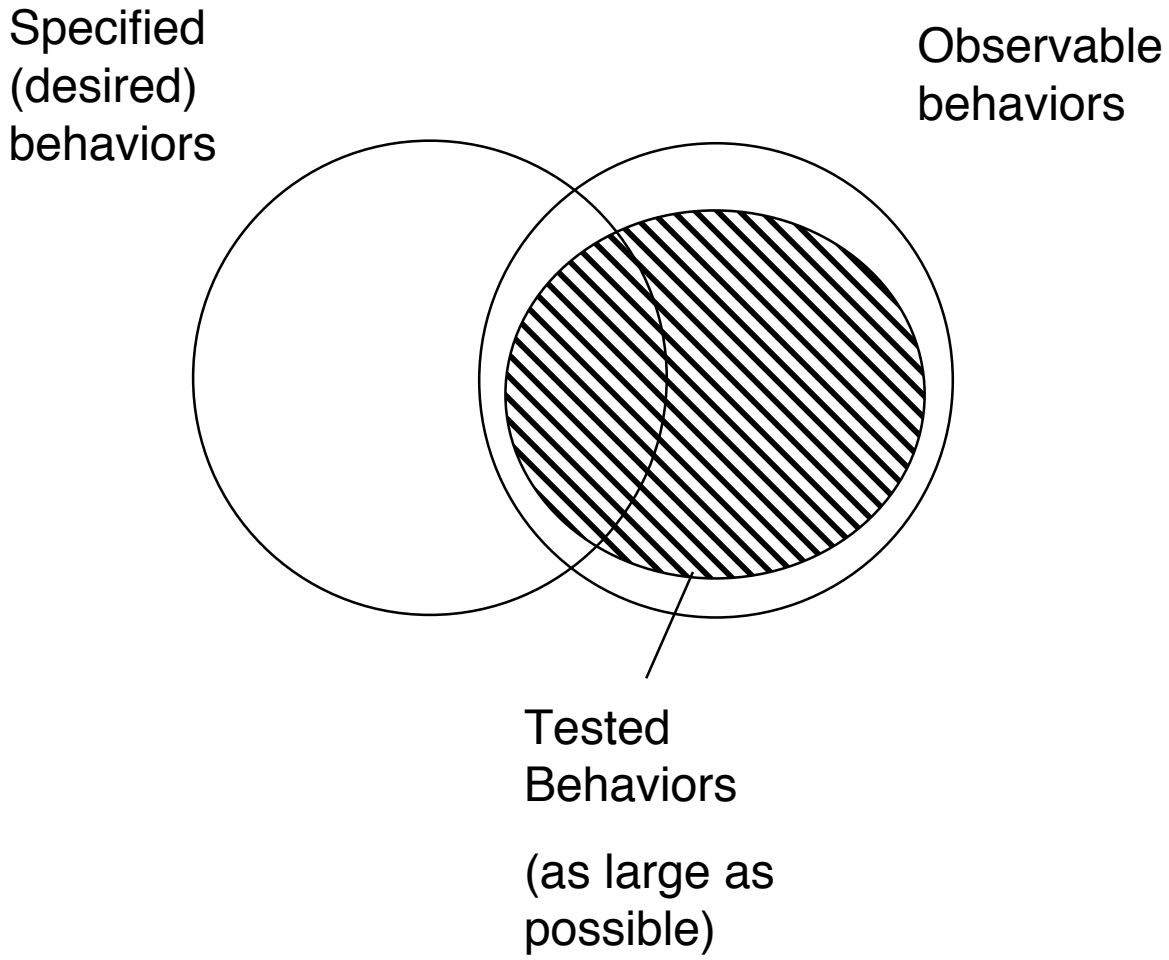
Testing Part 2

White-Box Testing

Good White-Box Test Plan



Consider the "triangle" program

- You are now given source code for the triangle program.
- How would the tests that you would perform differ from those in the black-box case?

Code for the triangle program (page 1 of n).

```
/**
 * file:    triangle.cc
 * author:  keller
 * purpose: pedagogical testing example
 *          Repeats the following until end-of-file:
 *             Inputs three numbers, then determines
 *             whether they form a triangle, and if so, what kind.
 *
 *          The absolute values of the numbers must be in the range
 *             between Conversion::MIN and Conversion::MAX as defined below,
 *          otherwise the entire case input will be rejected.
 *          These limits are set because the sides may be squared in analysis.
 *          Negative numbers will be interpreted as their absolute value.
 */

#include <assert.h>
#include <iostream>
#include <string>

/**
 * Conversion deals with conversion of input from string to floating numeral.
 * Provides check of whether the number contains spurious characters and
 * is within the designated range.
 */

class Conversion
{
public:
/** top of acceptable numeric range */
static const double MAX = 1e150;

/** bottom of acceptable numeric range */
static const double MIN = 1e-150;

/** Check whether argument is within range */
inline static bool inRange(double value) {return MIN <= value && value <= MAX;}
```

```
/**
 * Check whether an input string represents a valid floating point numeral
 * (as opposed to some non-digit characters, etc.)
 * and, if so, whether the number is in the specified range.
 *
 * @param inputString string to be checked for being a numeral, and possibly
 *         converted to number result.
 *
 * @param result number to which string was converted, if successful
 *
 * @return true if the input represents a valid number in the specified
 *         range and result is the numeric value,
 *         otherwise return false and 'result' is undefined.
 *
 * pre-condition: true <br>
 *
 * post-condition: If returned value is true, then result is the numeric
 *                 representation of the argument inputString.
 *                 If returned value is false, then 'result' is undefined.
 */

static bool convert(const std::string inputString, double& result)
  {
  char* endptr;

  // After strtod, endptr will point to end of converted string.
  // Conversion is successful iff *endptr is the null character (0).

  result = strtod(inputString.c_str(), &endptr);

  if( *endptr == 0 )
    {
    result = fabs(result);

    return inRange(result);
    }

  return false;
  }
};  // Conversion
```

```
/**
 * TriangleTester deals analyze three sides as a triangle and reading
 * sets of three sides from an input stream.
 */

class TriangleTester
{
public:
static const int SIDES = 3;              // This is about triangles.

static enum{ NOT_A_TRIANGLE,
             EQUILATERAL_TRIANGLE,
             SCALENE_TRIANGLE,
             RIGHT_SCALENE_TRIANGLE,
             ISOSCELES_TRIANGLE,
             RIGHT_ISOSCELES_TRIANGLE} Classification;
```

```
/**
 * Return the type of triangle, if any, that is formed by three sides.
 * @param a first side
 * @param b second side
 * @param c third side
 *
 * pre-condition: all sides are positive and in-range <br>
 *
 * post-condition: the correct classification is returned
 */

static int analyze(double a, double b, double c)
  {
  // Arrange the three sides to simplify subsequent analysis.

  order(a, b);
  order(b, c);

  if( a + b <= c )  // The only test needed for triangle-ness, due to ordering.
    {
    return NOT_A_TRIANGLE;
    }
  if( a == b && b == c )
    {
    return EQUILATERAL_TRIANGLE;
    }

  bool right = isRight(a, b, c);

  if( a == b || b == c )
    {
    return right ? RIGHT_ISOSCELES_TRIANGLE : ISOSCELES_TRIANGLE;
    }

  return right ? RIGHT_SCALENE_TRIANGLE : SCALENE_TRIANGLE;
  }
```

```
/**
 * Return 1 if the triangle is a right triangle, otherwise return 0.
 * @param a length of the first side
 * @param b length of the second side
 * @param c length of the third side
 * @return 1 if the triangle is a right triangle, otherwise return 0.
 *
 * pre-condition: a <= b && b <= c    <br>
 *
 * post-condition: return value indicates right triangle
 */

static inline int isRight(double a, double b, double c)
  {
  return a*a + b*b == c*c;
  }


/**
 * Order numbers a and b so that a <= b.
 * @param a one of the two numbers to be ordered
 * @param b the other of the two numbers to be ordered
 *
 * pre-condition: true <br>
 *
 * post-condition: a <= b
 */

static inline void order(double& a, double& b)
  {
  if( a > b )
    {
    double temp = a;
    a = b;
    b = temp;
    }
  // assert( a <= b );
  }
```

```
/**
 * Until end-of-file, read groups of three numbers and classify whether they
 * are all in range and could be the sides of a triangle, and if so,
 * what kind:
 *    [right] {equilateral, isosceles, scalene}
 *
 * @param in istream containing groups of three sides
 * @param out ostream on which results are shown
 */

static void test(std::istream& in, std::ostream& out)
  {
  std::string inputSide[SIDES];
  double side[SIDES];

  // read numbers as strings, exit loop if end-of-file

  while( in >> inputSide[0] >> inputSide[1] >> inputSide[2] )
    {
    for( int i = 0; i < SIDES; i++ )    // echo sides
      {
      out << inputSide[i] << " ";
      }

    // convert inputs to numeric and show any bad ones

    bool inputOk = true;

    for( int i = 0; i < SIDES; i++ )
      {
      if( !Conversion::convert(inputSide[i], side[i]) )
        {
        inputOk = false;
        out << "\nbad input: " << inputSide[i];
        }
      }

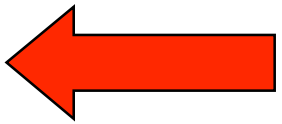
    if( inputOk )
      {
      switch( analyze(side[0], side[1], side[2]) )
        {
        case NOT_A_TRIANGLE:           out << "not a triangle.";          break;
        case SCALENE_TRIANGLE:         out << "scalene triangle.";        break;
        case RIGHT_SCALENE_TRIANGLE:   out << "right scalene triangle."; break;
        case ISOSCELES_TRIANGLE:       out << "isosceles triangle.";      break;
        case RIGHT_ISOSCELES_TRIANGLE: out << "right isosceles triangle.";break;
        case EQUILATERAL_TRIANGLE:     out << "equilateral triangle.";    break;
        }
      }
    out << std::endl;
    }
  }
};  // TriangleTester
```

White-Box in Conjunction with Verification Techniques

- Verification is based on techniques for reasoning about programs.
- These techniques can also be used to simplify the number of white-box test cases.

Using Live Assertions

- ```
  order(a, b);
  order(b, c);

  assert( a <= b && b <= c );
  ```
  
- The program will now tell us when the assumption is wrong.
- It will provide an indication of what has to be rethought.
  ```

Verification + W.B. Testing

- Could use verification to help fix the problem.
- Having *verified* the assertion

```
assert( a <= b && b <= c );
```

  will testing be simplified?
- How?

Some Ideas for More Effective Testability

Design for Test (DFT)

- **Instrument** your code as you build it; this could help with unit tests
  - Code-in traces, explanations, indications, ...
  - Being able to turn instrumentation on or off can help understand whether sub-systems are working correctly.

Example: C++ Instrumentation

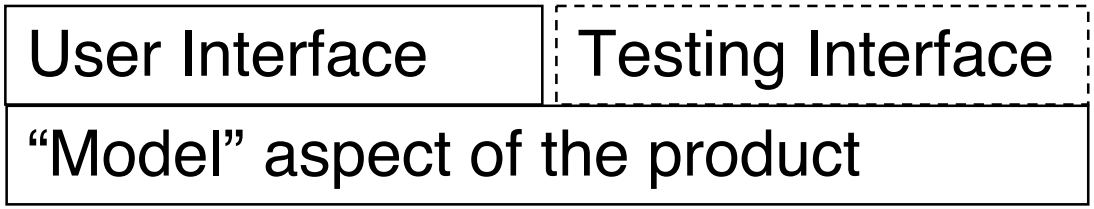
```
class Trace
{
static int currentLevel;

static void trace(string message, int level)
     {
     if( level >= currentLevel )
             cerr << message << endl;
     }

static void setLevel(int level)
     {
     currentLevel = level;
     }
};
```

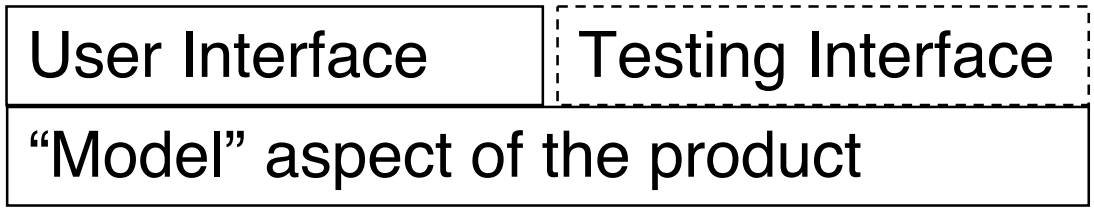
Design for Test
(used extensively in hardware domain)

- Build **testing interfaces** into the code. These are interfaces that can be seen by the developer but not the user.



- These interfaces allow greater automation in the testing process, since they can be driven by a testing program more readily than one requiring a user interface (such as a GUI or CLI).

Testing Interface



Build Self-Test Routines into the Code

- **Self-tests** can employ data generators that will stress test the code.

Code Inspection and Walkthrough

- Used for:
  - Identifying errors
  - Identifying test cases
  - Helping yourself and others to understand how the program works.

Classifying Programming Errors

Categories of Programming Errors

- Logic errors
- Pointer errors
- Numeric accuracy/precision/roundoff errors
- Input/output representation errors
- Data structure usage errors
- Memory errors
- User-interface errors (windows, etc.)
- Environment errors, such as misuse of file-system, devices, etc.
- Timing errors, such as in a real-time system

Exercise

- Each team take one category of error.
- List as many specific sub-categories of this error as you can.
- Are there testing approaches specific to this sub-category of error?
- **Example**: Logic Errors:

Logic Error Categories (1)

- Numeric and character boundary
  - Off-by-1
- Array-out-of-bound
  - Insufficient space allocated
  - Input or output buffer overflow
- Inequality comparison (used > instead of >= or <)
- Used ++ instead of --.
- Used pre-incrementation rather than post.

Logic Error Categories (2)

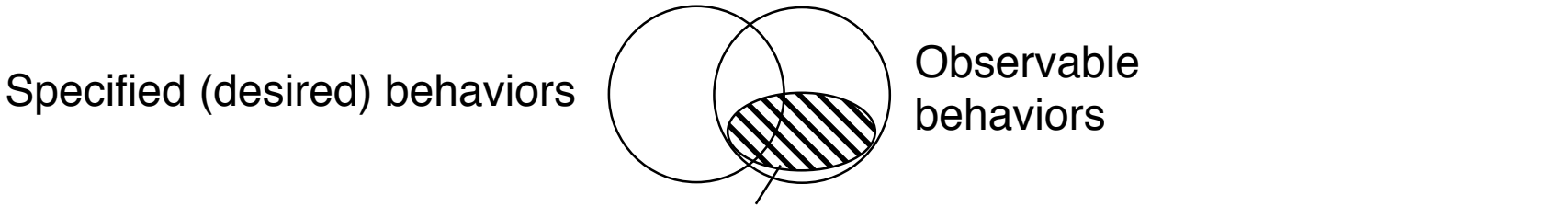
- Negation error
- Loop continuation criterion
  - Infinite loops
- Flag not cleared when used
- Flag, count, or sum not initialized
- Routine not reinitialized before subsequent use
- Dynamic type-casting exception

Theoretical Aspects of Whitebox Testing

Coverage Analysis

White-Box Coverage

- **"Coverage" notion**:
  - View the program as a directed graph (flowchart or data-flow diagram)
  - Develop (external or internal) tests that "cover", i.e. exercise program to various degrees.



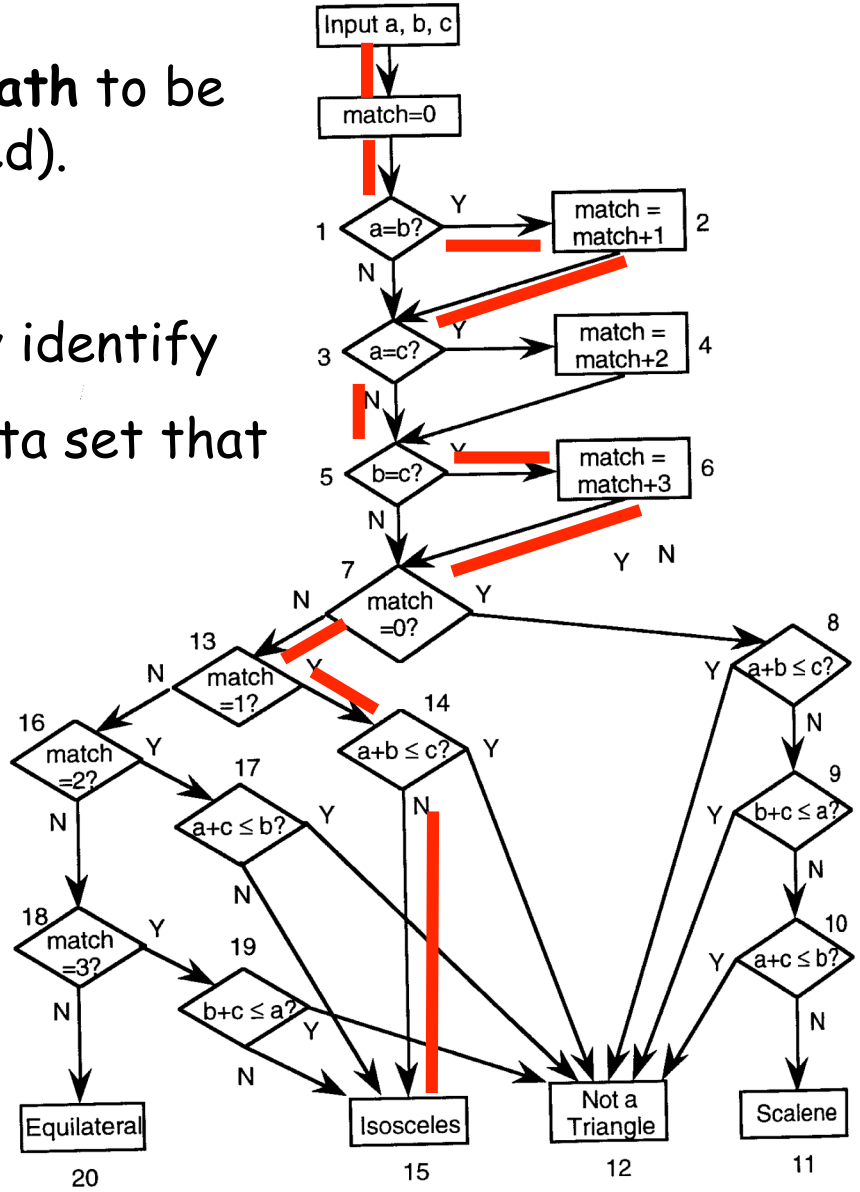
Minimal Coverage

- Tests should exercise, at least once, every:
  - variable
  - assignment box
  - decision box
  - simple path through flow chart

Testing Based on Paths

Example of **a single path** to be covered by test (in red).

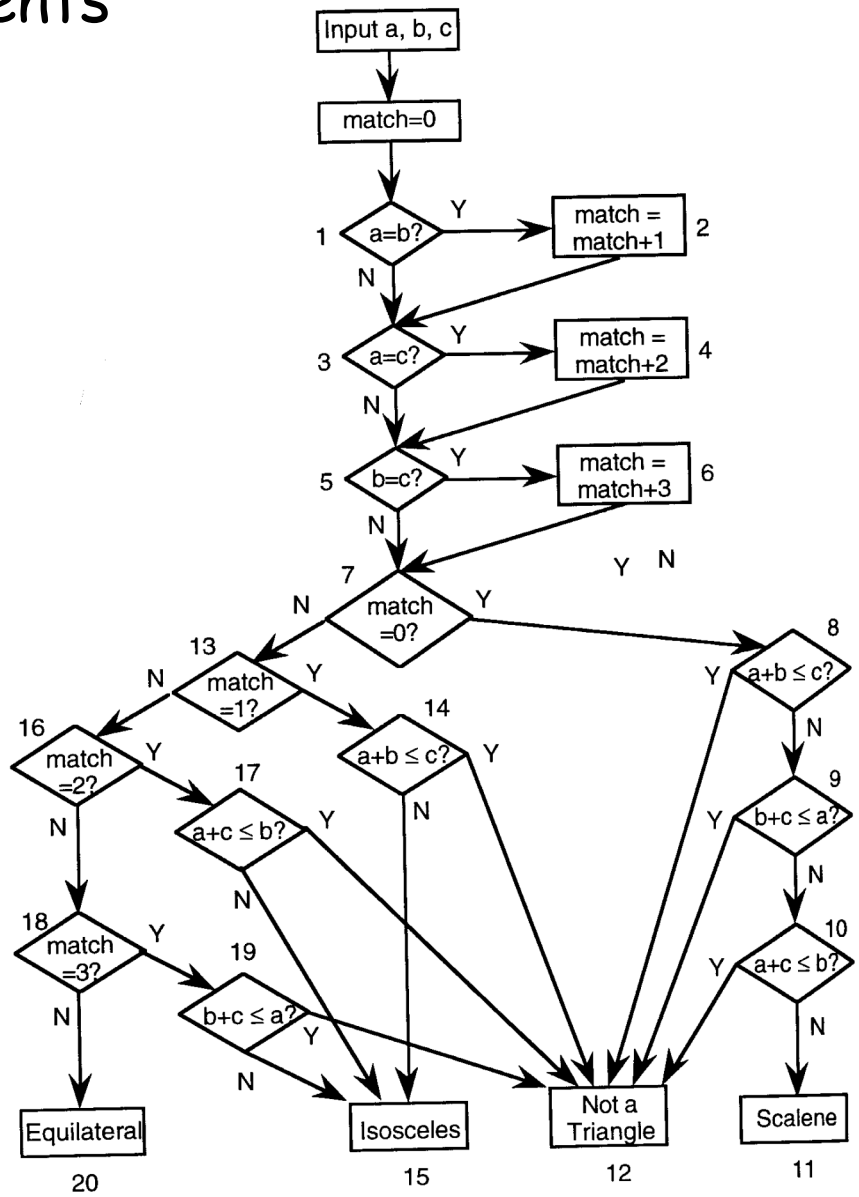
Insufficient to simply identify path; need to have data set that will exercise it.



Coverage "Basis"

- Infeasible to test ***all*** paths, etc.
- Instead, identify a "**basis**" from which all paths can be constructed.
- Make sure every element of basis is covered by ***some*** test
- **Example**: Basis could be set of all edges; Compute a set of tests that covers all.

Estimate the number of elements in a basis set of paths that will cover all edges.



D-D Path Nomenclature (Ed Miller 1977)

- D-D = "Decision to Decision"
- Path between two decisions, that contains no decision itself
- *Dependence* among D-D paths, e.g. a variable defined in one path is referenced in another.

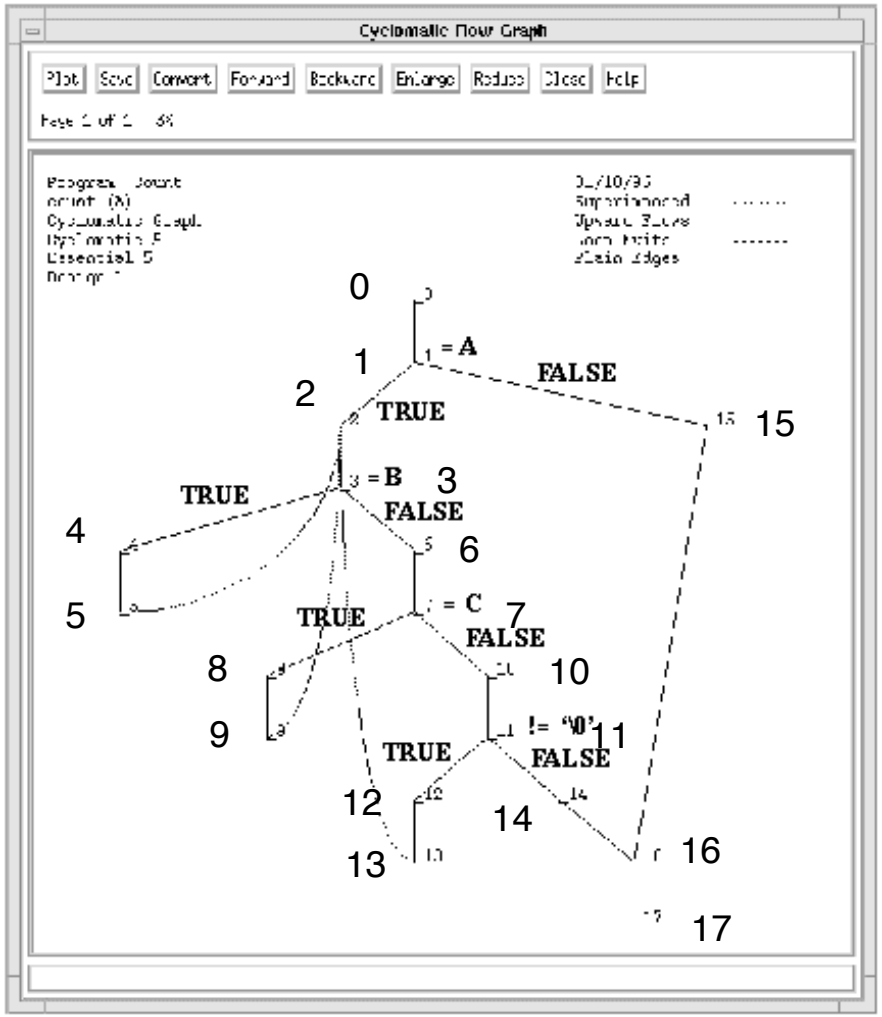
Classification of Structural Test Coverage

- C_0 Every statement tested
- C_1 Every D-D path tested
- C_1p Every **p**redicate & outcome tested
- C_d C_1 + every *inter-dependent pair* of D-D paths tested
- C_i^k Every path that contains up to k repetitions of a loop (e.g. k = 2) tested
- C_∞ Every path tested

"Baseline"-based method for constructing a basis

- Pick a single linear path through the program, the "baseline".
- Pick the next path by taking decisions alternate to the baseline.
- Repeat picking other alternates to the alternates, etc.
- Eventually a basis number of tests will be reached.

Baseline Testing Method (in McCabe's Cyclomatic Tool)



Test Path 1 (baseline): 0 1 2 3 4 5 2 3 6 7 10 11 14 16 17

11(1): string[index]=='A' ==> TRUE

13(3): string[index]=='B' ==> TRUE

13(3): string[index]=='B' ==> FALSE

18(7): string[index]=='C' ==> FALSE

25(11): string[index]!='\0' ==> FALSE

Test Path 2: 0 1 15 16 17

11(1): string[index]=='A' ==> FALSE

Test Path 3: 0 1 2 3 6 7 10 11 14 16 17

11(1): string[index]=='A' ==> TRUE

13(3): string[index]=='B' ==> FALSE

18(7): string[index]=='C' ==> FALSE

25(11): string[index]!='\0' ==> FALSE

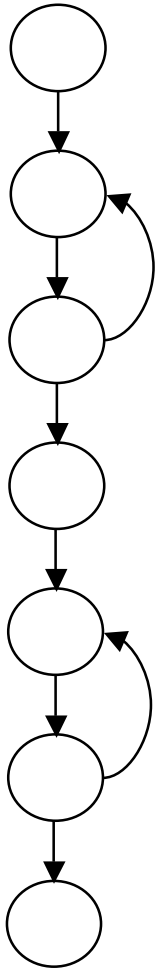
Test Path 4: 0 1 2 3 4 5 2 3 6 7 8 9 2 3 6 7 10 11 14 16 17

Complexity-Based Testing

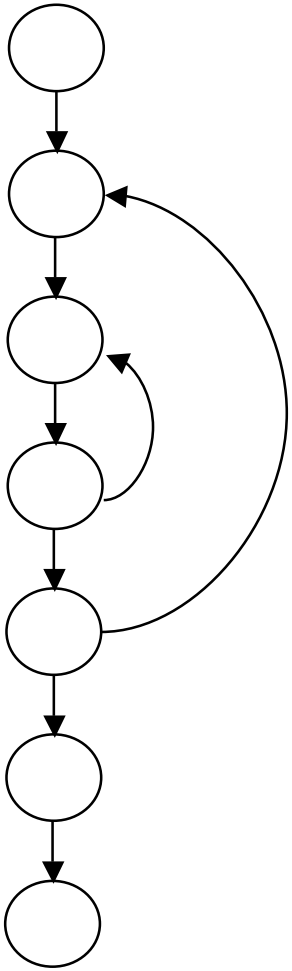
- Construct the flowchart for the program.
- Calculate the graph complexity *C*.
- **Find *C* independent paths** and corresponding test data for each.
- Execute program on test data.
- Check results with what is expected.

Classifying Loop Complexity (Informal)

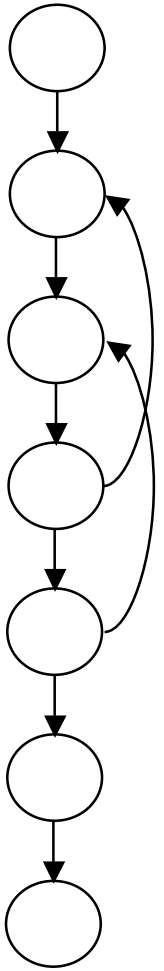
Concatenated



Nested



Intersecting



Program Graph Complexity Measures

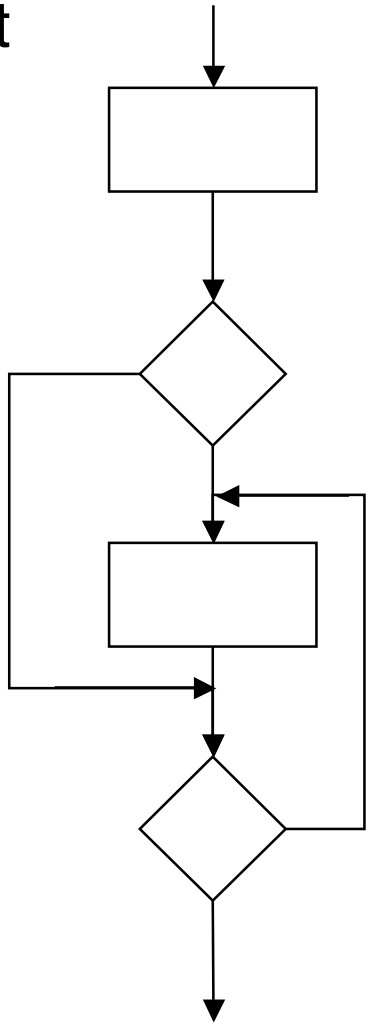
- Used to determine ***how much*** testing required for a given sub-system
- Examples
  - McCabe Cyclomatic complexity
  - Halstead software metric

McCabe Cyclomatic complexity for a program graph

- IEEE Trans. Softw. Engrg., Dec. 1976
- Popular, but theoretically questionable
- $\nu(G) = e - n + p$
  - e is the number of **edges** (arcs), which represent data processing **boxes**
  - n is the number of **nodes**, which represent **points** where edges are connected
  - p is the number of **separate parts** (connected sub-graphs)

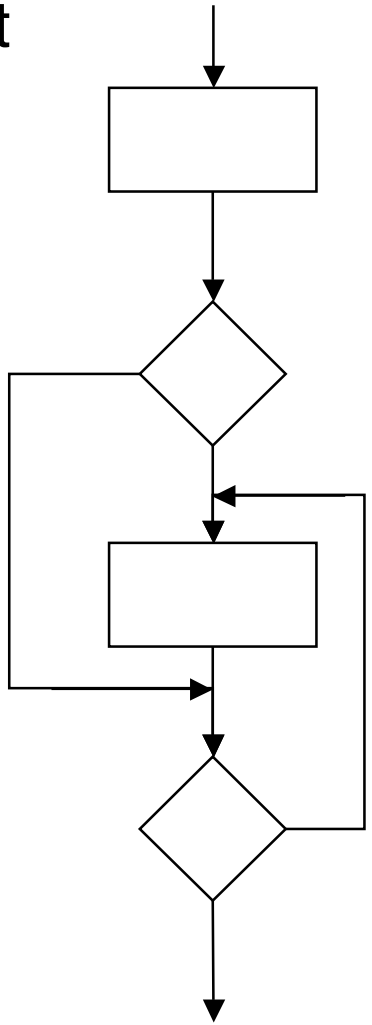
Example

Flowchart

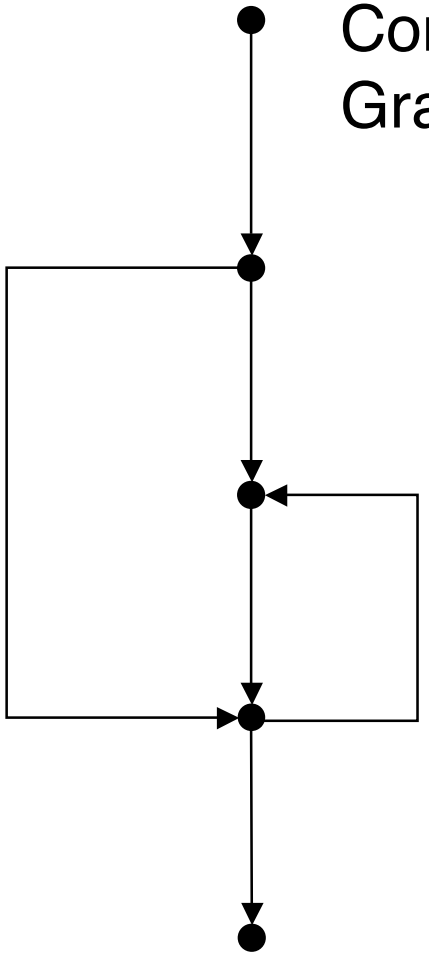


Example

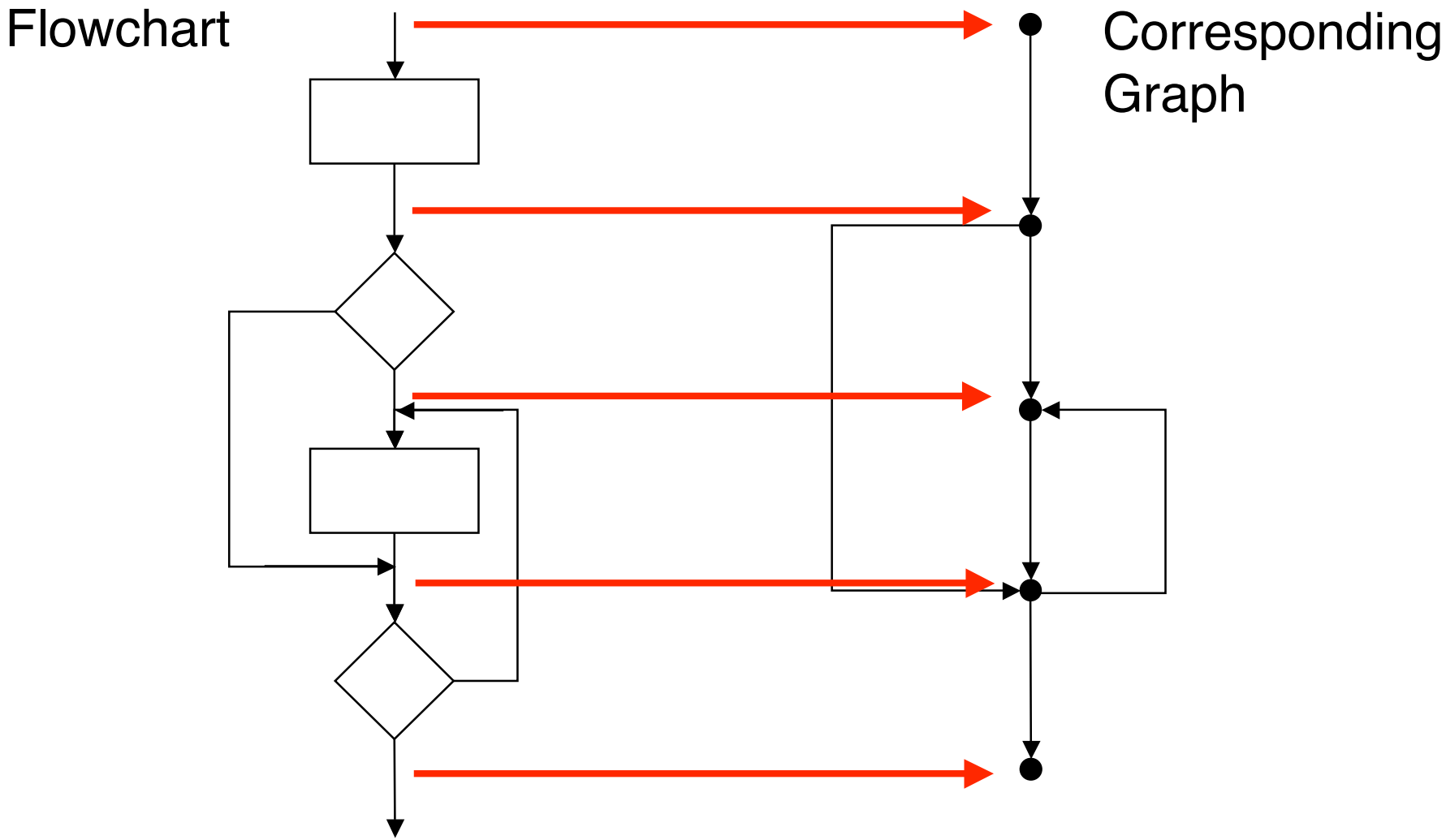
Flowchart



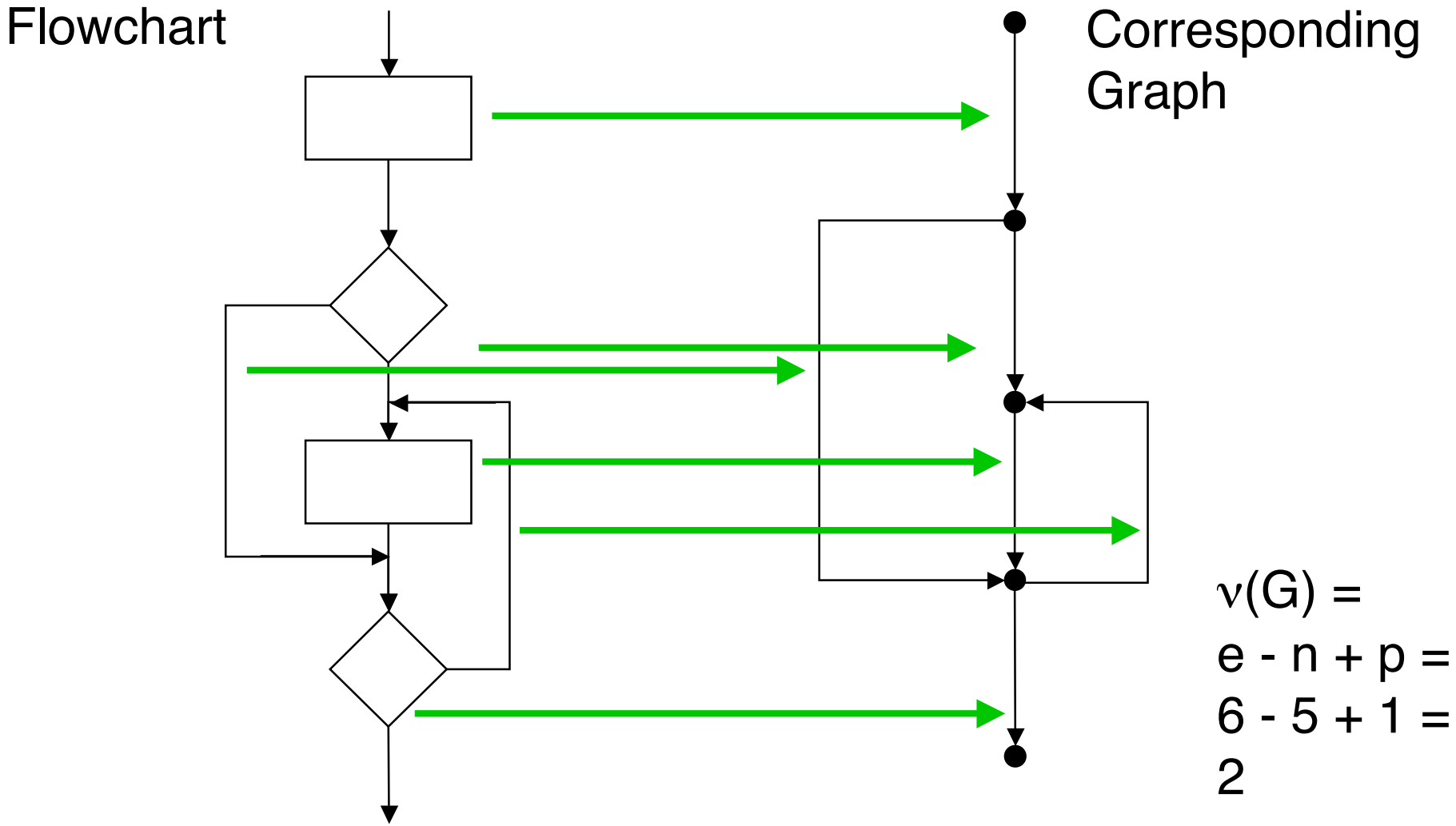
Corresponding Graph



Node Correspondence



Edge Correspondence



McCabe's measure drawn from Cyclomatic Number of a Graph

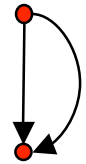
- Defined by Berge (*Theory of graphs and its applications*, 1962), motivated by Euler
- Originally defined for ***undirected*** graphs
- $\nu(G)$ = edges - nodes + separate parts
- **Property**: *Connecting* two nodes increases ν value by 1 if there was a path between the two nodes; otherwise it leaves ν value the same.

Illustration of Property

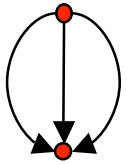
Connecting two nodes increases ν value by 1 if there was a path between the two nodes; otherwise leaves ν value the same.

ν= 0 - 2 + 2 = 0

ν= 1 - 2 + 1 = 0

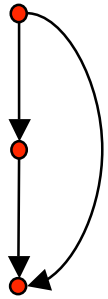


ν= 2 - 2 + 1 = 1

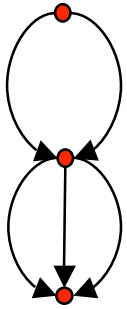


ν= 3 - 2 + 1 = 2

ν= 2 - 3 + 1 = 0



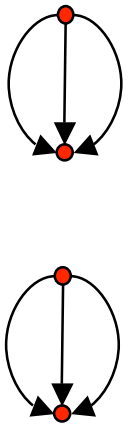
ν= 3 - 3 + 1 = 1

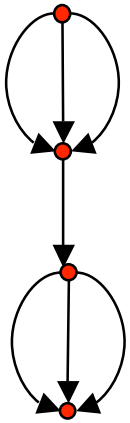


ν= 5 - 3 + 1 = 3

Illustration of Property

Connecting two nodes increases ν value by 1 if there was a path between the two nodes; otherwise leaves ν value the same.





$\nu = 6 - 4 + 2 = 4$

$\nu = 7 - 4 + 1 = 4$

Further Properties of the Cyclomatic *Number* of a Graph

- $\nu(G) = 0$ iff G contains no ***undirected*** cycles (i.e. is "straight-line").

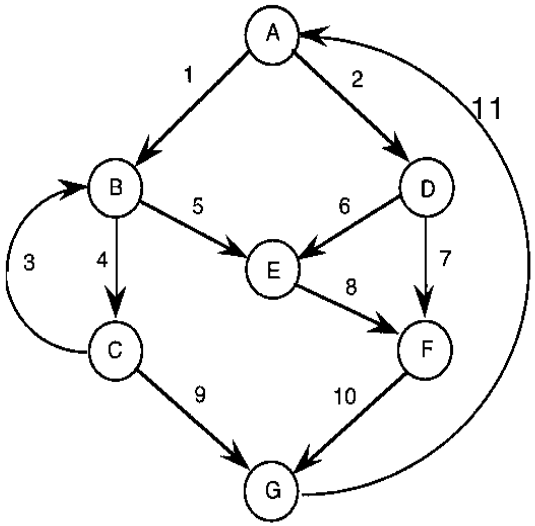
- If G is strongly connected (has only one component), then $\nu(G)$ is the maximum number of linearly- independent undirected cycles.

  [This can be used to compute size of a basis.]

Linear Independent Circuits?

- Think of the edges of a graph as components of a **vector**.
- A circuit is abstracted by the number of times each edge is traversed in the (undirected) circuit.
- One circuit can be constructed from others by *adding* (mod 2) the corresponding vectors.
- (But not all vectors so-constructed correspond to executable paths, or even circuits themselves.)

Adding Circuits



c1 = 1-4-9-11 = (1 0 0 1 0 0 0 0 1 0 1)

c2 = 2-7-10-11 = (0 1 0 0 0 0 1 0 0 1 1)

c3 = 1-4-9-10-7-2 = (1 1 0 1 0 0 1 0 1 1 0)

$\nu(G) = e - n + p$

$= 11 - 7 + 1$

$= \mathbf{5}$

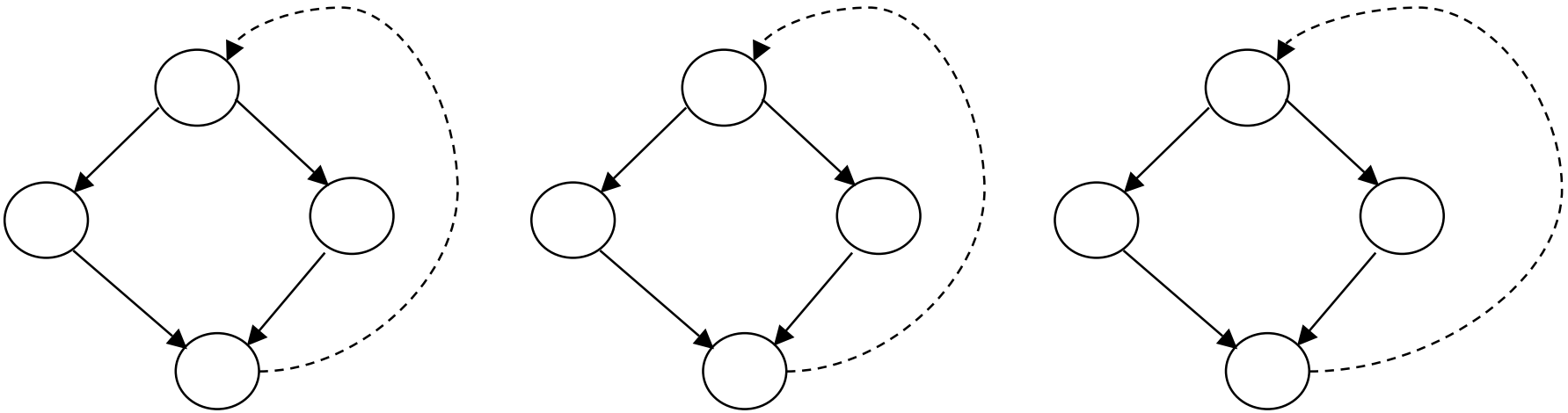
c3 = c1 + c2
c1 and c2 are linearly independent, etc.

5 circuits form a **basis**

McCabe Complexities of Typical Program Graphs

- McCabe: If a part of a graph is not strongly connected, add a **phantom arc** from finish to start;
- Might think of this phantom arc as representing the repeated use of the program graph.
- *Alternatively*, use the *modified* formula $\mu(G) = e - n + 2p$
- Above, 2p accounts for one *phantom* arc in each separate part.

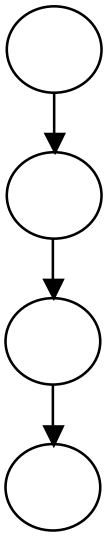
Adding one phantom edge per separate part



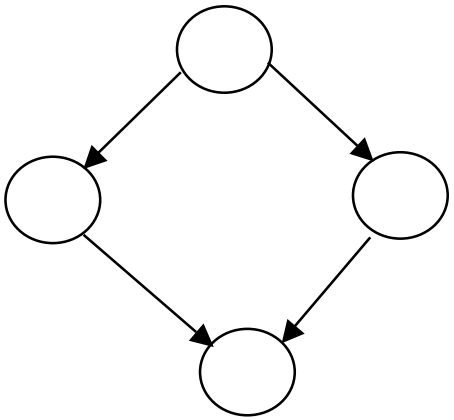
$\mu(G) = e - n + \mathbf{p}$ if phantom edges *counted* in e

$\mu(G) = e - n + \mathbf{2p}$ if phantom edges *not counted* in e

Complexities of Typical Program Graphs



$\mu(G) = e - n + 2p$
$= 3 - 4 + 2$
$= 1$

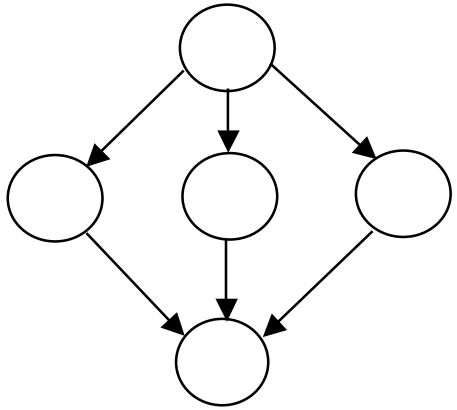


$\mu(G) = e - n + 2p$
$= 4 - 4 + 2$
$= 2$

Straight-line programs have $\mu(G) = 1$.

Two-way branch programs have $\mu(G) = 2$.

Complexities of Typical Program Graphs



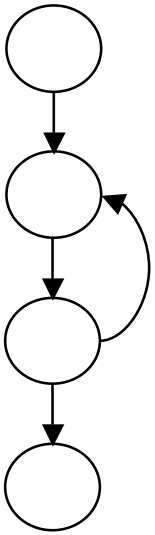
$\mu(G) = e - n + 2p$
$= 6 - 5 + 2$
$= 3$

Adding a branch increases e-n by 1, so by induction:

Simple k-way branch programs have $\mu(G) = k$.

Three-way branch programs have $\mu(G) = 3$.

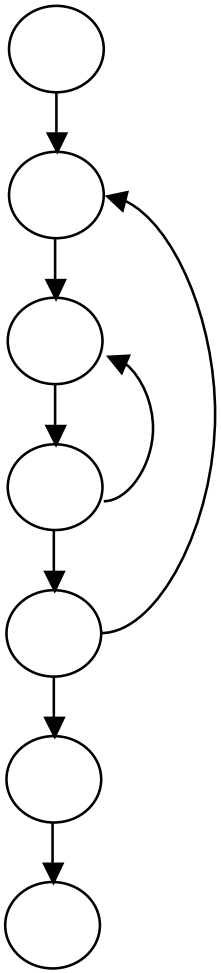
Complexities of Typical Program Graphs



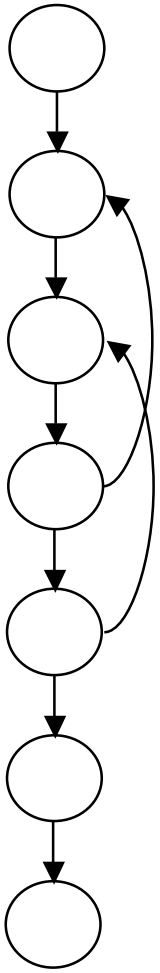
$\mu(G) = e - n + 2p$
$= 4 - 4 + 2$
$= 2$

'while' programs have $\mu(G) = 2$.

Nested



Intersecting

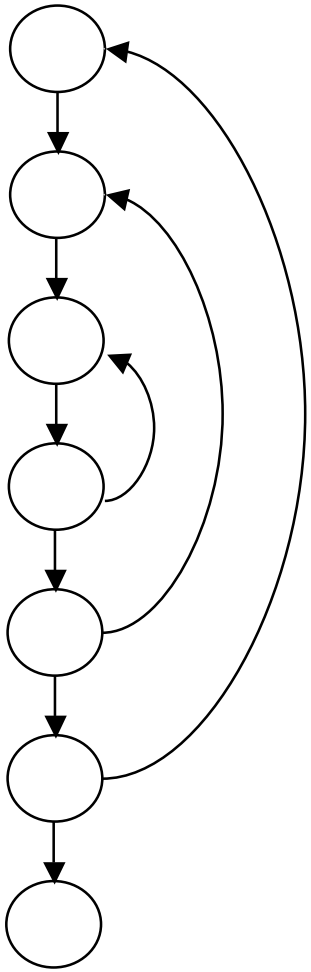


These have the same complexity. This may be misleading.

$\mu(G) = e - n + 2p$
$= 8 - 7 + 2$
$= 3$

Complexities of Typical Program Graphs

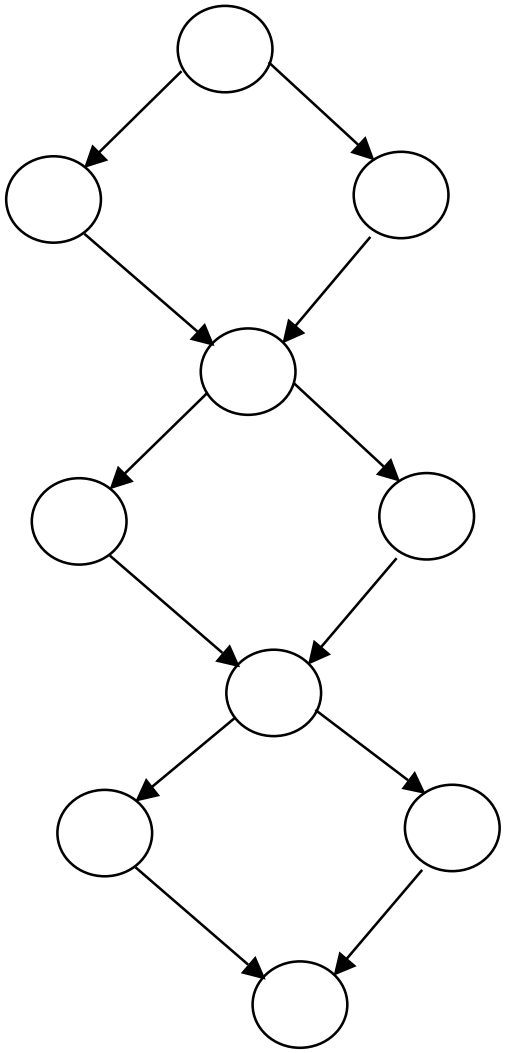
Triply-Nested



$$\mu(G) = e - n + 2p$$
$$= 9 - 7 + 2$$
$$= 4$$

Simple k-tuply nested programs have $\mu(G) = k+1$.

Complexities of Typical Program Graphs



$\mu(G) = e - n + 2p$

$= 12 \;\; 10 + 2$

$= 4$

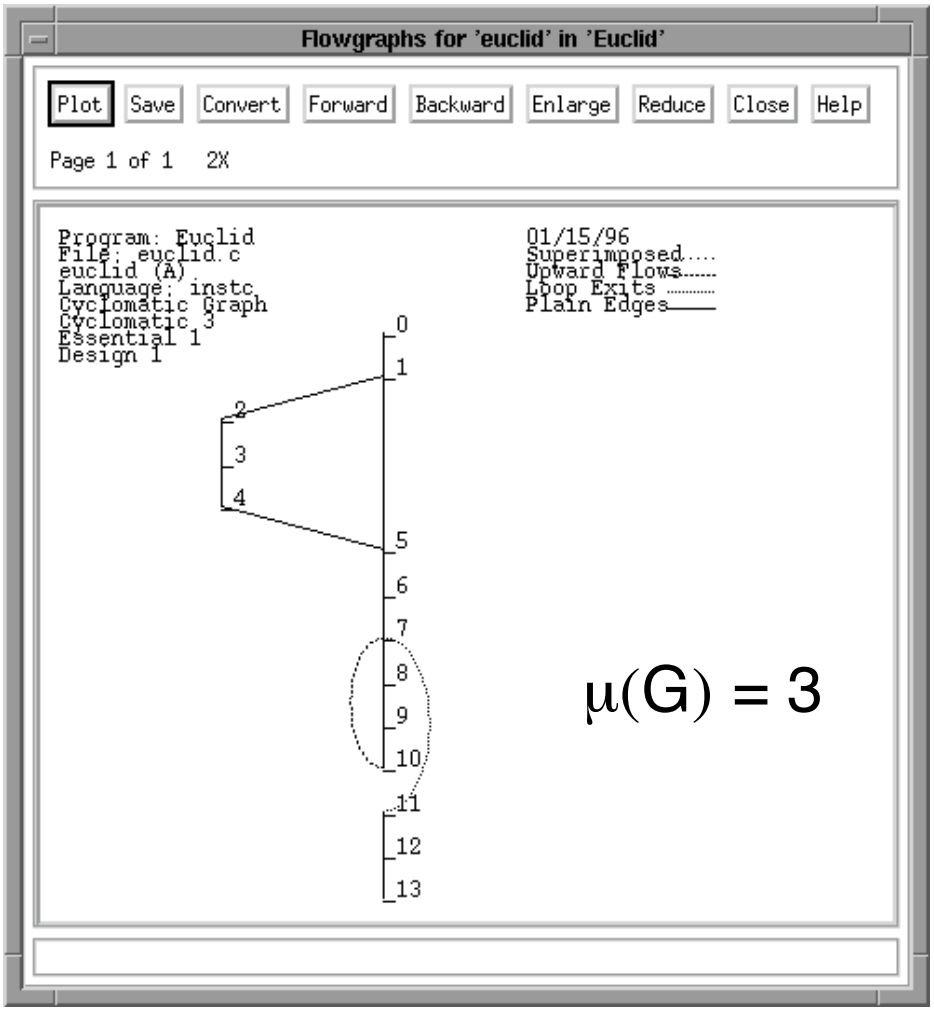
k-section “lattices”
have $\mu(G) = k+1$.

Somehow this seems to hide
the possible “combinatorial explosion.”

Uses of complexity $\mu(G)$

- $\mu(G)$ can be used to indicate the **minimum number of test cases required.**
- Keep $\mu(G)$ small (e.g. < 10) for "understandable" programs.
- Certain constructs (e.g. switch) may be exempted from the count in some organizations.

Cyclomatic Tools



$\mu(G) = 3$

```
Module: euclid
Basis Test Paths: 3 Paths

Test Path B1: 0 1 5 6 7 11 12 13

        8(    1): n>m ==> FALSE

        14(    7): r!=0 ==> FALSE

Test Path B2: 0 1 2 3 4 5 6 7 11 12 13

        8(    1): n>m ==> TRUE

        14(    7): r!=0 ==> FALSE

Test Path B3: 0 1 5 6 7 8 9 10 7 11 12 13

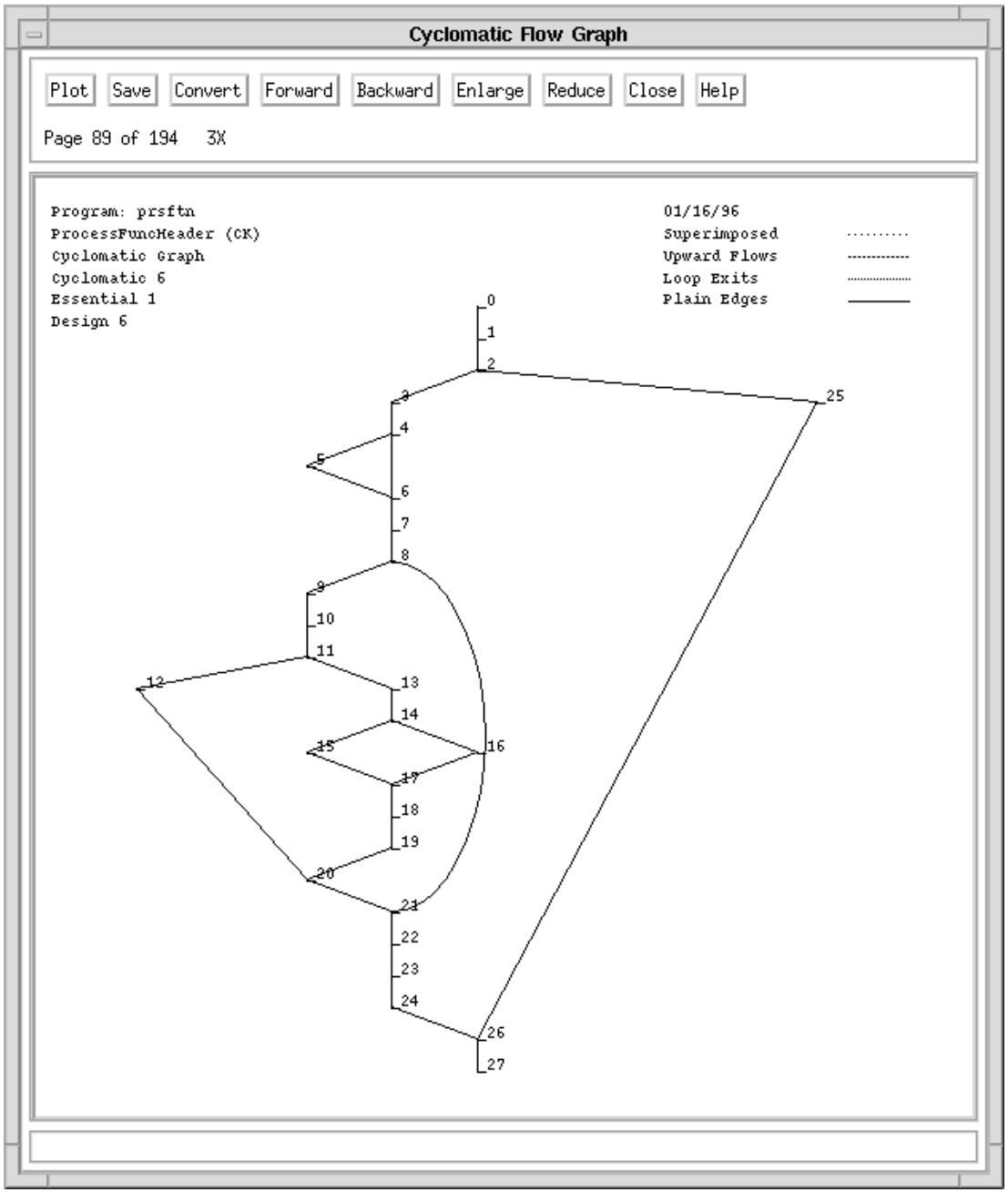
        8(    1): n>m ==> FALSE

        14(    7): r!=0 ==> TRUE

        14(    7): r!=0 ==> FALSE
```

Source: Watson & McCabe, 1996

$\mu(G) = 6$

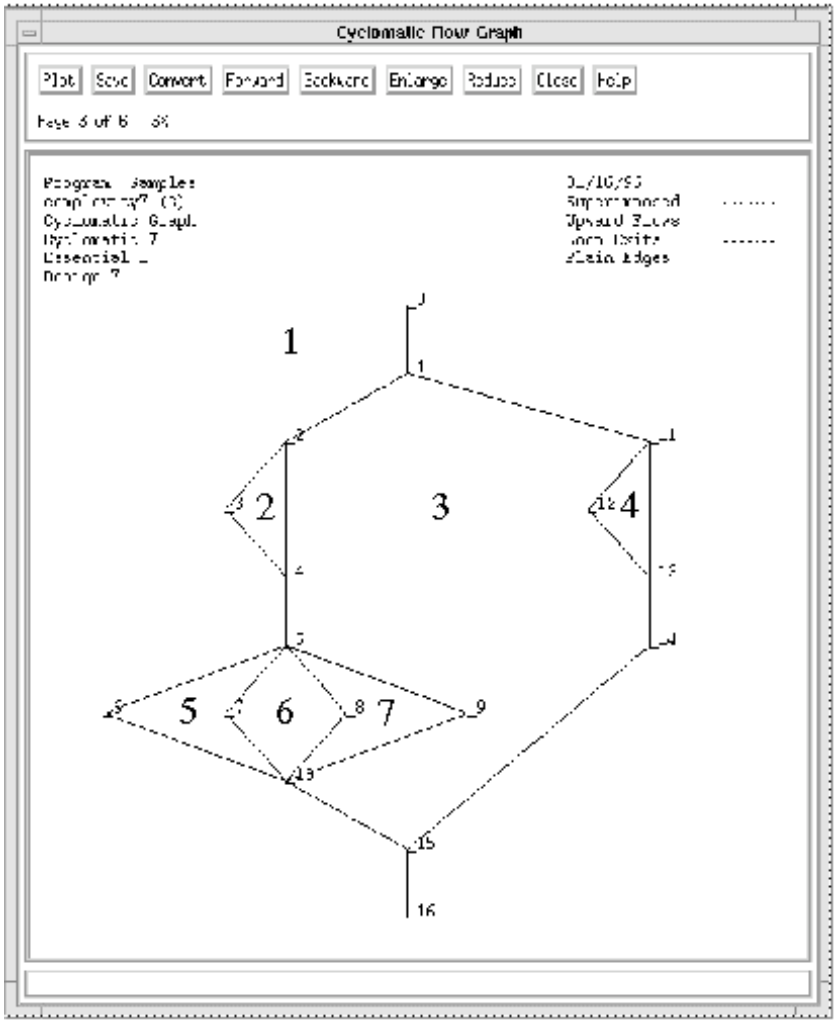

Cyclomatic Flow Graph
Plot
Save
Convert
Forward
Backward
Enlarge
Reduce
Close
Help
Page 89 of 194 3X
Program: prsftn
ProcessFuncHeader (CK)
Cyclomatic Graph
Cyclomatic 6
Essential 1
Design 6
01/16/96
Superimposed
Upward Flows
Loop Exits
Plain Edges
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27

Simplified Complexity Measures

- Counting binary predicate tests only:
  $\eta(G)$ = predicates + 1

- Counting ***regions*** = e - n + 2p
  (from Euler's formula: N-E+F=2, F = faces or regions)

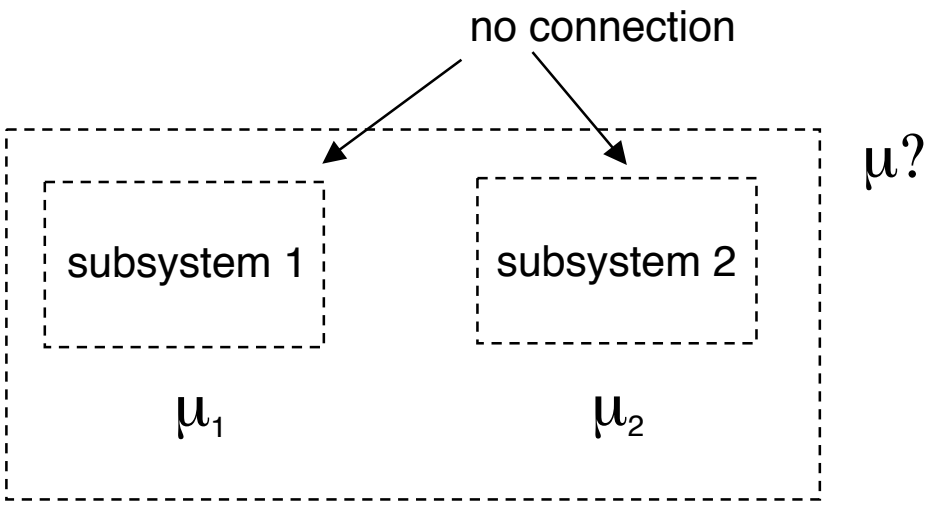
Counting Regions



regions numbered

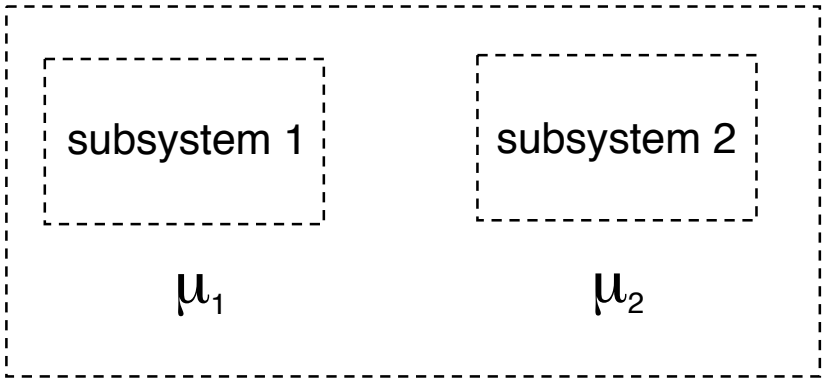
Use of μ(G) in **Integration Testing**

- Composition of modules' complexity: what is μ of overall system in terms of individual μ 's?



Use of μ(G) in Integration Testing

- Composition of modules' complexity: what is μ of overall system in terms of individual μ's?



μ?

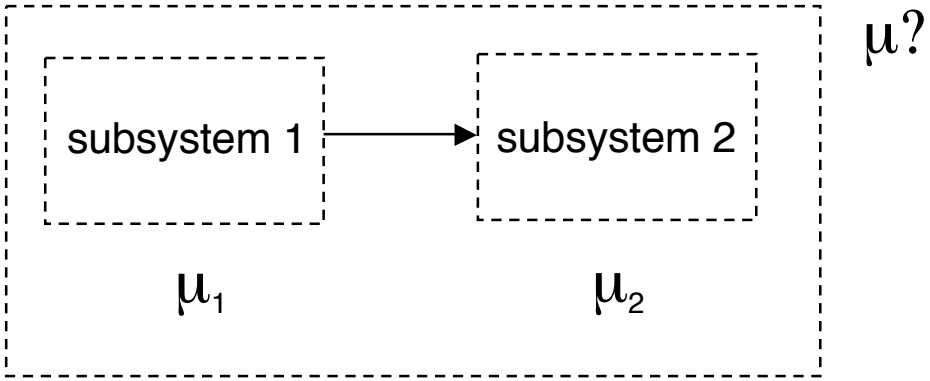
$e = e_1 + e_2$
$n = n_1 + n_2$
$p = p_1 + p_2$

$\mu = \mu_1 + \mu_2$

$\mu_1(G) = e_1 - n_1 + 2p_1$ $\mu_2(G) = e_2 - n_2 + 2p_2$

Use of $\mu(G)$ in Integration Testing

- What if connected by one edge?



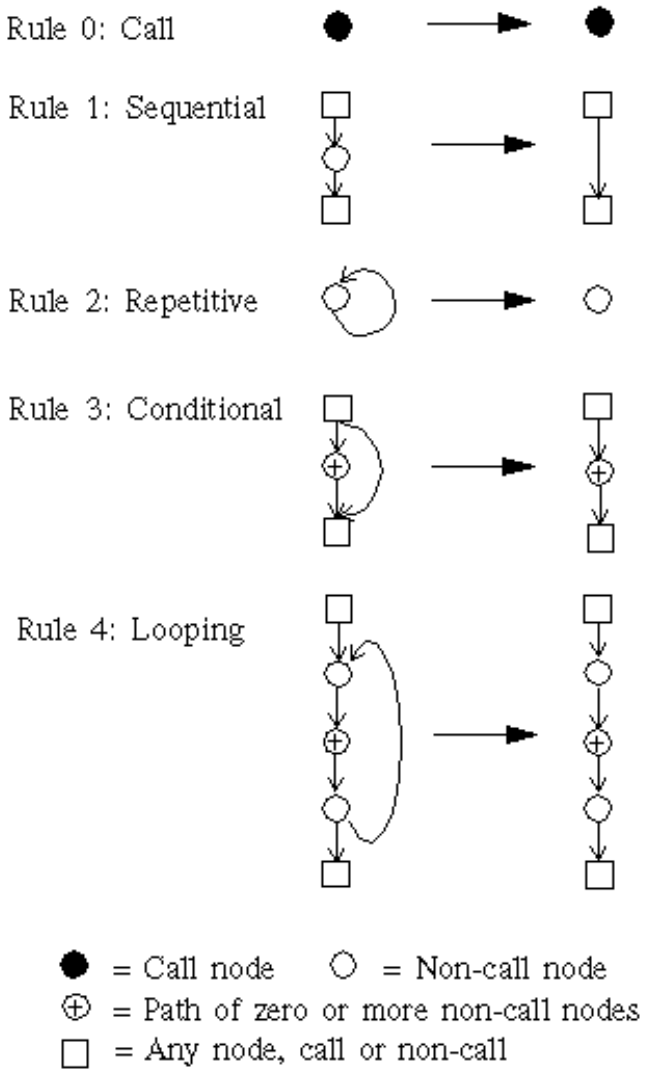
$\mu = \mu_1 + \mu_2 + 1 - 1$

one fewer separate part

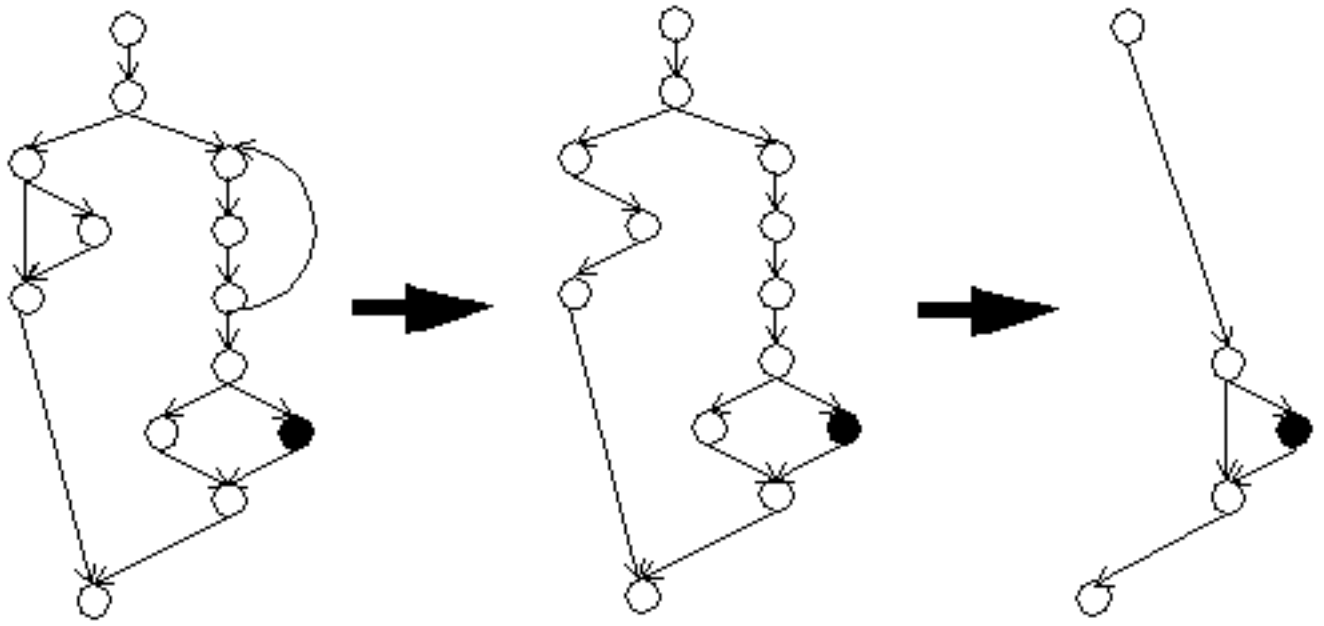
one more edge

So $\mu = \mu_1 + \mu_2$, even if the subsystems are singly connected.

"Design-Reduction" Technique: Lumping hierarchically



"Design-Reduction" Technique



Software Test Standards Reference

(Order from http://www.12207.com/test1.htm)

IEEE 829	Software Test Documentation
IEEE 1008	Software Unit Testing
IEEE 1044	Classification for Software Anomalies
IEEE 1044.1	Guide to Classification for Software Anomalies
IEEE 1074	Standard for Developing Software Life Cycle Processes
ISO/IEC 12207	Information Technology-Software Life Cycle Processes
AECL CE-1001-STD REV.1	Standard for Software Engineering of Safety Critical Software
ANSI/AAMI SW68	Medical device software - Software life cycle processes
IEC 60880	Software for Computers in the Safety Systems of Nuclear Power Stations
IEC 60601-1-4	Medical Electrical Equipment--Part 1: General Requirements for Safety-4. Collateral Standard: Programmable Electrical Medical Systems
RTCA DO-178B/ED-12B	Software Considerations in Airborne Systems and Equipment Certification