Chapter 8

Path Testing

Outline

- Preliminaries
- Program graphs
- DD-Paths
- Test coverage metrics
- Basis path testing
- Essential complexity (from McCabe)

Code-Based (Structural) Testing

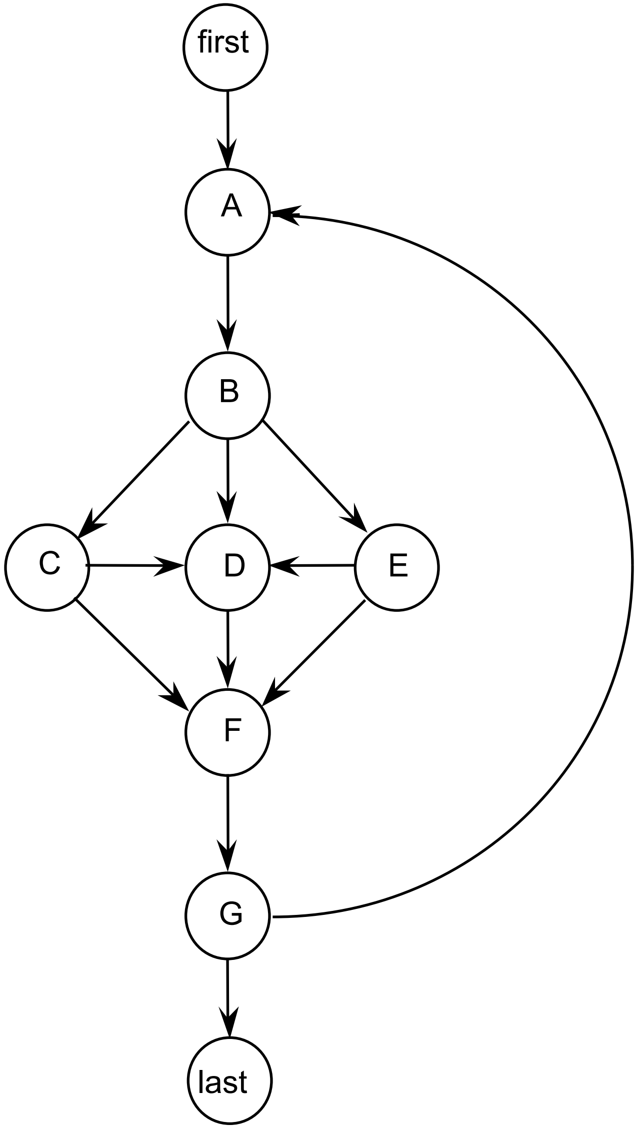
- Complement of/to Specification-based (Functional) Testing
- Based on Implementation
- Powerful mathematical formulation
  - program graph
  - define-use path
  - Program slices
- Basis for Coverage Metrics (a better answer for gaps and redundancies)
- Usually done at the unit level
- Not very helpful to identify test cases
- Extensive commercial tool support

Path Testing

- Paths derived from some graph construct.
- When a test case executes, it traverses a path.
- Huge number of paths implies some simplification is needed.
- Big Problem: infeasible paths.
- Big Question: what kinds of faults are associated with what kinds of paths?
- By itself, path testing can lead to a false sense of security.

Common Objection to Path-Based Testing

(Trillions of Paths)

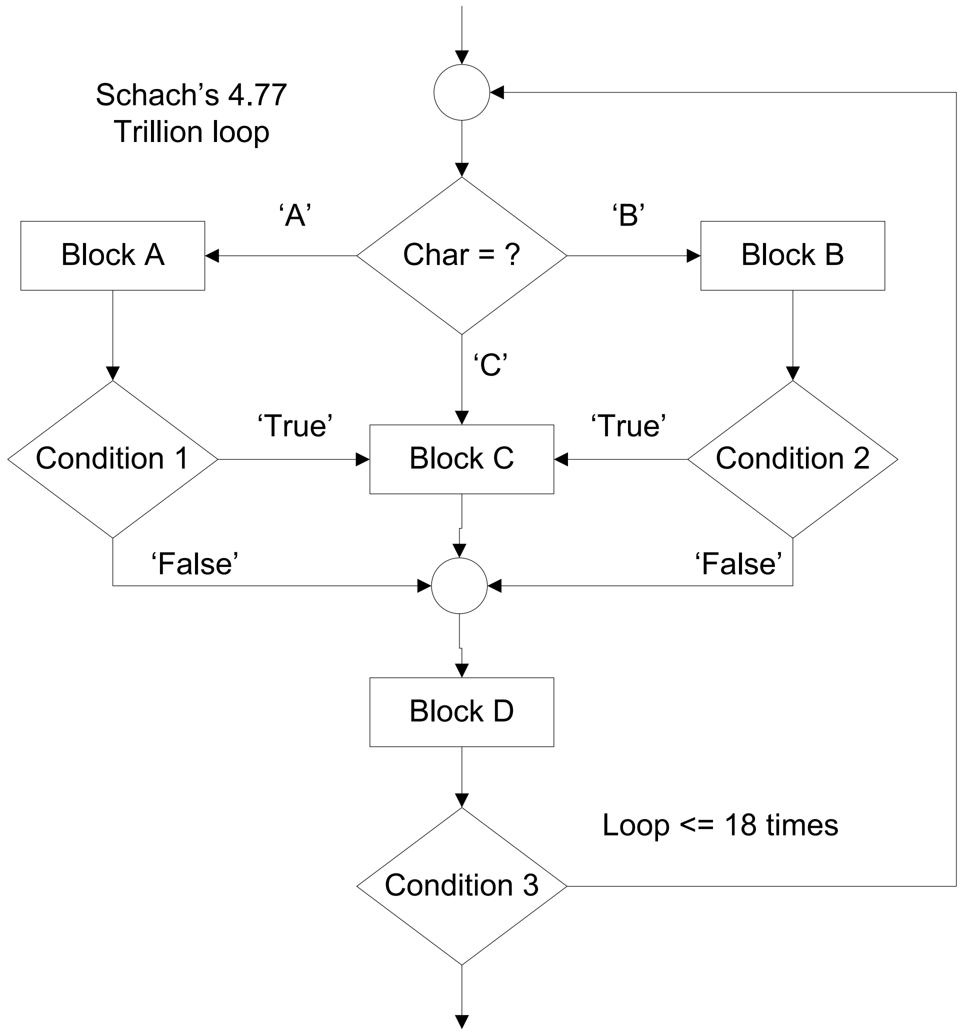


If the loop executes up to 18 times, there are 4.77 Trillion paths. Impossible, or at least, infeasible, to test them all. [Schach]

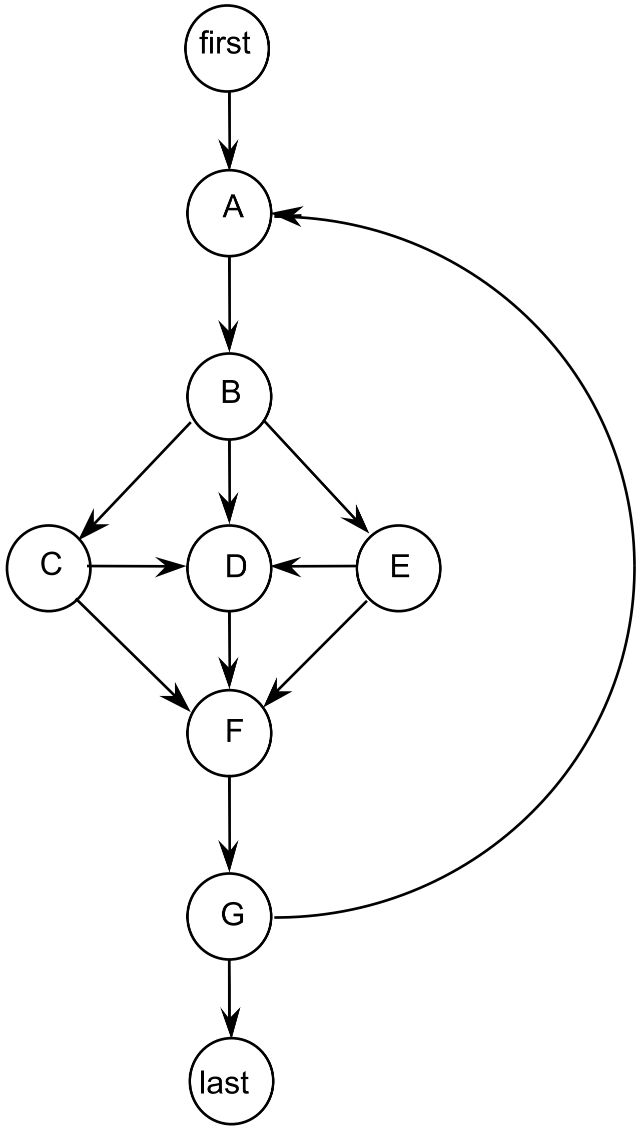
$5^0 + 5^1 + 5^2 + \ldots + 5^{18} = 4{,}768{,}371{,}582{,}030$

Stephen R. Schach, *Software Engineering*, , (2nd edition) Richard D. Irwin, Inc. and Aksen Associates, Inc. 1993

Schach's flowchart



Test Cases for Schach's "Program"



1. First-A-B-C-F-G-Last
2. First-A-B-C-D-F-G-Last
3. First-A-B-D-F-G-A-B-D-F-G-Last
4. First-A-B-E-F-G-Last
5. First-A-B-E-D-F-G-Last

These test cases cover

- Every node
- Every edge
- Normal repeat of the loop
- Exiting the loop

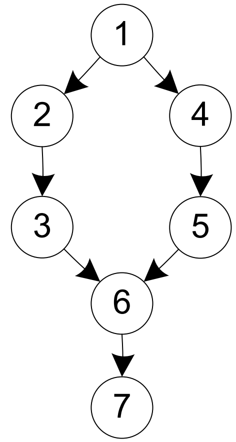
Program Graphs

Definition: Given a program written in an imperative programming language, its *program graph* is a directed graph in which nodes are statement fragments, and edges represent flow of control. (A complete statement is a “default” statement fragment.)

Program Graphs of Structured Programming Constructs

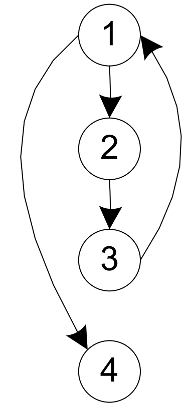
If-Then-Else

```
If <condition>
   Then
      <then statements>
   Else
      <else statements>
End If
<next statement>
```



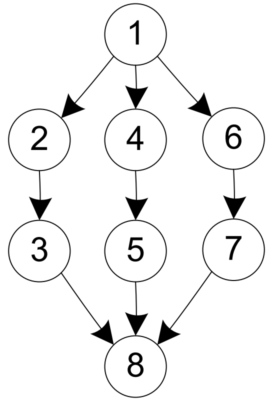
Pre-test Loop

```
While <condition>
   <repeated body>
End While
<next statement>
```



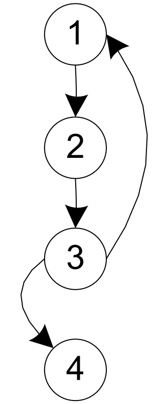
Case/Switch

```
Case  n Of 3
   n=1:
      <case 1  statements>
   n=2:
      <case 2  statements>
   n=3:
      <case 3  statements>
End Case
```



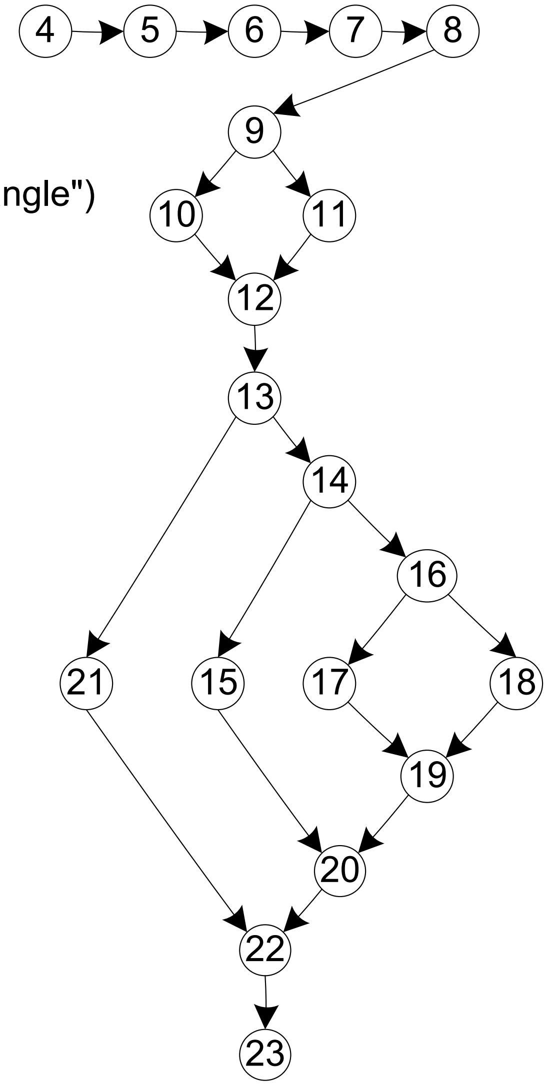
Post-test Loop

```
Do
   <repeated body>
Until <condition>
<next statement>
```



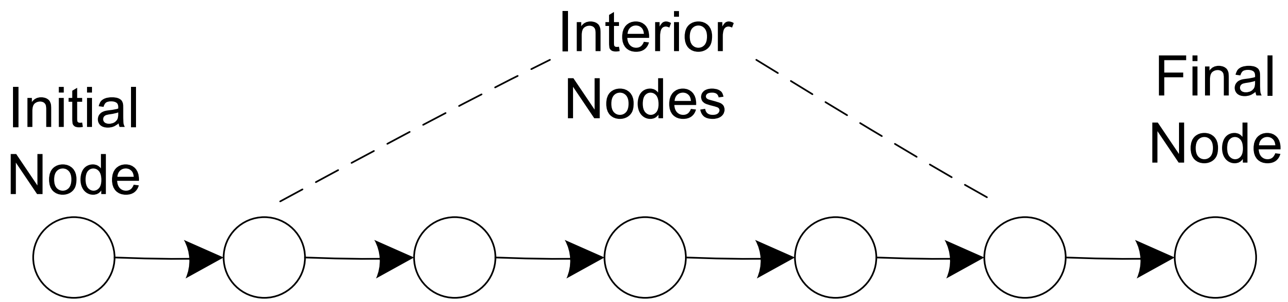
Sample Program Graph

```
1  Program triangle2
2  Dim a,b,c As Integer
3  Dim IsATriangle As Boolean
4  Output("Enter 3 integers which are sides of a triangle")
5  Input(a,b,c)
6  Output("Side A is ",a)
7  Output("Side B is ",b)
8  Output("Side C is ",c)
9  If (a < b + c) AND (b < a + c) AND (c < a + b)
10     Then IsATriangle = True
11     Else IsATriangle = False
12  EndIf
13  If IsATriangle
14     Then  If (a = b) AND (b = c)
15             Then Output ("Equilateral")
16             Else    If (a≠b) AND (a≠c) AND (b≠c)
17                         Then  Output ("Scalene")
18                         Else  Output ("Isosceles")
19                     EndIf
20             EndIf
21     Else    Output("Not a Triangle")
22  EndIf
23  End triangle2
```



DD-Paths

- Originally defined by E. F. Miller (1977?)
- "DD" is short for "decision to decision"
- Original definition was for early (second generation) programming languages
- Similar to a "chain" in a directed graph
- Bases of interesting test coverage metrics



DD-Paths

A *DD-Path* (decision-to-decision) is a chain in a program graph such that

Case 1: it consists of a single node with indeg = 0,
Case 2: it consists of a single node with outdeg = 0,
Case 3: it consists of a single node with indeg ≥ 2 or outdeg ≥ 2,
Case 4: it consists of a single node with indeg = 1 and outdeg = 1,
Case 5: it is a maximal chain of length ≥ 1.

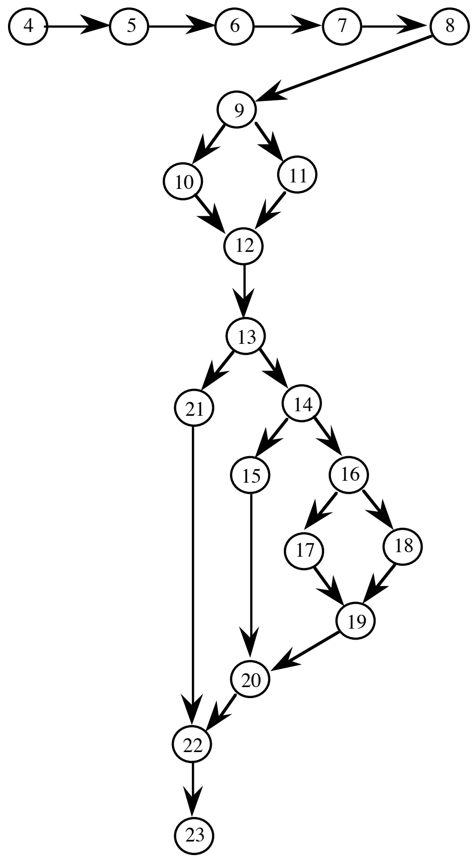
DD-Path Graph

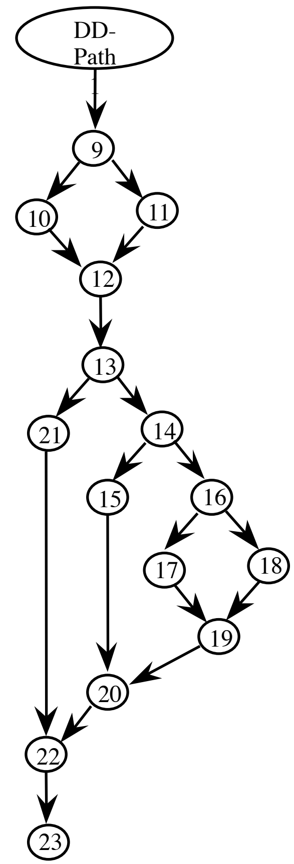
Given a program written in an imperative language, its *DD-Path graph* is the directed graph in which nodes are DD-Paths of its program graph, and edges represent control flow between successor DD-Paths.

- a form of condensation graph
- 2-connected components are collapsed into an individual node
- single node DD-Paths (corresponding to Cases 1 - 4) preserve the convention that a statement fragment is in exactly one DD-Path

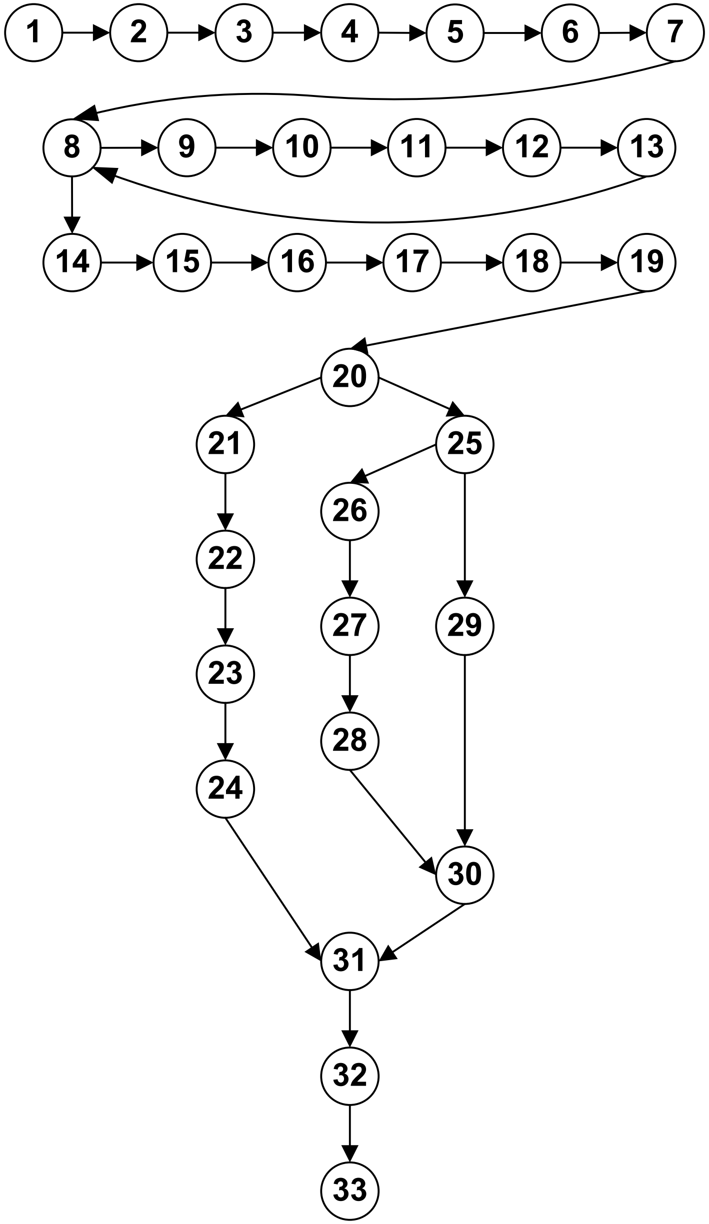
DD-Path Graph of the Triangle Program

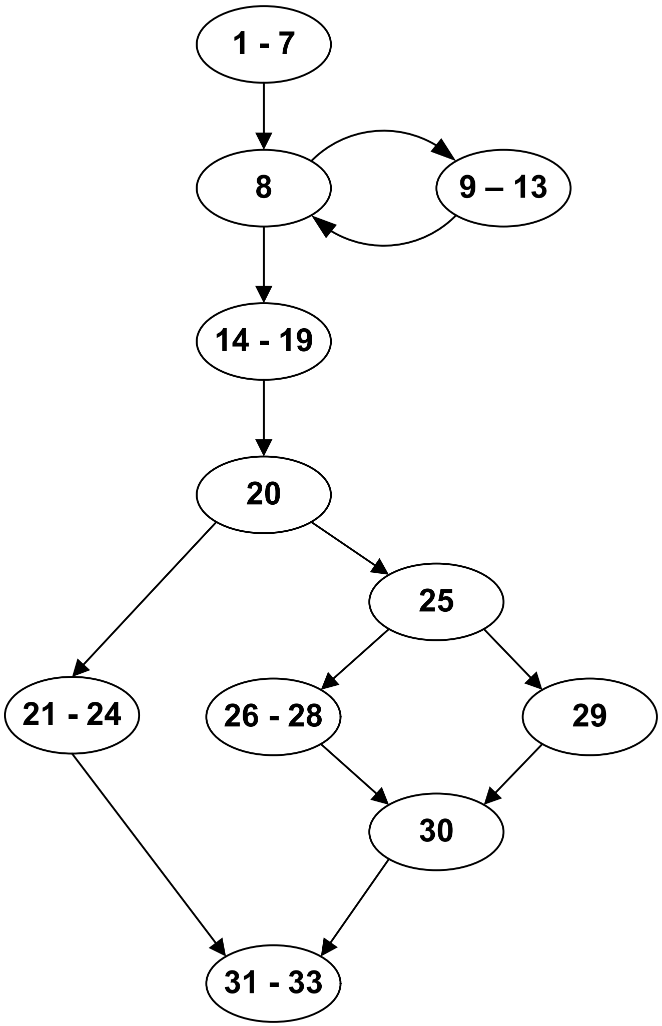
(not much compression because this example is control intensive, with little sequential code.)





DD-Path Graph of Commission Problem





Exercises and Questions

- Compute the cyclomatic complexity of
  - the commission problem program graph
  - the commission problem DD-Path graph
- Are the complexities equal?
- Repeat this for the Triangle Program examples
- What conclusions can you draw?

Code-Based Test Coverage Metrics

- Used to evaluate a given set of test cases
- Often required by
  - contract
  - U.S. Department of Defense
  - company-specific standards
- Elegant way to deal with the gaps and redundancies that are unavoidable with specification-based test cases.
- BUT
  - coverage at some defined level may be misleading
  - coverage tools are needed

Code-Based Test Coverage Metrics

(E. F. Miller, 1977 dissertation)

- C_0: Every statement
- C_1: Every DD-Path
- C_{1p}: Every predicate outcome
- C_2: C_1 coverage + loop coverage
- C_d: C_1 coverage +every pair of dependent DD-Paths
- C_{MCC}: Multiple condition coverage
- C_{ik}: Every program path that contains up to k repetitions of a loop (usually k = 2)
- C_{stat}: "Statistically significant" fraction of paths
- C_∞: All possible execution paths

Test Coverage Metrics from Program Graphs

- Every node
- Every edge
- Every chain
- Every path

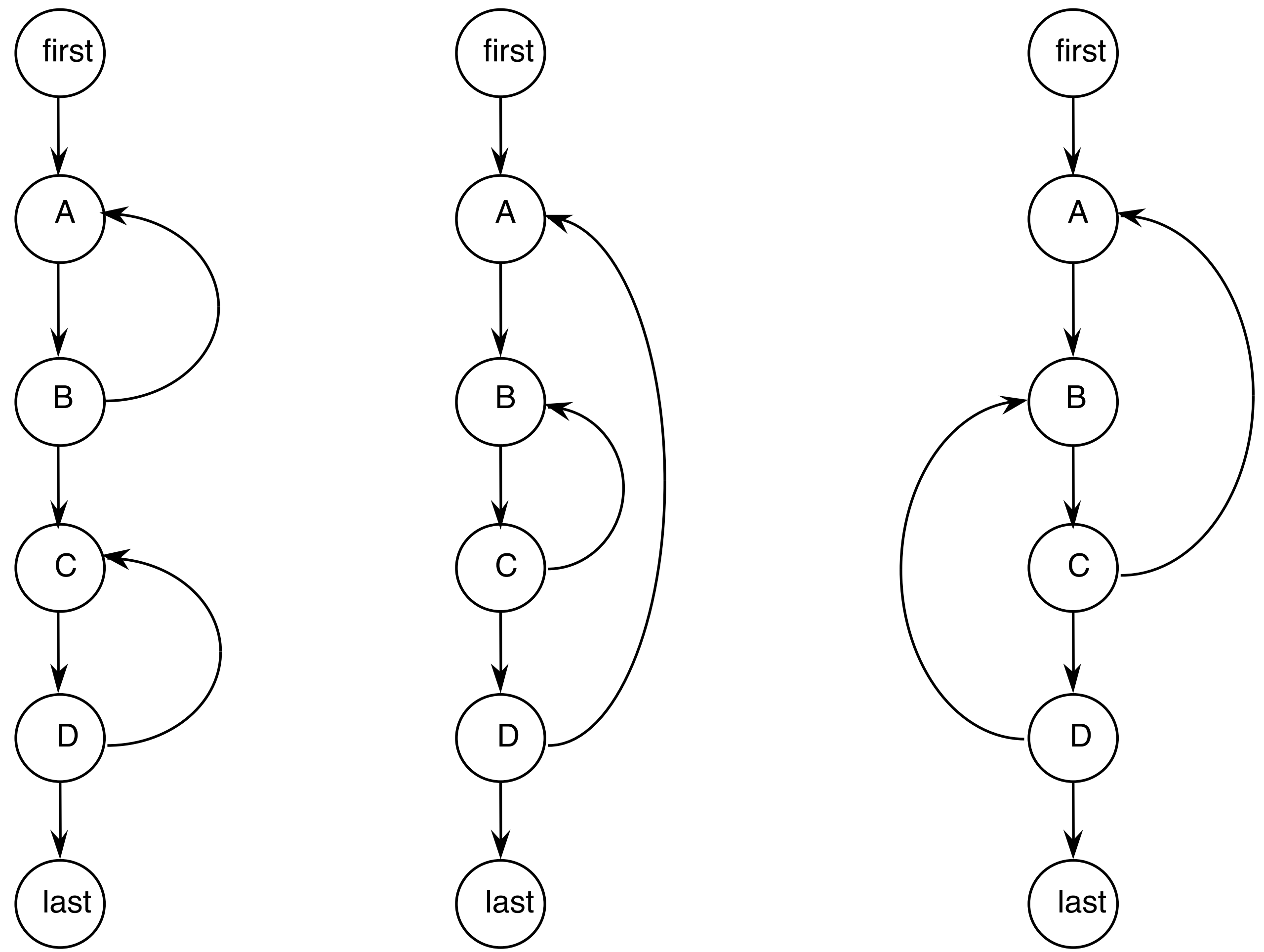
- How do these compare with Miller's coverage metrics?

Testing Loops

Huang's Theorem: (Paraphrased)
Everything interesting will happen in two loop traversals: the normal loop traversal and the exit from the loop.

Exercise:
Discuss Huang's Theorem in terms of graph based coverage metrics.

Concatenated, Nested, and Knotted Loops



Strategy for Loop Testing

- Huang's theorem suggests/assures 2 tests per loop is sufficient. (Judgment required, based on reality of the code.)
- For nested loops:
  - Test innermost loop first
  - Then "condense" the loop into a single node (as in condensation graph, see Chapter 4)
  - Work from innermost to outermost loop
- For concatenated loops: use Huang's Theorem
- For knotted loops: Rewrite! (see McCabe's cyclomatic and essential complexity)

Multiple Condition Testing

- Consider the multiple condition as a logical proposition, i.e., some logical expression of simple conditions.
- Make the truth table of the logical expression.
- Convert the truth table to a decision table.
- Develop test cases for each rule of the decision table (except the impossible rules, if any).
- Next 3 slides: multiple condition testing for

```
If (a < b + c) AND (b < a + c) AND (c < a + b)
    Then IsATriangle = True
    Else  IsATriangle = False
Endif
```

Truth Table for Triangle Inequality

$(a<b+c)$ AND $(b<a+c)$ AND $(c<a+b)$

$(a<b+c)$	$(b<a+c)$	$(c<a+b)$	$(a<b+c)$ AND $(b<a+c)$ AND $(c<a+b)$
T	T	T	T
T	T	F	F
T	F	T	F
T	F	F	F
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	F

Decision Table for

(a<b+c) AND (b<a+c) AND (c<a+b)

c1: a<b+c	T	T	T	T	F	F	F	F
c2: b<a+c	T	T	F	F	T	T	F	F
c3: c<a+b	T	F	T	F	T	F	T	F
a1: impossible				X		X	X	X
a2:Valid test case #	1	2	3		4			

Multiple Condition Test Cases for

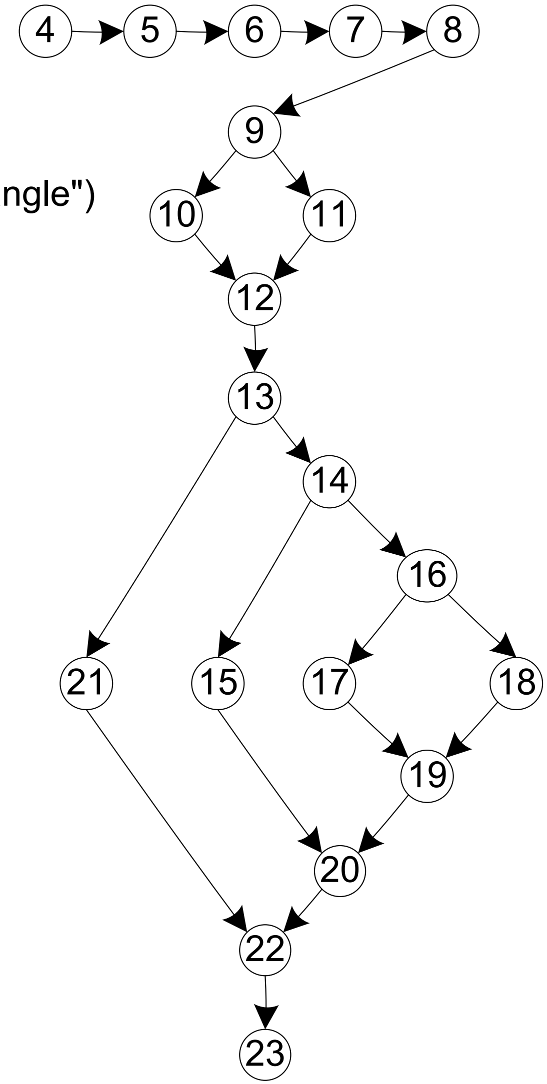
(a<b+c) AND (b<a+c) AND (c<a+b)

Test Case		a	b	c	expected output
1	all true	3	4	5	TRUE
2	$c \geq a + b$	3	4	9	FALSE
3	$b \geq a + c$	3	9	4	FALSE
4	$a \geq b + c$	9	3	4	FALSE

Note: could add test cases for $c = a + b$, $b = a + c$, and $a = b + c$.

Program Graph for the Triangle Program

```
1  Program triangle2
2  Dim a,b,c As Integer
3  Dim IsATriangle As Boolean
4  Output("Enter 3 integers which are sides of a triangle")
5  Input(a,b,c)
6  Output("Side A is ",a)
7  Output("Side B is ",b)
8  Output("Side C is ",c)
9  If (a < b + c) AND (b < a + c) AND (c < a + b)
10    Then IsATriangle = True
11    Else IsATriangle = False
12  EndIf
13  If IsATriangle
14    Then  If (a = b) AND (b = c)
15            Then Output ("Equilateral")
16            Else    If (a≠b) AND (a≠c) AND (b≠c)
17                        Then  Output ("Scalene")
18                        Else  Output ("Isosceles")
19                    EndIf
20            EndIf
21    Else    Output("Not a Triangle")
22  EndIf
23  End triangle2
```



Dependent DD-Paths
(often correspond to infeasible paths)

- Look at the Triangle Program code and program graph
- If a path traverses node 10 (Then IsATriangle = True), then it must traverse node 14.
- Similarly, if a path traverses node 11 (Else IsATriangle = False), then it must traverse node 21.
- Paths through nodes 10 and 21 are infeasible.
- Similarly for paths through 11 and 14.
- Hence the need for the C_d coverage metric.

Test Coverage for Compound Conditions

- Extension of/to Multiple Condition Testing
- Modified Condition Decision Coverage (MCDC)
- Required for "Level A" software by DO-178B
- Three variations*
  - Masking MCDC
  - Unique-Cause MCDC
  - Unique-Cause + Masking MCDC
- Masking MCDC is
  - the weakest of the three, AND
  - is recommended for DO-178B compliance

* Chilenski, John Joseph, "An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion," DOT/FAA/AR-01/18, April 2001.

[http://www.faa.gov/about/office_org/headquarters_offices/ang/offices/tc/library/]

Chilenski's Definitions

- Conditions can be either *simple* or *compound*
  - isATriangle is a simple condition
  - (a < b + c) AND (b < a + c) is a compound condition
- Conditions are *strongly coupled* if changing one always changes the other.
  - (x = 0) and (x ≠ 0) are strongly coupled in
    ((x = 0) AND a) OR ((x ≠ 0) AND b)
- Conditions are *weakly coupled* if changing one may change one but not all of the others.
  - All three conditions are weakly coupled in
  ((x = 1) OR (x = 2) OR (x = 3))
- *"Masking"* is based on the Domination Laws
  - (x AND false)
  - (x OR true)

MCDC requires...

- Every statement must be executed at least once,
- Every program entry point and exit point must be invoked at least once,
- All possible outcomes of every control statement are taken at least once,
- Every non-constant Boolean expression has been evaluated to both True and False outcomes,
- Every non-constant condition in a Boolean expression has been evaluated to both True and False outcomes, and
- Every non-constant condition in a Boolean expression has been shown to independently affect the outcomes (of the expression).

MCDC Variations

- "*Unique-Cause MCDC* [requires] a unique cause (toggle a single condition and change the expression result) for all possible (uncoupled) conditions."

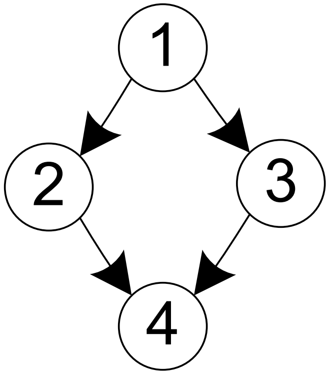
- "*Unique-Cause + Masking MCDC* [requires] a unique cause (toggle a single condition and change the expression result) for all possible (uncoupled) conditions. In the case of strongly coupled conditions, masking [is allowed} for that condition only, i.e., all other (uncoupled) conditions will remain fixed."

MCDC Variations (continued)

- "*Masking MCDC* allows masking for all conditions, coupled and uncoupled. (toggle a single condition and change the expression result) for all possible (uncoupled) conditions. In the case of strongly coupled conditions, masking [is allowed} for that condition only (i.e., all other (uncoupled) conditions will remain fixed."

Example

1. If (a AND (b OR c))
2. Then y = 1
3. Else y = 2
4. EndIf



Decision Table for (a AND (b OR c))

Conditions	rule 1	rule 2	rule 3	rule 4	rule 5	rule 6	rule 7	rule 8
a	T	T	T	T	F	F	F	F
b	T	T	F	F	T	T	F	F
c	T	F	T	F	T	F	T	F
a AND (b OR c)	True	True	True	False	False	False	False	False
Actions								
y = 1	x	x	x	—	—	—	—	—
y = 2	—	—	—	x	x	x	x	x

For MCDC Coverage of (a AND (b OR c))

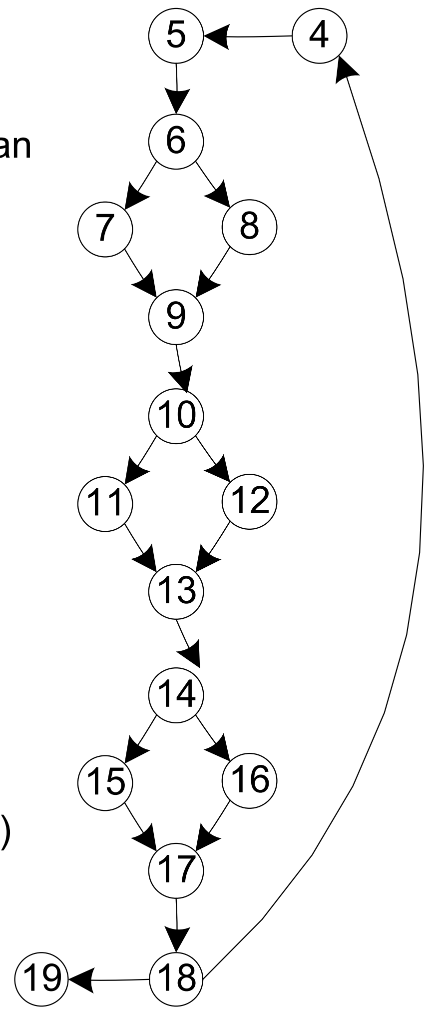
- Rules 1 and 5 toggle condition a
- Rules 2 and 4 toggle condition b
- Rules 3 and 4 toggle condition c

- If we expand (a AND (b OR c)) to

  ((a AND b) OR (a AND C)), we cannot do unique cause testing on variable a because it appears in both sub-expressions.

A NextDate Example

```
NextDate Fragment
Dim day, month, year As Integer
Dim dayOK, monthOK, yearOK As Boolean
Do
      Input(day, month, year)
      If 0 < day < 32
          Then dayOK  = True
          Else  dayOK = False
      EndIf
    If 0 < month < 13
        Then monthOK  = True
        Else  monthOK  = False
    EndIf
    If 1811 < year < 2013
        Then yearOK = True
        Else  yearOK = False
    EndIf
Until (dayOK AND monthOK AND yearOK)
End Fragment
```



Corresponding Decision Table

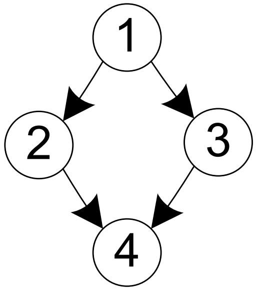
Conditions	**rule 1**	**rule 2**	**rule 3**	**rule 4**	**rule 5**	**rule 6**	**rule 7**	**rule 8**
dayOK	T	T	T	T	F	F	F	F
monthOK	T	T	F	F	T	T	F	F
YearOK	T	F	T	F	T	F	T	F
The Until condition	True	False	False	False	False	False	False	False
Actions								
Leave the loop	x	—	—	—	—	—	—	—
Repeat the loop	—	x	x	x	x	x	x	x

Test Cases and Coverage

- Decision Coverage: Rule 1 and any of Rules 2 – 8
- Multiple Condition Coverage: all eight rules are needed
- Modified Condition Decision Coverage:
  - rules 1 and 2 toggle yearOK
  - rules 1 and 3 toggle monthOK
  - rules 1 and 4 toggle dayOK

One more example

1. If (a < b + c) AND (a < b + c) AND (a < b + c)
2.   Then IsA Triangle = True
3.   Else   IsA Triangle = False
4. EndIf



In the three conditions, there are interesting dependencies that create four impossible rules.

Corresponding Decision Table

Conditions	**rule 1**	**rule 2**	**rule 3**	**rule 4**	**rule 5**	**rule 6**	**rule 7**	**rule 8**
(a < b + c)	T	T	T	T	F	F	F	F
(b < a + c)	T	T	F	F	T	T	F	F
(c < a + b)	T	F	T	F	T	F	T	F
IsATriangle = True	x	—	—	—	—	—	—	—
IsATriangle = False	—	x	x	—	x	—	—	—
impossible	—	—	—	x		x	x	x

Two Strategies of MCDC Testing

- Rewrite the code as a decision table
  - algebraically simplify
  - watch for impossible rules
  - eliminate masking when possible
- Rewrite the code as nested If logic
  - (see example of the Triangle Program fragment on the next slide)

Test Cases and Coverage

- Decision Coverage: Rule 1 and Rule 2. (also, rules 1 and 3, or rules 1 and 5.
- Rules 4, 6, 7, and 8 are impossible.
- Condition Coverage
  - rules 1 and 2 toggle ($c < a + b$)
  - rules 1 and 3 toggle ($b < a + c$)
  - rules 1 and 5 toggle ($a < b + c$)
- Modified Condition Decision Coverage:
  - rules 1 and 2 toggle ($c < a + b$)
  - rules 1 and 3 toggle ($b < a + c$)
  - rules 1 and 5 toggle ($a < b + c$)

Code-Based Testing Strategy

- Start with a set of test cases generated by an "appropriate" (depends on the nature of the program) specification-based test method.
- Look at code to determine appropriate test coverage metric.
  - Loops?
  - Compound conditions?
  - Dependencies?
- If appropriate coverage is attained, fine.
- Otherwise, add test cases to attain the intended test coverage.

Test Coverage Tools

- Commercial test coverage tools use “instrumented” source code.
  - New code added to the code being tested
  - Designed to “observe” a level of test coverage
- When a set of test cases is run on the instrumented code, the designed test coverage is ascertained.
- Strictly speaking, running test cases in instrumented code is not sufficient
  - Safety critical applications require tests to be run on actual (delivered, non-instrumented) code.
  - Usually addressed by mandated testing standards.

Sample DD-Path Instrumentation

(values of array DDpathTraversed are set to 1 when corresponding instrumented code is executed.)

```
    DDpathTraversed(1) = 1
    Output("Enter 3 integers which are sides of a triangle")
    Input(a,b,c)
    Output("Side A is ",a)
    Output("Side B is ",b)
    Output("Side C is ",c)
   'Step 2:  Is A Triangle?
    DDpathTraversed(2) = 1
    If (a < b + c) AND (b < a + c) AND (c < a + b)
    Then  DDpathTraversed(3) = 1
            IsATriangle = True
    Else  DDpathTraversed(4) = 1
            IsATriangle = False
    EndIf
```

Instrumentation Exercise

- How could you instrument the Triangle Program to record how many times a set of test cases traverses the individual DD-Paths?
- Is this useful information?

Basis Path Testing

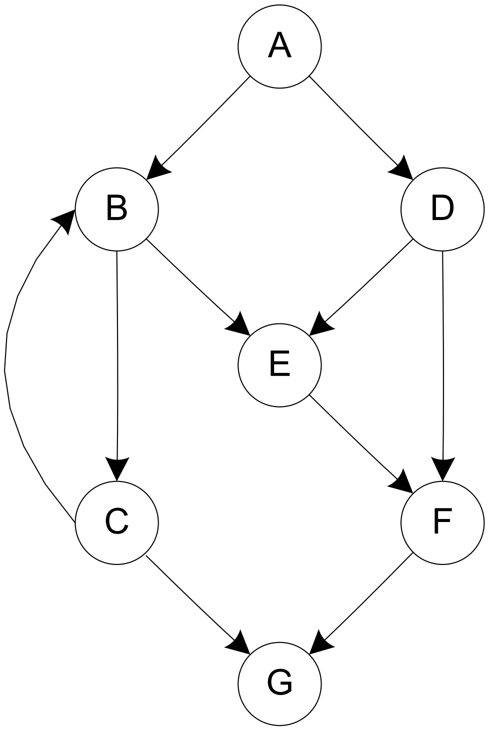
- Proposed by Thomas McCabe in 1982
- Math background
  - a Vector Space has a set of independent vectors called basis vectors
  - every element in a vector space can be expressed as a linear combination of the basis vectors
- Example: Euclidean 3-space has three basis vectors
  - (1, 0, 0) in the x direction
  - (0, 1, 0) in the y direction
  - (0, 0, 1) in the z direction
- The Hope: If a program graph can be thought of as a vector space, there should be a set of basis vectors. Testing them tests many other paths.

(McCabe) Basis Path Testing

- in math, a basis "spans" an entire space, such that everything in the space can be derived from the basis elements.
- the cyclomatic number of a strongly connected directed graph is the number of linearly independent cycles.
- given a program graph, we can always add an edge from the sink node to the source node to create a strongly connected graph. (assuming single entry, single exit)
- computing $V(G) = e - n + p$ from the modified program graph yields the number of independent paths that must be tested.
- since all other program execution paths are linear combinations of the basis path, it is necessary to test the basis paths. (Some say this is sufficient; but that is problematic.)
- the next few slides follow McCabe's original example.

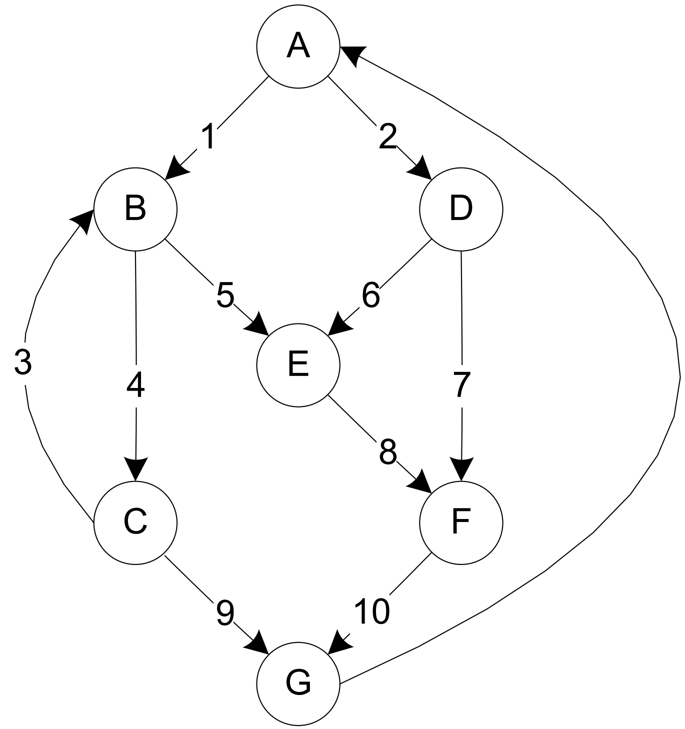
McCabe's Example

McCabe's Original Graph



$$V(G) = 10 - 7 + 2(1) = 5$$

Derived, Strongly Connected Graph



$$V(G) = 11 - 7 + 1 = 5$$

McCabe's Baseline Method

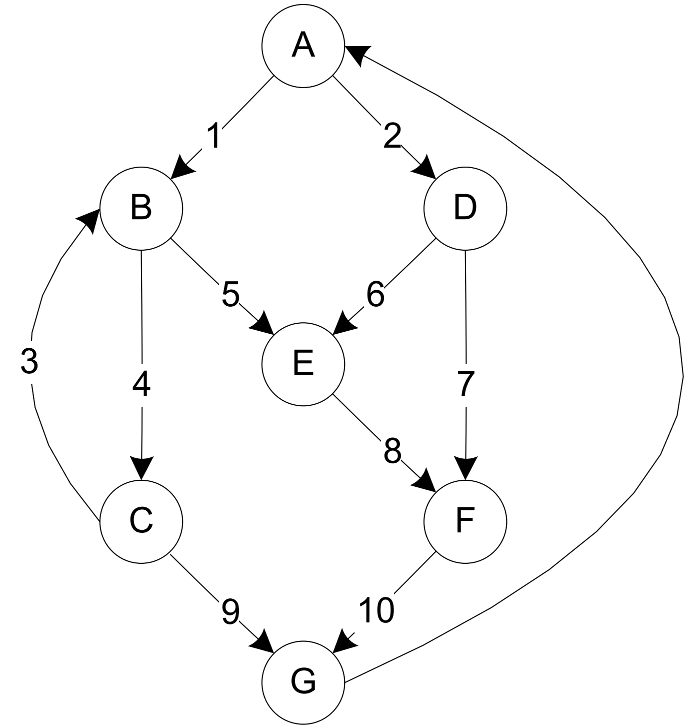
- Pick a "baseline" path that corresponds to normal execution. (The baseline should have as many decisions as possible.)
- To get succeeding basis paths, retrace the baseline until you reach a decision node. "Flip" the decision (take another alternative) and continue as much of the baseline as possible.
- Repeat this until all decisions have been flipped. When you reach V(G) basis paths, you're done.
- If there aren't enough decisions in the first baseline path, find a second baseline and repeat steps 2 and 3.

Following this algorithm, we get basis paths for McCabe's example.

McCabe’s Example (with numbered edges)

Resulting basis paths

First baseline path
p1: A, B, C, G
Flip decision at C
p2: A, B, C, B, C, G
Flip decision at B
p3: A, B, E, F, G
Flip decision at A
p4: A, D, E, F, G
Flip decision at D
p5: A, D, F, G



Path/Edge Incidence

Path / Edges	*e1*	*e2*	*e3*	*e4*	*e5*	*e6*	*e7*	*e8*	*e9*	*e10*
p1: A, B, C, G	1	0	0	1	0	0	0	0	1	0
p2: A, B, C, B, C, G	1	0	**1**	2	0	0	0	0	1	0
p3: A, B, E, F, G	1	0	0	0	**1**	0	0	1	0	1
p4: A, D, E, F, G	0	1	0	0	0	**1**	0	1	0	1
p5: A, D, F, G	0	1	0	0	0	0	**1**	0	0	1
ex1: A, B, C, B, E, F, G	1	0	1	1	1	0	0	1	0	1
ex2: A, B, C, B, C, B, C, G	1	0	2	3	0	0	0	0	1	0

Sample paths as linear combinations of basis paths

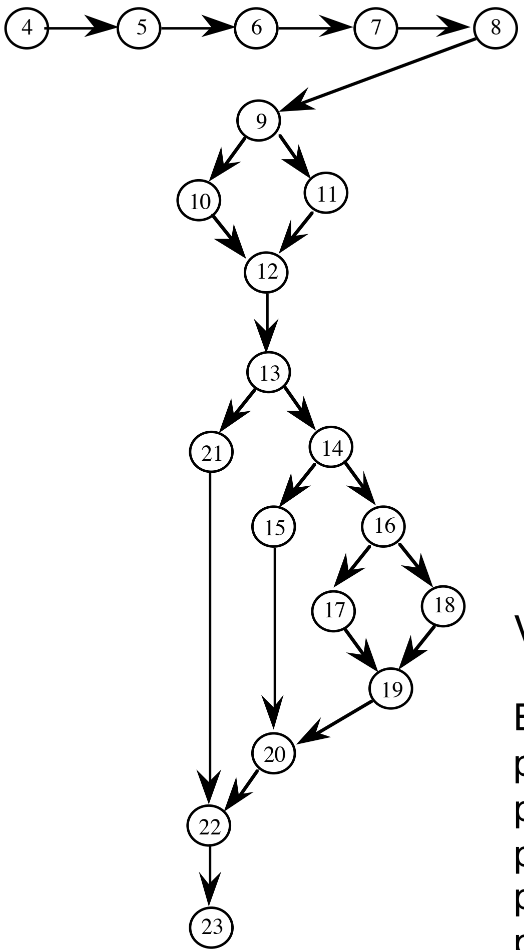
ex1 = p2 + p3 - p1

ex2 = 2p2 - p1

Problems with Basis Paths

- What is the significance of a path as a linear combination of basis paths?
- What do the coefficients mean? What does a minus sign mean?
- In the path ex2 = 2p2 - p1 should a tester run path p2 twice, and then not do path p1 the next time? This is theory run amok.
- Is there any guarantee that basis paths are feasible?
- Is there any guarantee that basis paths will exercise interesting dependencies?

McCabe Basis Paths in the Triangle Program



There are 8 topologically possible paths.
4 are feasible, and 4 are infeasible.

Exercise: Is every basis path feasible?

V(G) = 23 - 20 + 2(1) = 5

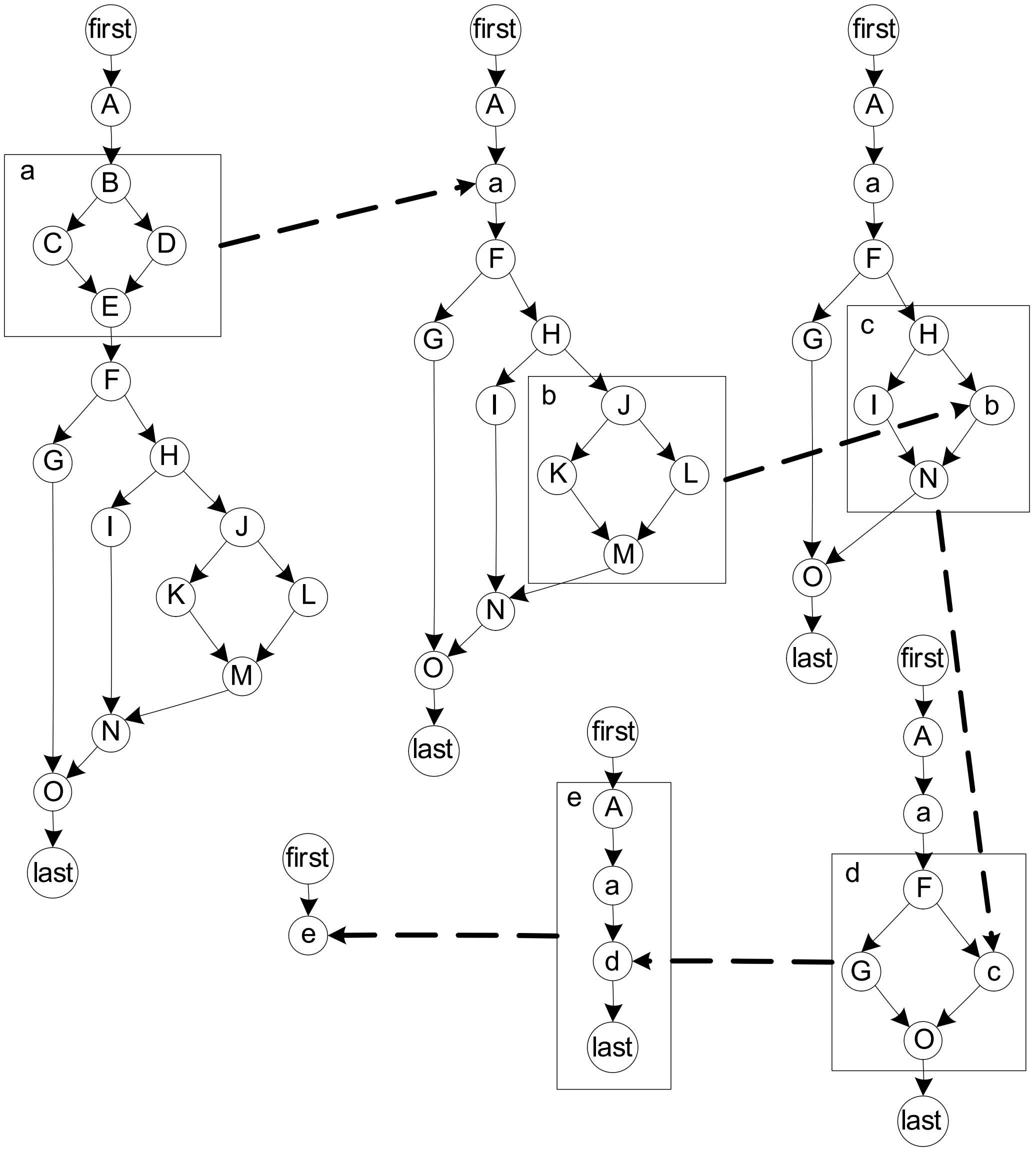
Basis Path Set B1

p1: 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 16, 18, 19, 20, 22, 23 (mainline)
p2: 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 16, 18, 19, 20, 22, 23 (flipped at 9)
p3: 4, 5, 6, 7, 8, 9, 11, 12, 13, 21, 22, 23 (flipped at 13)
p4: 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 15, 20, 22, 23 (flipped at 14)
p5: 4, 5, 6, 7, 8, 9, 11, 12, 13, 14, 16, 17, 19, 20, 22, 23 (flipped at 16)

Essential Complexity

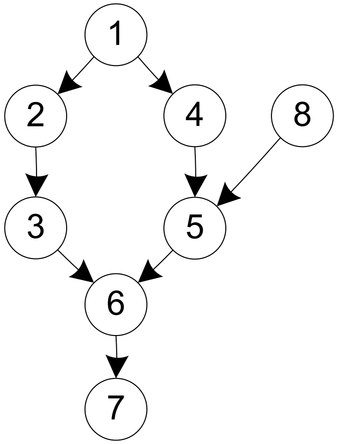
- McCabe's notion of Essential Complexity deals with the extent to which a program violates the precepts of Structured Programming.
- To find Essential Complexity of a program graph,
  - Identify a group of source statements that corresponds to one of the basic Structured Programming constructs.
  - Condense that group of statements into a separate node (with a new name)
  - Continue until no more Structured Programming constructs can be found.
  - The Essential Complexityof the original program is the cyclomatic complexity of the resulting program graph.
- The essential complexity of a Structured Program is 1.
- Violations of the precepts of Structured Programming increase the essential complexity.

Condensation with Structured Programming Constructs

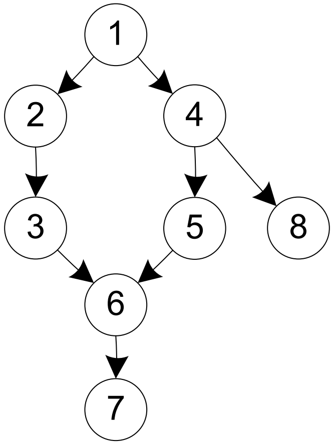


Violations of Structured Programming Precepts

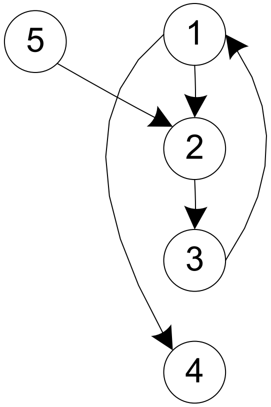
Branching into a decision



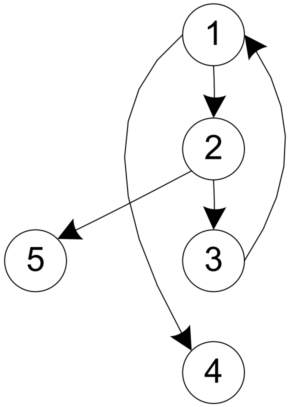
Branching out of a decision



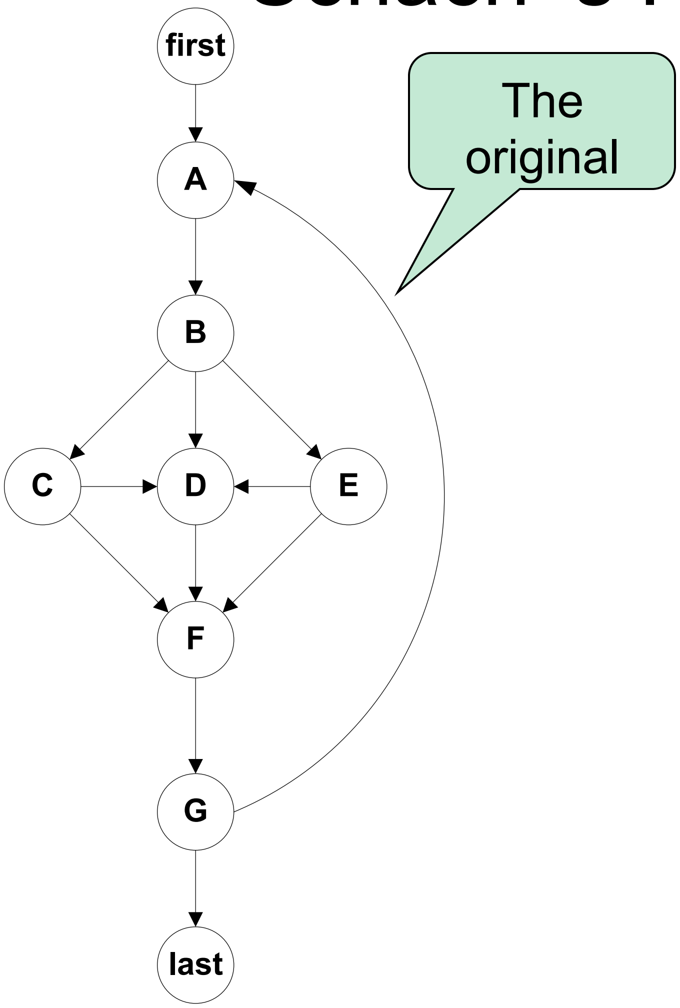
Branching into a loop

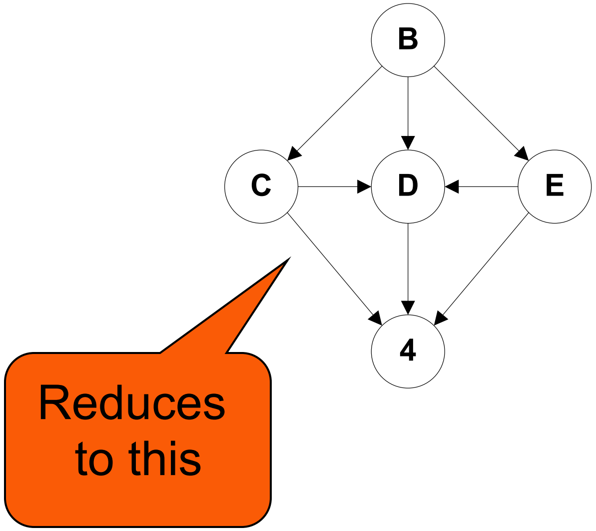


Branching out of a loop



Essential Complexity of Schach's Program Graph





V(G) = 8 – 5 + 2(1)
= 5

Essential complexity is 5

Cons and Pros of McCabe's Work

- Issues
  - Linear combinations of execution paths are counter-intuitive. What does 2p2 – p1 really mean?
  - How does the baseline method guarantee feasible basis paths?
  - Given a set of feasible basis paths, is this a sufficient test?

- Advantages
  - McCabe's approach does address both gaps and redundancies.
  - Essential complexity leads to better programming practices.
  - McCabe proved that violations of the structured programming constructs increase cyclomatic complexity, and violations cannot occur singly.

Conclusions For Code-Based Testing

- Excellent supplement (complement?) to specification-based testing because..
  - highlights gaps and redundancies
  - supports a useful range of test coverage metrics
- Test coverage metrics help manage the testing process.
- Tool support is widely available.
- Not much help for identifying test cases.
- Satisfaction of a test coverage metric does not guarantee the absence of faults.

Postscript (for mathematicians only!)

For a set V to be a vector space, two operations (addition and scalar multiplication) must be defined for elements in the set. In addition, the following criteria must hold for all vectors x, y, and z ϵ V, and for all scalars k, l, 0, and 1:

a. if $x, y \in V$, the vector $x + y \in V$.
b. $x + y = y + x$.
c. $(x + y) + z = x + (y + z)$.
d. there is a vector $0 \in V$ such that $x + 0 = x$.
e. for any $x \in V$, there is a vector $-x \in V$ such that $x + (-x) = 0$.
f. for any $x \in V$, the vector $kx \in V$, where k is a scalar constant.
g. $k(x + y) = kx + ky$.
h. $(k + l)x = kx + lx$.
i. $k(lx) = (kl)x$.
j. $1x = x$.