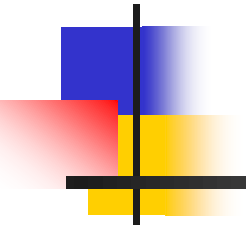
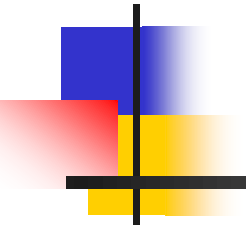


Software Testing & Quality Assurance

Structural Testing

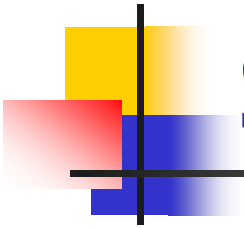
- Path Testing
- Test Coverage Metrics
- Cyclomatic and Essential Complexity
- Guidelines and Observations



Credits & Readings

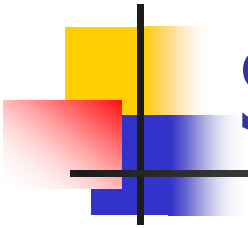
The material included in these slides are mostly adopted from the following books:

- *Software Testing: A Craftsman's Approach*, by Paul Jorgensen, CRC PRESS, third edition, ISBN: 0-8493-7475-8
- Cem Kaner, Jack Falk, Hung Q. Nguyen, *"Testing Computer Software"* Wiley (see also http://www.testingeducation.org/)
- Paul Ammann and Jeff Offutt, *"Introduction to Software Testing"*, Cambridge University Press
- Beizer, Boris, "*Software Testing and Quality Assurance"*, Van Nostrand Reinhold
- Glen Myers, *"The Art of Software Testing"*
- Stephen R. Schach, "*Software Engineering",* Richard D. Irwin, Inc. and Aksen Associates, Inc.



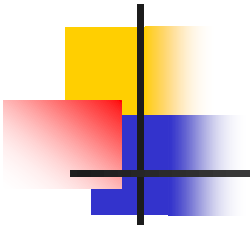
Structural Testing

- Also known as glass/white/open box testing
- A software testing technique whereby explicit knowledge of the internal workings of the item being tested are used to select the test data
- Functional vs. Structural Testing
  - Functional Testing uses program specification
  - Structural Testing is based on specific knowledge of the source code to define the test cases and to examine outputs



Structural Testing

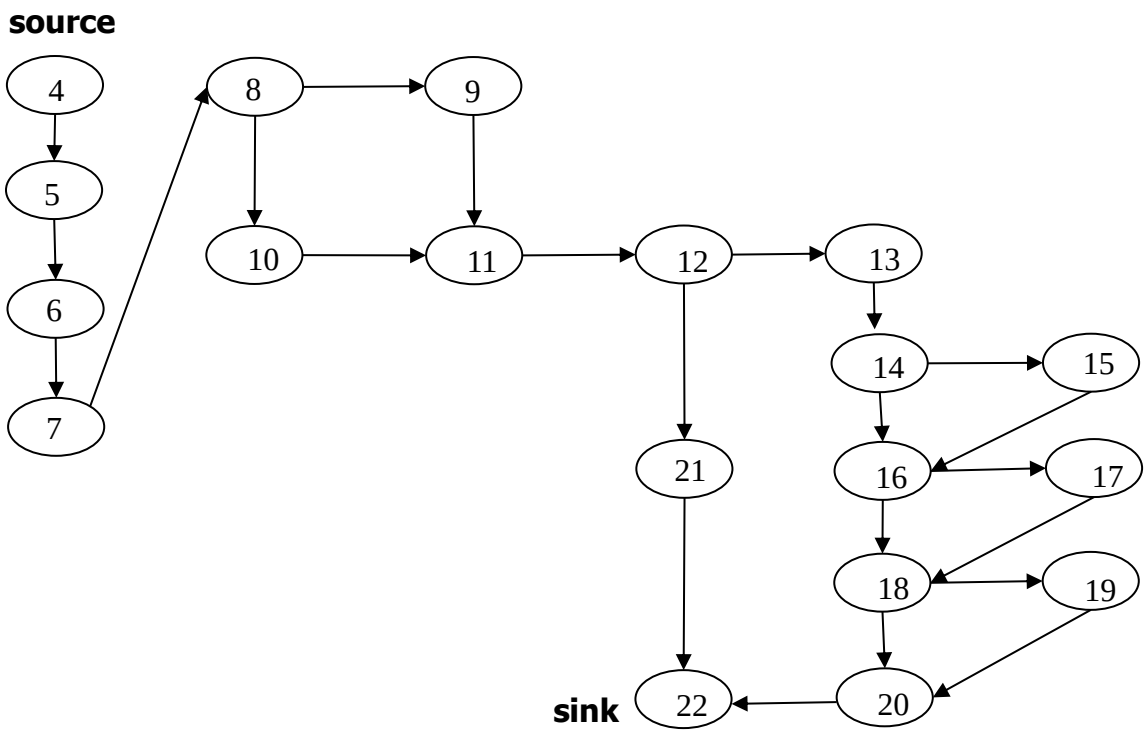
- Structural testing methods are very amenable to:
  - Rigorous definitions
    - Control flow, data flow, coverage criteria
  - Mathematical analysis
    - Graphs, path analysis
  - Precise measurement
    - Metrics, coverage analysis



Path Testing

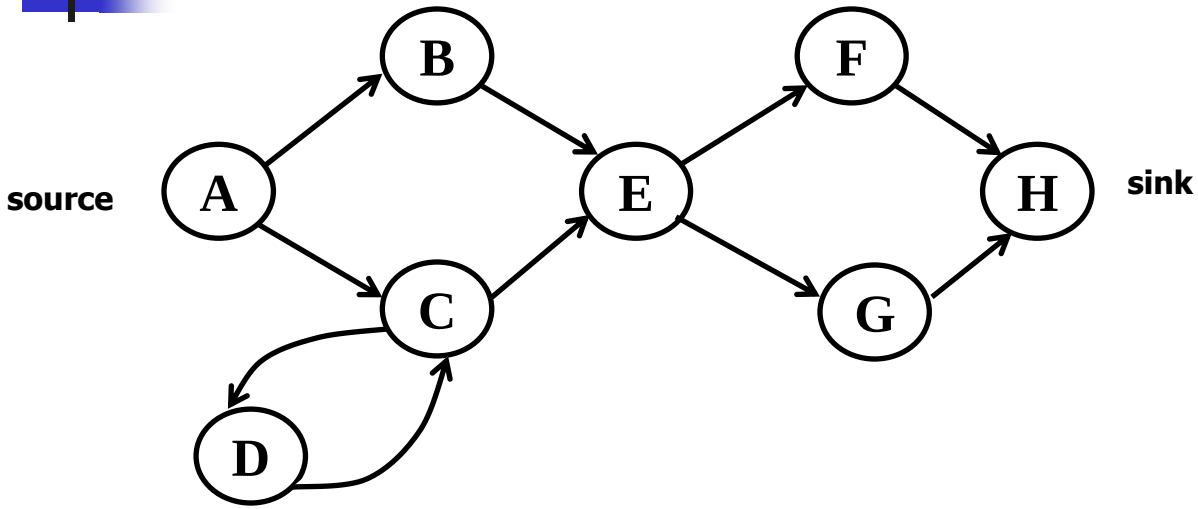
- When a test case executes, it traverses a *path*
  - Some paths can be executed by many test cases
  - Some paths cannot be executed by any test cases
- Paths are derived from some graph construct (e.g. program graph)
- Issues:
  - huge number of paths; some simplification is needed
  - impossible paths
- Challenge: what kinds of faults are associated with what kinds of paths?

Program Graph



- Testers use graphs to model software and then seek ways to provide "coverage"
- A *program graph* (a.k.a. control flow graph) is a directed graph in which
  - A node corresponds to entire statements or fragments of a statement
  - An edge represents flow of control (i.e. there is an edge from node i to node j iff the statement or statement fragment corresponding to node j can be executed immediately after the statement or statement fragment corresponding to node i)

Program Graph Coverage



Path Coverage:

Cover every path

- **ABEFH**
- **ABEGH**
- **ACEGH**
- **ACEFH**
- **ACDCEFH**
- **ACDCEGH**
- **A(CD)*EGH**

Statement/Segment coverage:

Cover every statement/segment (node)

- **ABEFH**
- **ACDCEGH**

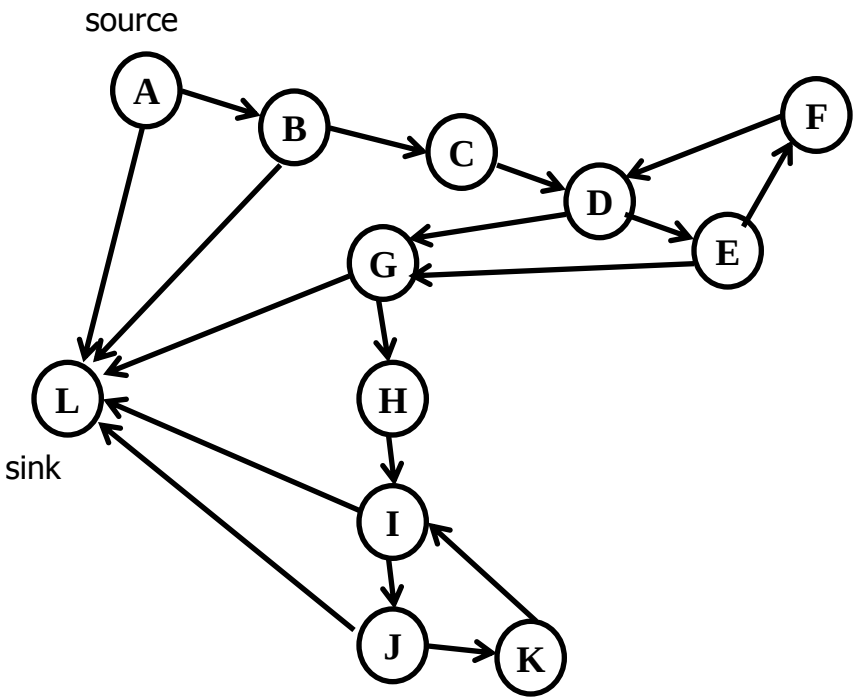
Branch coverage:

Cover every branch (edge)

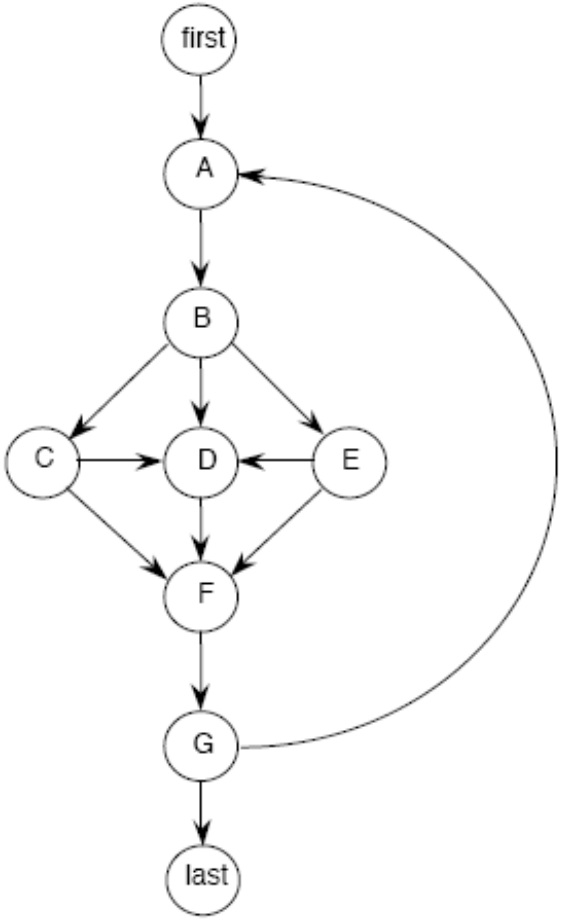
- **ABEFH**
- **ACDCEGH**

Path Expressions

AL
ABL
ABCDGL
ABCDEGL
ABC(DEF)*DGL
ABC(DEF)*DEGL
ABCDGHIL
ABCDGHIJL
ABCDGH(IJK)*IL
ABC(DEF)*DEGH(IJK)*IJL
…



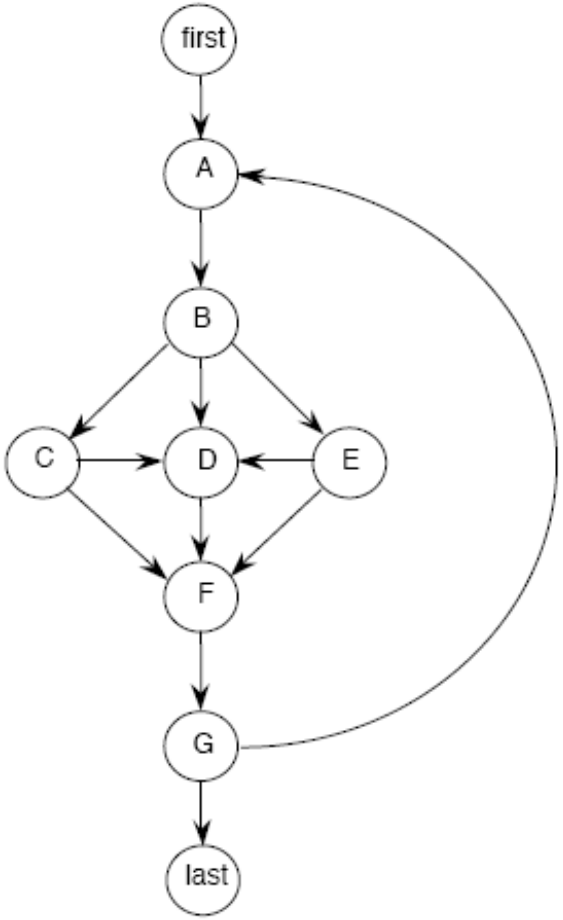
Enormous Number of Paths



- If the loop executes up to 18 times, there are more than 4 trillion paths ($5^1+5^2+5^3+\ldots+5^{18}$)
- BUT
  - Who could ever test all of these?
  - We need an elegant, mathematically sensible alternative

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

Test Cases for Coverage



Some Test Cases

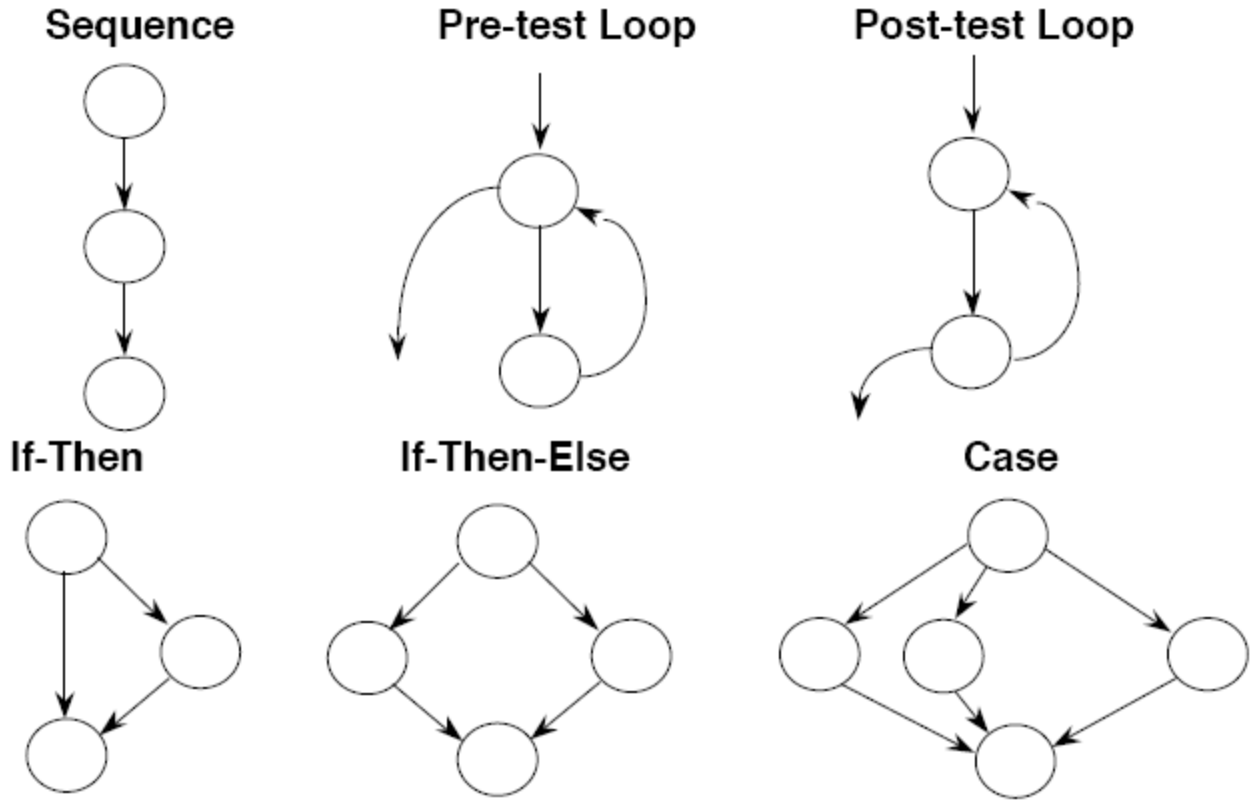
1. First-A-B-C-F-G-Last
2. First-A-B-C-D-F-G-Last
3. First-A-B-D-F-G-A-B-D-F-G-Last
4. First-A-B-E-F-G-Last
5. First-A-B-E-D-F-G-Last

These test cases cover

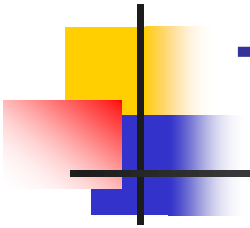
- Every node
- Every edge
- Normal repeat of the loop
- Exiting the loop

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

Program Graphs of Structured Programming Constructs



Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.

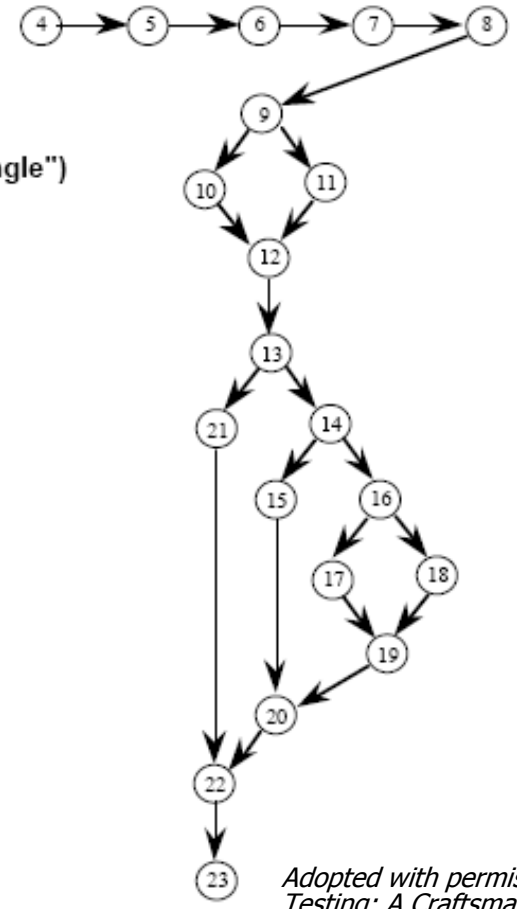


Triangle Program Specification

- Inputs: a, b, and c are non-negative integers, taken to be sides of a triangle
- Output: type of triangle formed by a, b, and c
  - Not a triangle
  - Scalene (no equal sides)
  - Isosceles (exactly 2 sides equal)
  - Equilateral (3 sides equal)
- To be a triangle, a, b, and c must satisfy all three triangle inequalities:
  - $a < b + c$
  - $b < a + c$
  - $c < a + b$

Program Graph for the Triangle Program

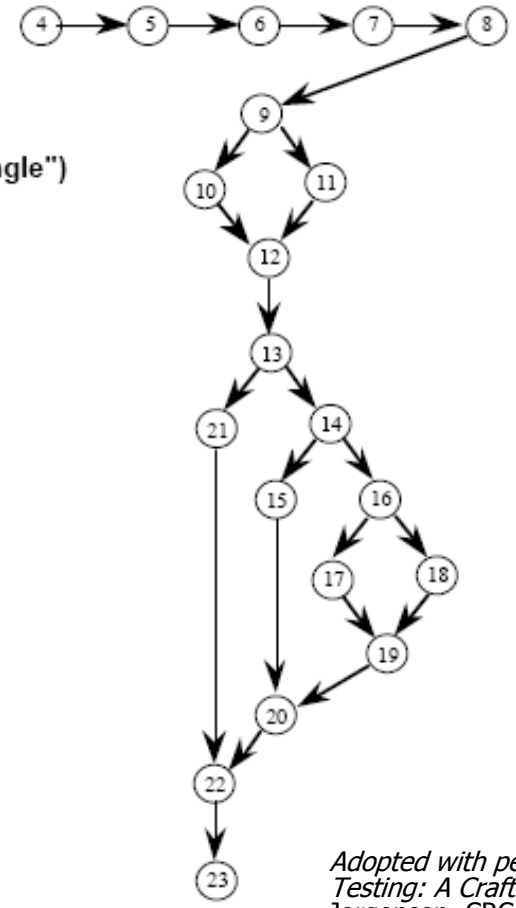
```
1.  Program triangle
2.  Dim a,b,c As Integer
3.  Dim IsATriangle As Boolean
    'Step 1:  Get Input
4.  Output("Enter 3 integers which are sides of a triangle")
5.  Input(a,b,c)
6.  Output("Side A is ",a)
7.  Output("Side B is ",b)
8.  Output("Side C is ",c)
    'Step 2:  Is A Triangle?
9.  If (a < b + c) AND (b < a + c) AND (c < a + b)
10.    Then IsATriangle = True
11.    Else IsATriangle = False
12  EndIf
    'Step 3:  Determine Triangle Type
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
```



Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.

Trace Code for a = 5, b = 5, c = 5

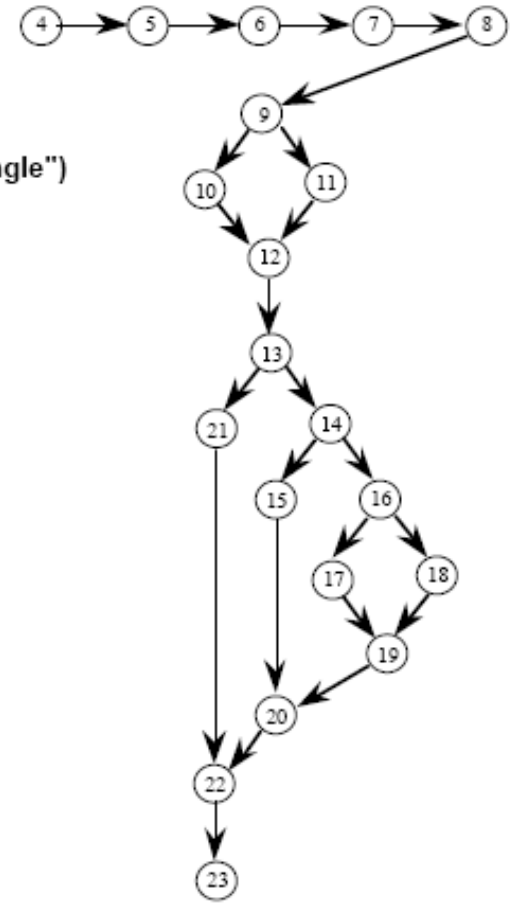
```
1.  Program triangle
2.  Dim a,b,c As Integer
3.  Dim IsATriangle As Boolean
    'Step 1:  Get Input
4.  Output("Enter 3 integers which are sides of a triangle")
5.  Input(a,b,c)
6.  Output("Side A is ",a)
7.  Output("Side B is ",b)
8.  Output("Side C is ",c)
    'Step 2:  Is A Triangle?
9.  If (a < b + c) AND (b < a + c) AND (c < a + b)
10.    Then IsATriangle = True
11.    Else IsATriangle = False
12  EndIf
    'Step 3:  Determine Triangle Type
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
```



Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.

In Class Activity

```
1.  Program triangle
2.  Dim a,b,c As Integer
3.  Dim IsATriangle As Boolean
    'Step 1:  Get Input
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)
    'Step 2:  Is A Triangle?
9.  If (a < b + c) AND (b < a + c) AND (c < a + b)
10.    Then IsATriangle = True
11.    Else IsATriangle = False
12  EndIf
    'Step 3:  Determine Triangle Type
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
```



Trace the code for following test cases:

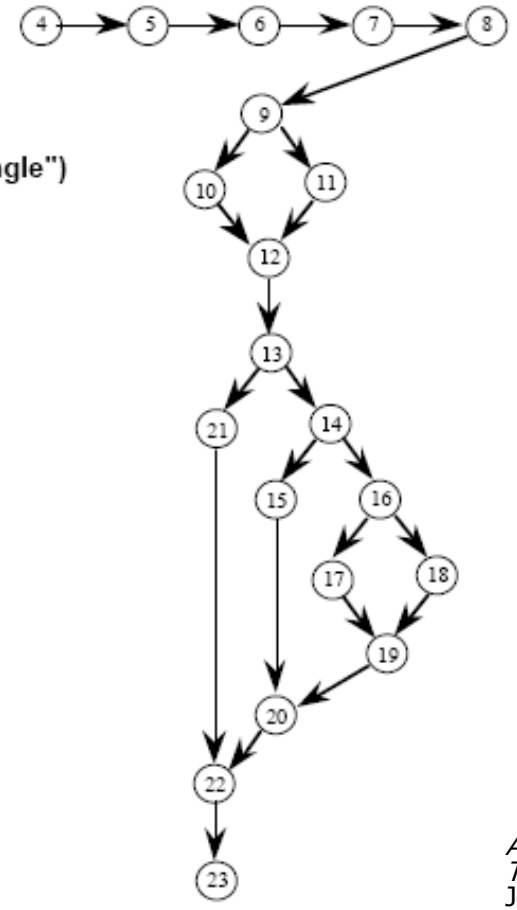
T1: a = 2, b = 5, c = 5
T2: a = 3, b = 4, c = 5
T3: a = 2, b = 3, c = 7

Show the path coverage on the program graph

Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.

Impossible Paths

```
1.  Program triangle
2.  Dim a,b,c As Integer
3.  Dim IsATriangle As Boolean
    'Step 1:  Get Input
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)
    'Step 2:  Is A Triangle?
9.  If (a < b + c) AND (b < a + c) AND (c < a + b)
10.    Then IsATriangle = True
11.    Else IsATriangle = False
12  EndIf
    'Step 3:  Determine Triangle Type
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
```



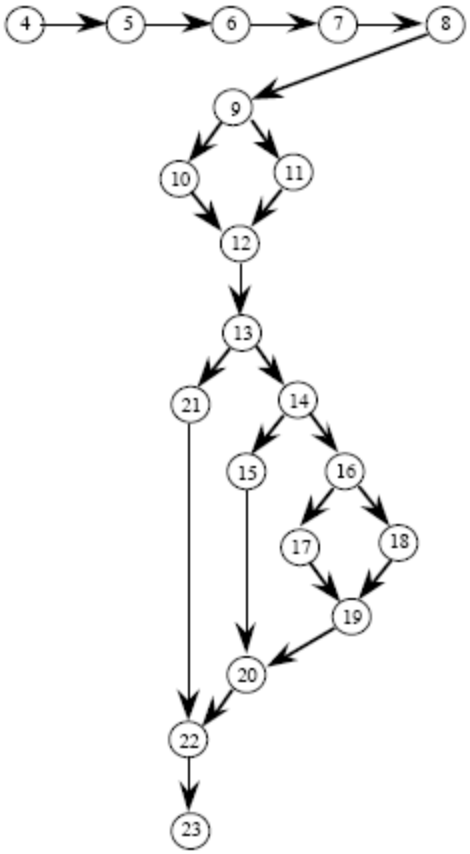
Can you find values of a, b, and c such that the path traverses nodes 10 and 21?

Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.

DD-Path definition

- A decision-to-decision path (DD-Path) is a chain in a program graph that satisfies any of the following cases:
  - Case 1: it consists of a single node with indegree=0 (E.g. node 4) or
  - Case 2: it consists of a single node with outdegree=0 (E.g. node 23) or
  - Case 3: it consists of a single node with indegree ≥ 2 or outdegree ≥ 2 (E.g. node 9, 12, 13, 14, 16, 19, 20, 22) or
  - Case 4: it consists of a single node with indegree =1, and outdegree = 1, or (E.g. nodes 10, 11, 15, 17, 18, 21) or
  - Case 5: it is a maximal chain of length ≥ 1 (E.g. nodes 5-8)

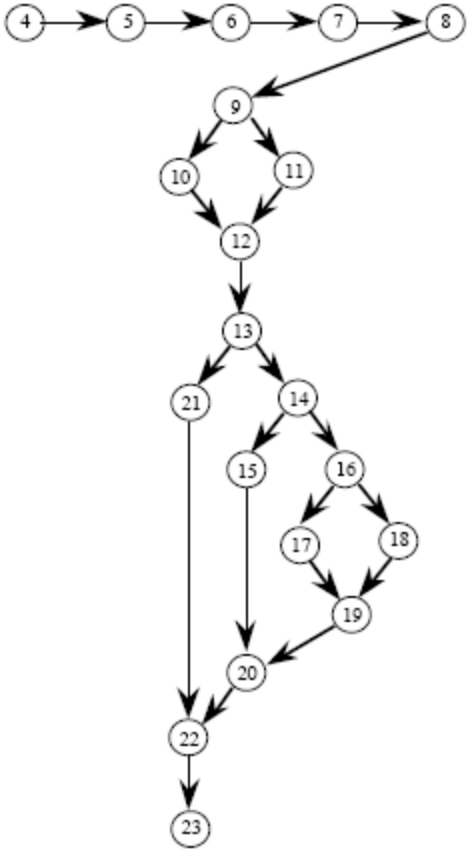
Program Graph for Triangle



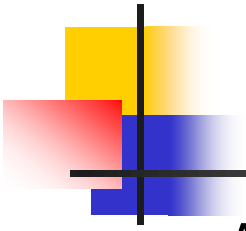
DD-Paths for the Triangle

Program Graph Node	DD-path Name	Definition Case
4	First	1
5-8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3
21	G	4
22	O	3
23	Last	2

Program Graph for Triangle



Note: since the Triangle program is logic intensive and computationally sparse, it yields many short DD-Paths as shown in the this table. In other programs we will find longer chains.

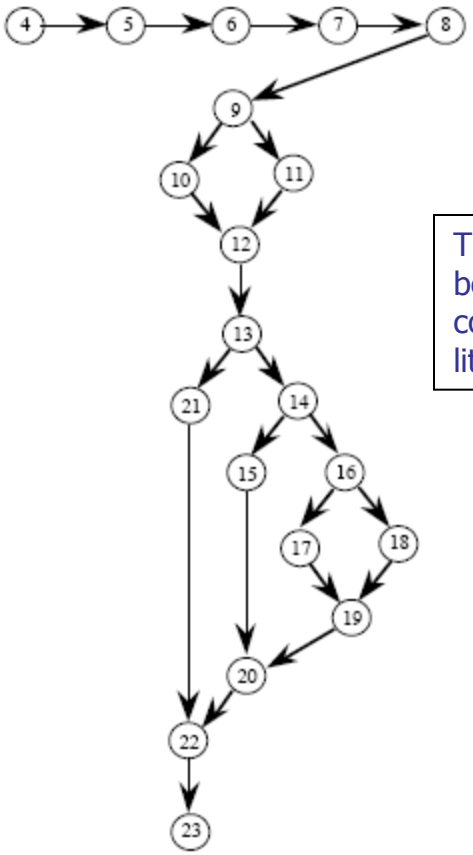


DD-Path Graph

- A program's *DD-Path graph* is a directed graph in which
  - nodes are DD-Paths of its program graph, and
  - edges represent control flow between successor DD-Paths
- Also known as *Control Flow Graph*
  - it is a form of condensation graph
  - maximal chains are collapsed into an individual node (corresponding to Case 5)
  - single node DD-Paths (corresponding to Cases 1 - 4) preserve the convention that a statement (or a statement fragment) is in exactly one DD-Path
  - The process of generating DD-path graphs manually can be a tedious task (for larger programs)
  - There exist commercial tools that automate this process

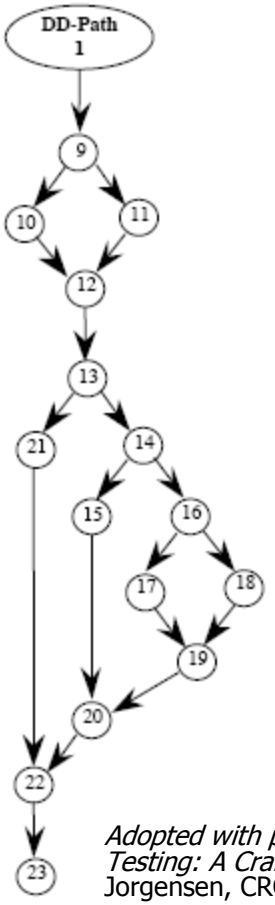
DD-Path Graph for Triangle

Program Graph



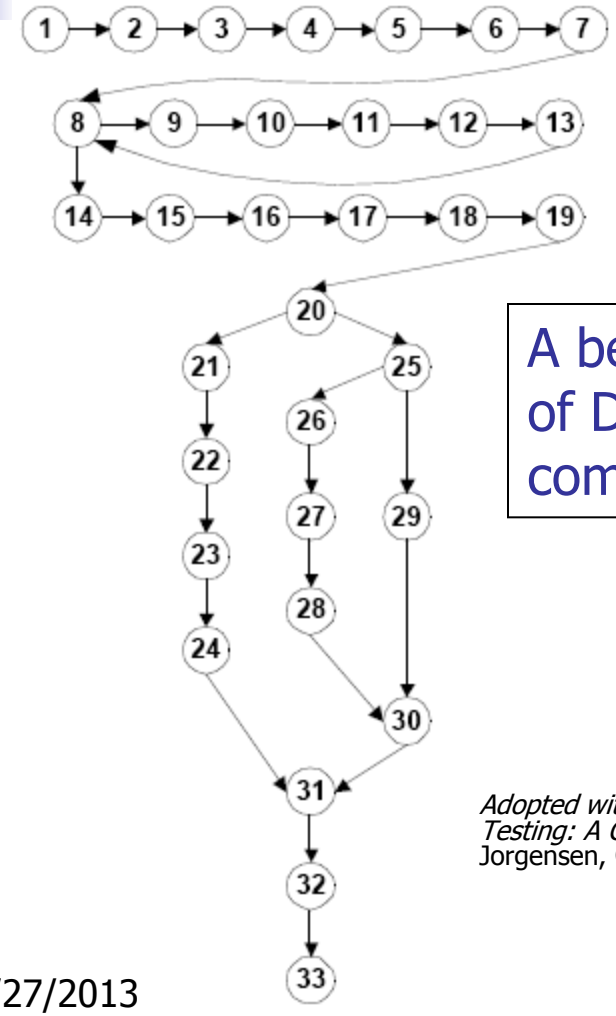
There is not much compression here because this example is control intensive, with little sequential code.

DD-Path Graph

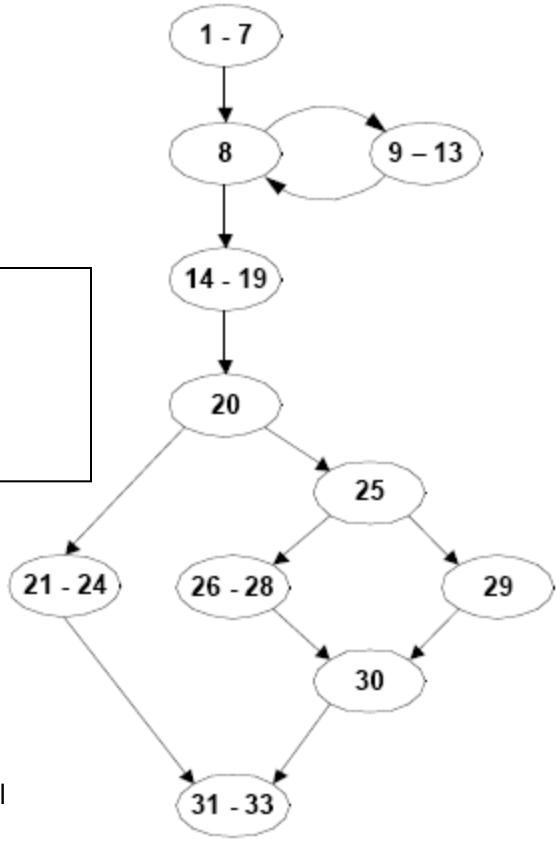


Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.

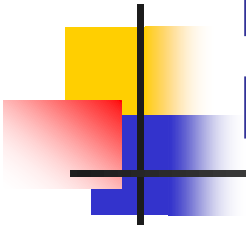
DD-Path Graph for the Commission Problem



A better example of DD-Path compression



Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.



Program Graphs with Statement Fragments

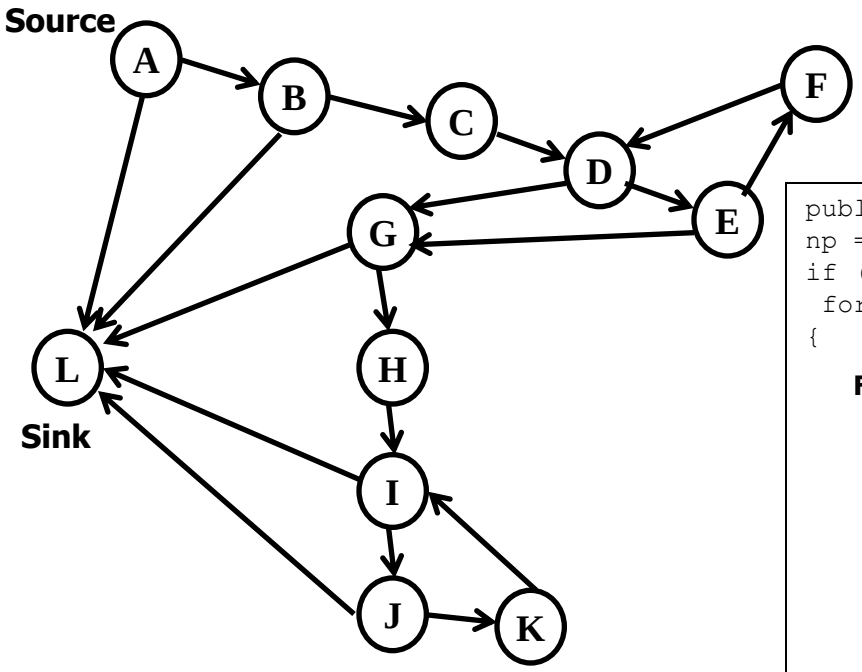
- Create graph nodes when you encounter any of the following
  - a predicate
  - a loop control
  - a break
  - a method exit/return
- Let's try an example…

```
public int displayLastMsg(int nToPrint) {
  np = 0;
  if ((msgCounter > 0) && (nToPrint > 0)) {
    for (int j = lastMsg; (( j != 0) && (np < nToPrint)); --j) {
      System.out.println(messageBuffer[j]);
      ++np;
    }
    if (np < nToPrint) {
      for (int j = SIZE; ((j != 0) && (np < nToPrint)); --j) {
        System.out.println(messageBuffer[j]);
        ++np;
      }
    }
  }
  return np;
}
```

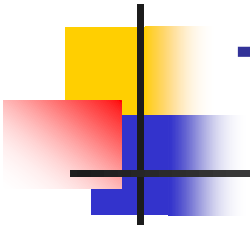
Stop when you encounter any of the following

- a predicate
- a loop control/counter
- a break
- a method exit/return

Program Graph




```
public int displayLastMsg(int nToPrint) {
np = 0;          A                    B
if ((msgCounter > 0) && (nToPrint > 0)) {
 for (int j = lastMsg; (( j != 0) && (np < nToPrint)); --j)
{             C                  D              E
    F   System.out.println(messageBuffer[j]);
        ++np;
     }          G
     if (np < nToPrint) {
       for (int j = SIZE; ((j != 0) && (np < nToPrint)); --j) {
                   H             I              J
      K  System.out.println(messageBuffer[j]);
         ++np;
       }
     }
   }
   return np;    L
}
```



Test Coverage Metrics

Graph-based

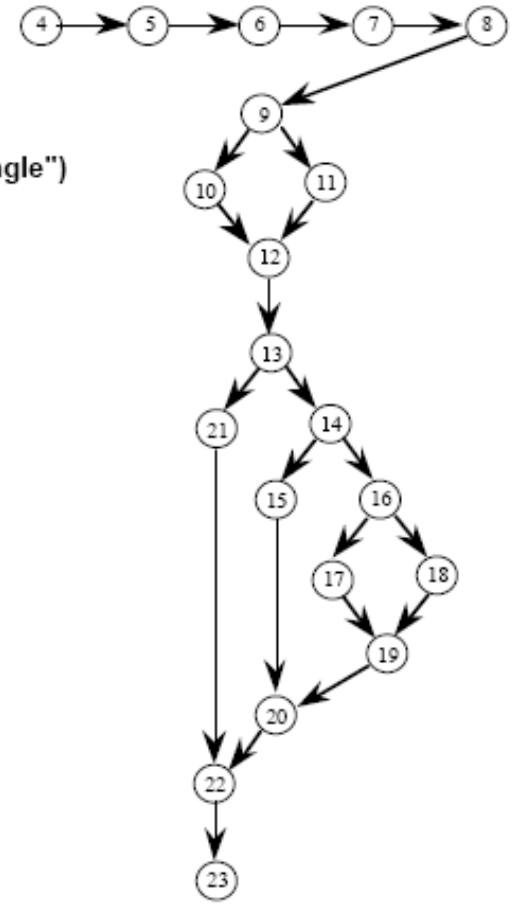
- Every node
- Every edge
- Every path

Code-based (E. F. Miller, 1977 dissertation)

C_0: Every statement is executed at least once
C_1: Every DD-Path is executed at least once
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

Dependent DD-Paths

```
1.  Program triangle
2.  Dim a,b,c As Integer
3.  Dim IsATriangle As Boolean
    'Step 1:  Get Input
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)
    'Step 2:  Is A Triangle?
9.  If (a < b + c) AND (b < a + c) AND (c < a + b)
10.    Then IsATriangle = True
11.    Else IsATriangle = False
12  EndIf
    'Step 3:  Determine Triangle Type
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
```



Often correspond to infeasible paths

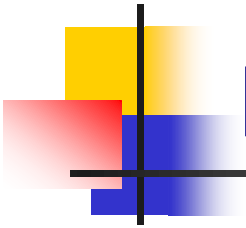
If a path traverses node 10 (Then IsATriangle =True), then it must traverse node 14

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

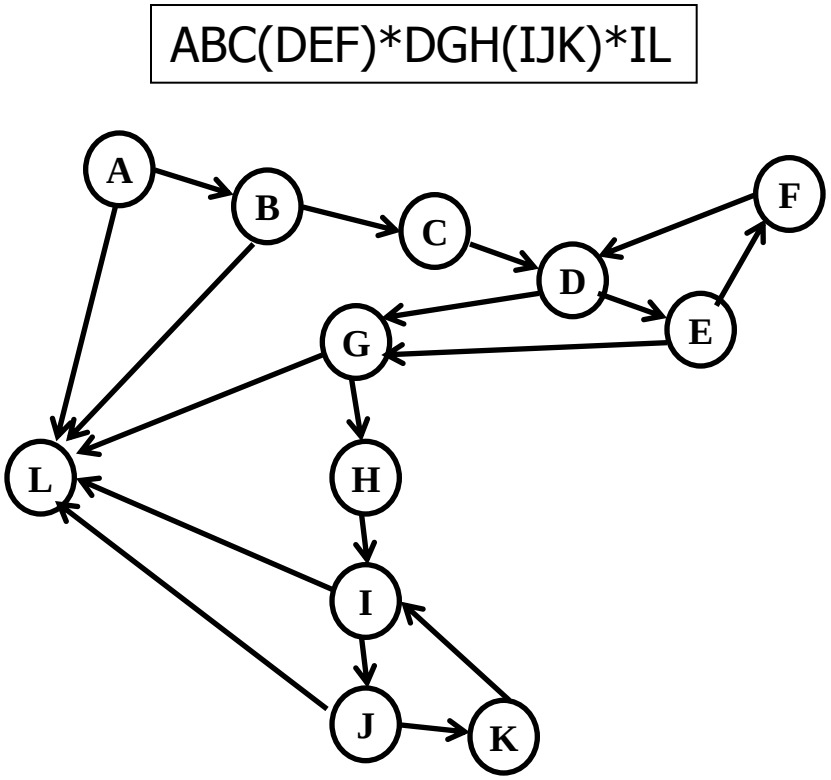


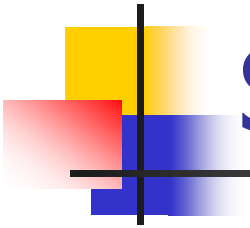
Popular Coverage Metrics

- Statement Coverage
  - Every statement is executed at least once
- DD-Path Coverage
  - Every DD-Path is executed at least once
- Branch Coverage
  - Every predicate outcome (T/F) is executed at least once
- Multiple-Condition Coverage
  - All T/F combinations of simple conditions in compound predicates are considered at least once

Statement Coverage

- Achieved when all statements in a method have been executed at least once
- A test case that will follow the path expression shown above will achieve statement coverage in this example
- One test case is enough to achieve statement coverage!

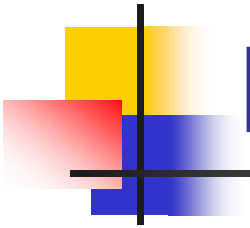




Statement Coverage Issues

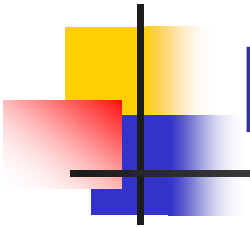
- A predicate may be tested for only one value (can miss many bugs)
- Loop bodies may only be iterated once
- Statement coverage can be achieved without branch coverage. Important cases may be missed

```
String s = null;
if (x != y) s = “Hi”;
String s2 = s.substring(1);
```



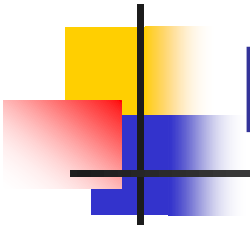
DD-Path Coverage

- It covers segments not just single statements
- May produce drastically different numbers
  - Assume two segments P and Q
  - P has one statement, Q has nine
  - Exercising only one of the segments will give 10% or 90% statement coverage
  - Segment coverage will be 50% in both cases



Branch Coverage

- At least one True and one False evaluation for each predicate
- Short-circuit evaluation means that many predicates might not be evaluated
- A compound predicate is treated as a single statement
  - If n clauses, 2^n combinations, but only 2 are tested



Multiple-Condition Coverage

- All True-False combinations of simple conditions in compound predicates are considered at least once
- A truth table may be necessary and can be constructed as follows:
  - Consider the multiple condition as a 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)

Apply Multiple-Condition Coverage to the Following Code

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

Truth Table for (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

Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.

Decision Table for (a<b+c) AND (b<a+c) AND (c<a+b)

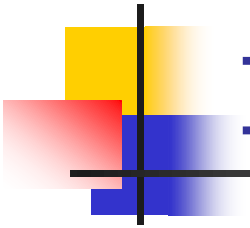
c1: a<b+c	T	T	T	T	F	F	F	F
c2: b<a+c	T	T	F	F	T	T	F	F
c3: c<a+b	T	F	T	F	T	F	T	F
a1: impossible				X		X	X	X
a2:Valid test case #	1	2	3		4			

Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.

Multiple Condition Test Cases for (a<b+c) AND (b<a+c) AND (c<a+b)

Test Case	True Conditions	a	b	c	Expected
1	All three conditions True	3	4	5	TRUE
2	a < b + c b < a + c	3	4	9	FALSE
3	a < b + c c < a + b	3	9	4	FALSE
4	b < a + c c < a + b	9	3	4	FALSE

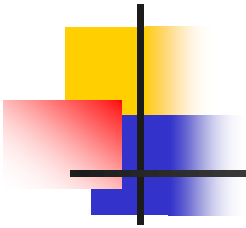
Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.



In Class Activity

What test cases are needed for this code fragment ?

```
If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
  Then Output (“Scalene”)
  Else Output (“Isosceles”)
Endif
```

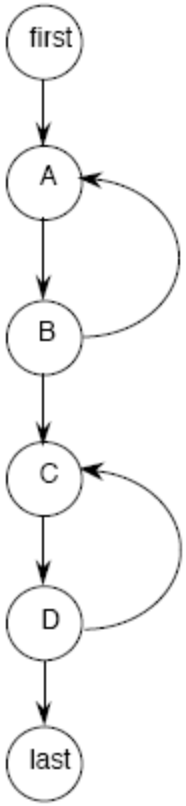


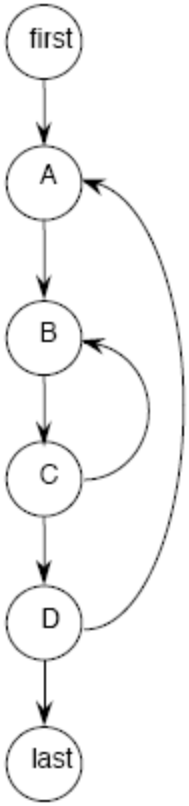
Dealing with Loops

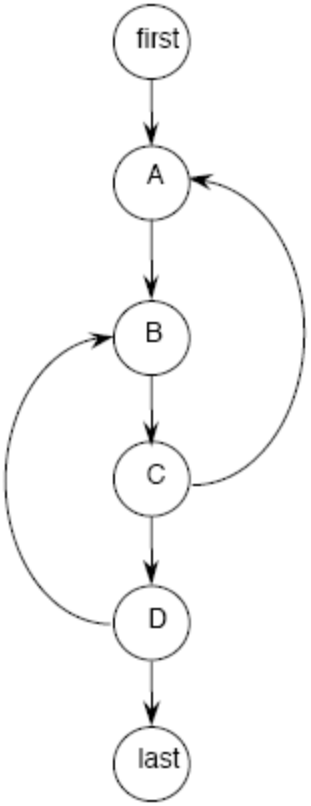
- Loops are highly fault-prone, so they need to be tested carefully
- A simple view: Every loop involves a decision to traverse the loop or not
- A better view: Boundary value analysis on the index variable
- Huang's Theorem: (Paraphrased)

  Everything interesting will happen in two loop traversals: the normal loop traversal and the exit from the loop

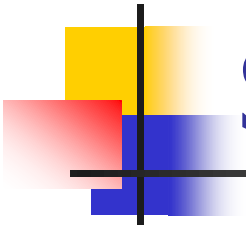
Concatenated, Nested, and Knotted Loops





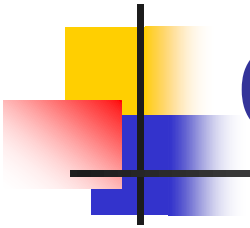


Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.



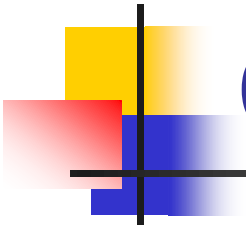
Strategy for Loop Testing

- Huang's theorem suggests/assures 2 tests per loop is sufficient
- For nested loops:
  - Test innermost loop first
  - Then "condense" the loop into a single node (as in condensation graph)
  - Work from innermost to outermost loop
- For concatenated loops: use Huang's Theorem
- For knotted loops: Rewrite! (violates structure programming principles and results in high cyclomatic complexity)



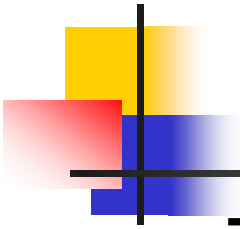
Code-Based Testing Strategy

- 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 (e.g. statement, branch, etc.)
- If appropriate coverage is attained by the initial test cases, fine
- Otherwise, add test cases to attain the intended test coverage



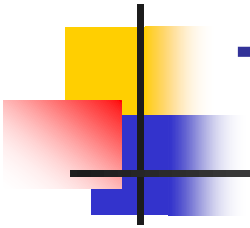
Coverage Usefulness

- 100% coverage is never a guarantee of bug-free software
- Coverage reports can
  - point out inadequate test suites
  - suggest the presence of surprises, such as blind spots (gaps) in the test design
  - Help identify parts of the implementation that require further structural testing



How to Measure Coverage

- The source code is *instrumented*
  - Depending on the code coverage model, code that writes to a trace file is inserted in every branch, statement etc.
- Most commercial tools measure DD-Path and branch 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

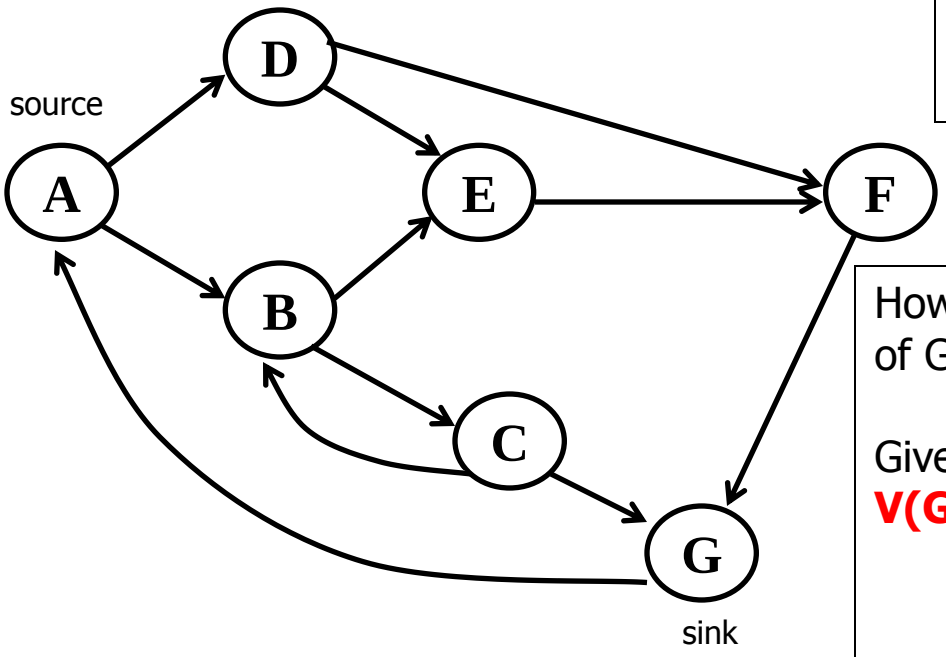
Sample DD-Path Instrumentation

Values of array DDpathTraversed are set to 1 when corresponding instrumented code is executed

```
      DDpathTraversed(1) = 1
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)
      'Step 2: Is A Triangle?
      DDpathTraversed(2) = 1
9. If (a < b + c) AND (b < a + c) AND (c < a + b)
10. Then DDpathTraversed(3) = 1
      IsATriangle = True
11. Else DDpathTraversed(4) = 1
      IsATriangle = False
12 EndIf
```

Cyclomatic Complexity[1]

Consider the following strongly connected directed graph G (node G connects back to node A)



- Cyclomatic complexity is a metric used to measure the complexity of programs
- Programs with high cyclomatic complexity require more testing

How do we calculate the cyclomatic complexity V of G?

Given a strongly connected graph G:

V(G) = e – n + 1 = 11 – 7 + 1 = 5 where

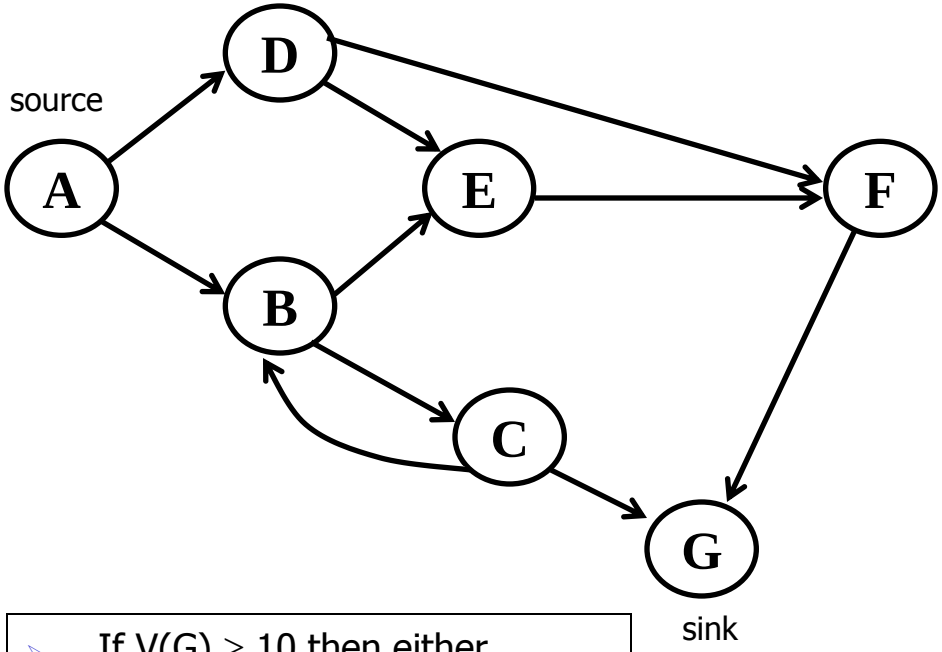
e = number of edges

n = number of nodes

It means that G contains 5 independent circuits

Cyclomatic Complexity[2]

An arbitrary directed graph G(V,E)
(Note: violations of structured programming)



- If V(G) ≥ 10 then either simplify the unit or plan to do more testing

Another way to calculate V(G):

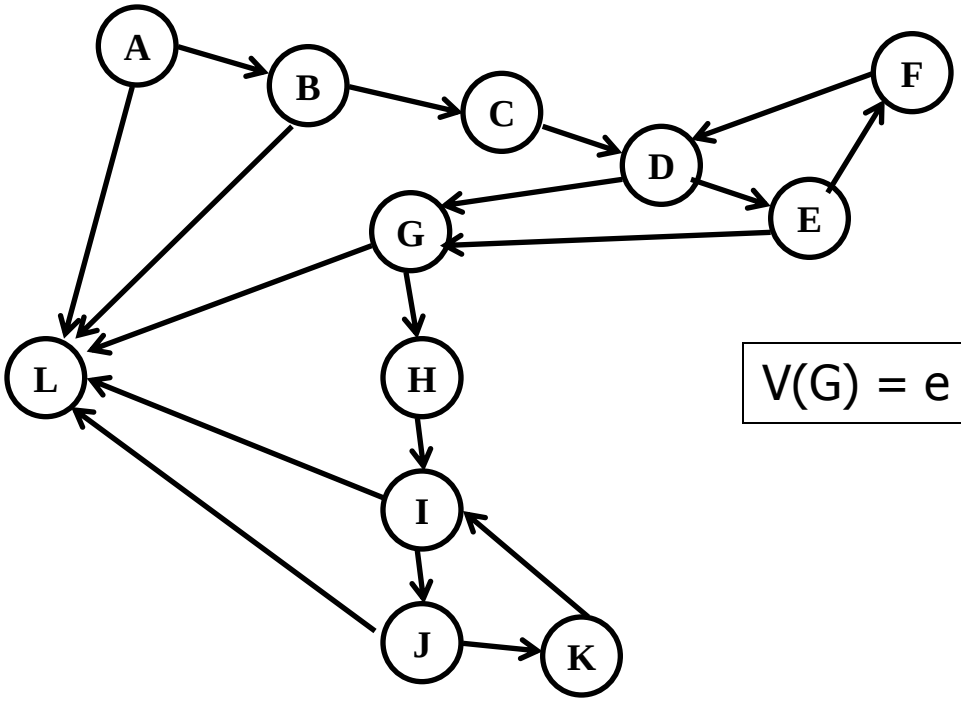
Given an arbitrary directed graph G:
V(G) = e – n + 2p = 10 – 7 + 2(1) = 5

e = number of edges
n = number of nodes
p= number of connected regions (which is normally 1)

It means that G has 5 linearly independent paths from node A to node G as follows:
P1: A, B, C, G
P2: A, B, C, B, C, G
P3: A, B, E, F, G
P4: A, D, E, F, G
P5: A, D, F, G

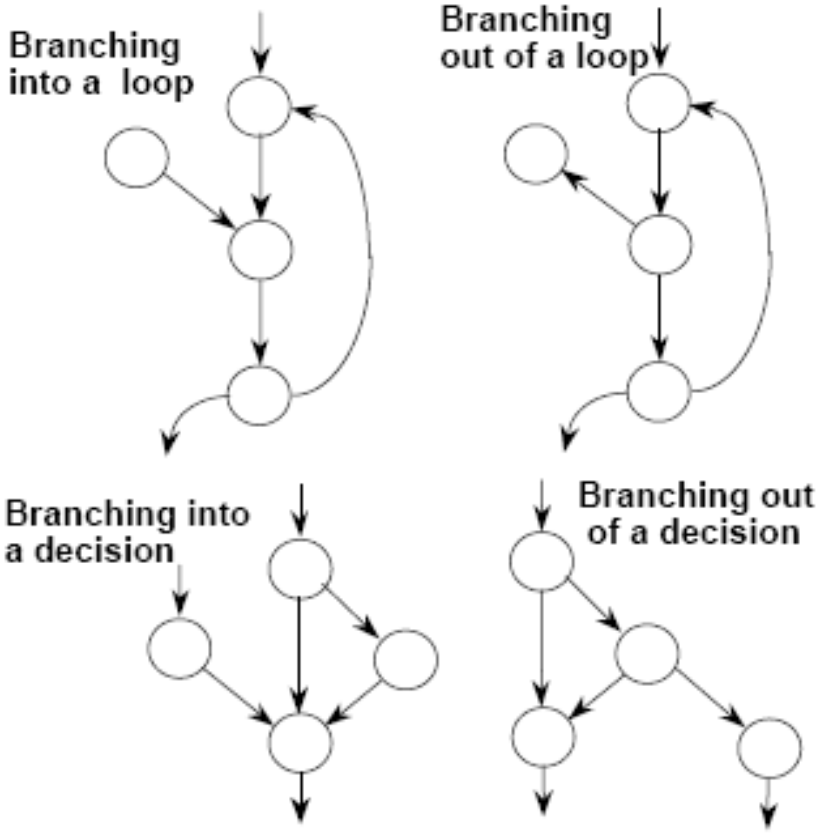
In Class Activity

Can you calculate the Cyclomatic Complexity of this arbitrary directed graph?

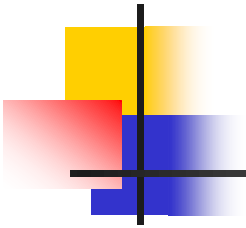


V(G) = e – n + 2p = 18-12+2(1) = 8

Violations of Structured Programming



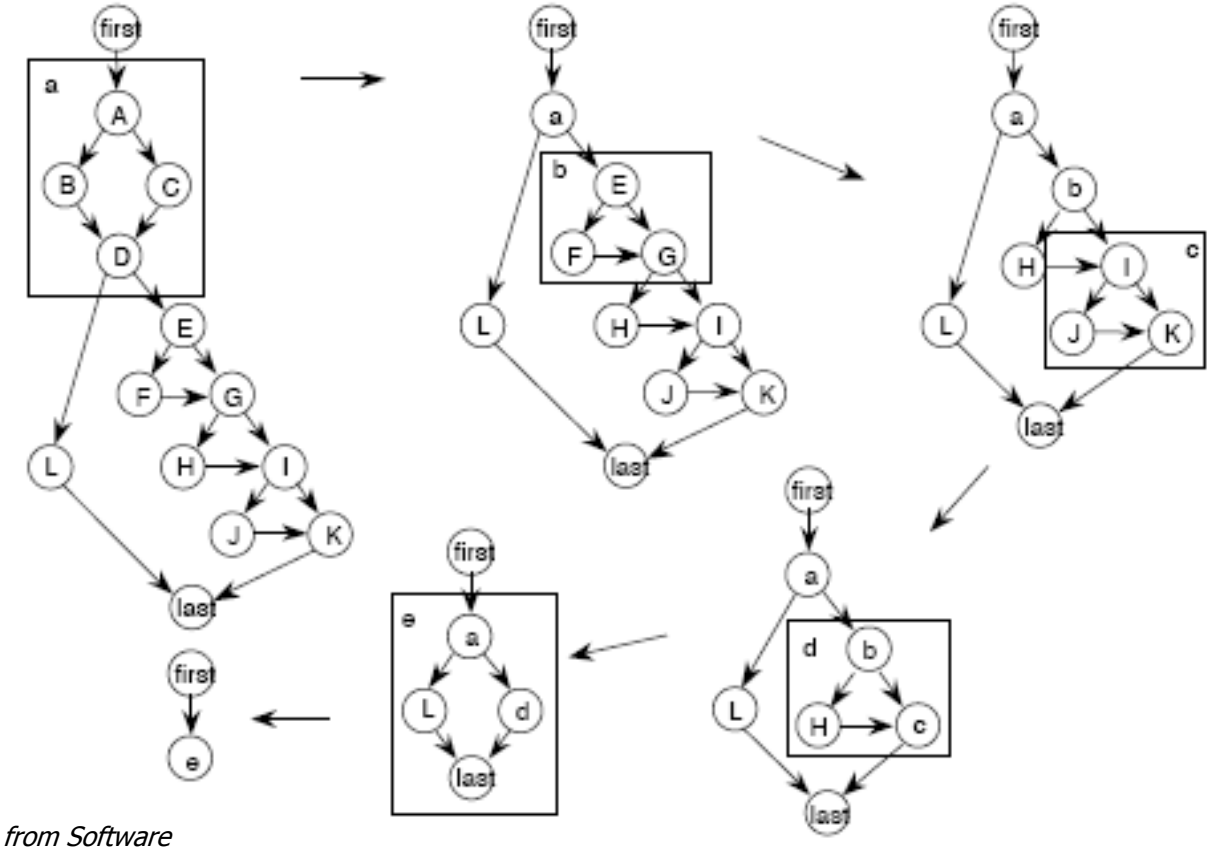
Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.



Essential Complexity

- McCabe's notion of Essential Complexity deals with the extent to which a program violates the principles of Structured Programming
- To find the 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 Complexity of a Structured Program is 1
- Violations of Structured Programming increase the Essential Complexity

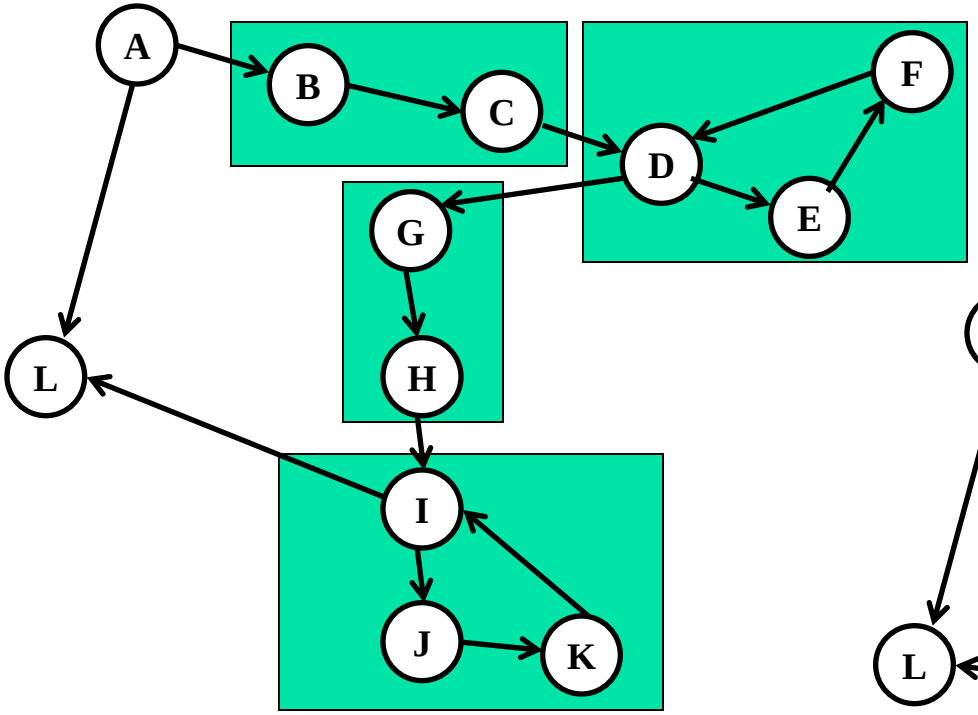
Example



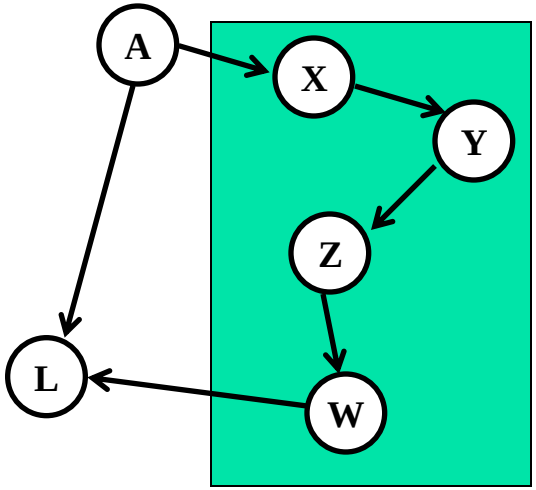
Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.

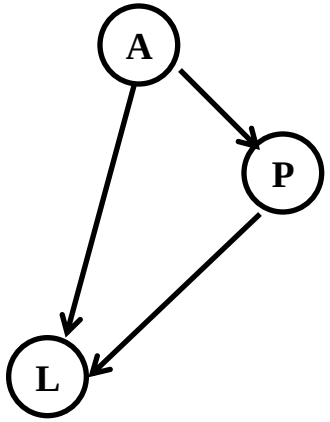
In Class Activity

Can you condense the following directed graph and then calculate its Essential Complexity?



E(G) = e – n + 2p = 3 - 3 + 2(1) = 2



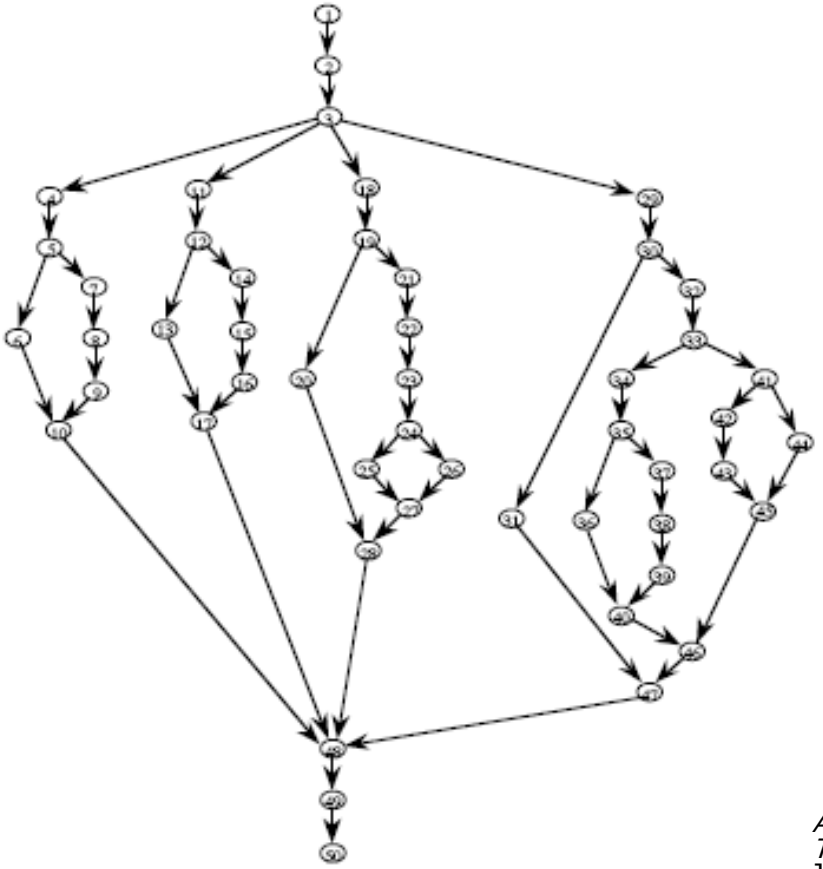


NextDate Pseudo-code

```
Program NextDate
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
1.  Output ("Enter today's date in the form MM DD YYYY")
2.  Input (month,day,year)
3.  Case month Of
4.  Case 1: month Is 1,3,5,7,8,Or 10: '31 day months (except Dec.)
5.     If day < 31
6.           Then tomorrowDay = day + 1
7.           Else
8.                 tomorrowDay = 1
9.                 tomorrowMonth = month + 1
10.    EndIf
11. Case 2: month Is 4,6,9,Or 11 '30 day months
12.    If day < 30
13.          Then tomorrowDay = day + 1
14.          Else
15.                tomorrowDay = 1
16.                tomorrowMonth = month + 1
17.    EndIf
18. Case 3: month Is 12: 'December
19.    If day < 31
20.          Then tomorrowDay = day + 1
21.          Else
22.                tomorrowDay = 1
23.                tomorrowMonth = 1
24.                If year = 2012
25.                      Then Output ("2012 is over")
26.                      Else tomorrow.year = year + 1
27.                EndIf
28.    EndIf
29. Case 4: month is 2: 'February
30.    If day < 28
31.          Then tomorrowDay = day + 1
32.          Else
33.                If day = 28
34.                      Then
35.                            If ((year MOD 4)=0)AND((year MOD 400)≠0)
36.                                  Then tomorrowDay = 29 'leap year
37.                                  Else          'not a leap year
38.                                        tomorrowDay = 1
39.                                        tomorrowMonth = 3
40.                            EndIf
41.                      Else  If day = 29
42.                                        Then tomorrowDay = 1
43.                                              tomorrowMonth = 3
44.                                        Else  Output("Cannot have Feb.", day)
45.                                  EndIf
46.                EndIf
47.    EndIf
48. EndCase
49. Output ("Tomorrow's date is", tomorrowMonth,
                      tomorrowDay, tomorrowYear)
50. End NextDate
```

Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.

NextDate Program Graph

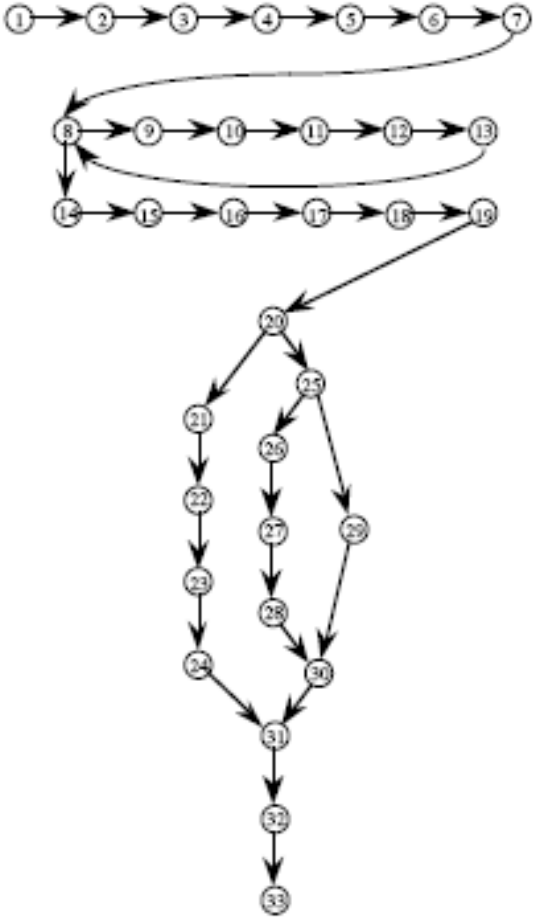


Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.

Commission Pseudo-code

```
Program Commission
Dim lockPrice, stockPrice, barrelPrice As Real
Dim locks, stocks, barrels As Integer
Dim totalLocks, totalStocks As Integer
Dim totalBarrels As Integer
Dim lockSales, stockSales As Real
Dim barrelSales As Real
Dim sales, commission As Real
1   lockPrice = 45.0
2   stockPrice = 30.0
3   barrelPrice = 25.0
4   totalLocks = 0
5   totalStocks = 0
6   totalBarrels = 0
7   Input(locks)
8   While NOT(locks = -1)
9     Input(stocks, barrels)
10    totalLocks = totalLocks + locks
11    totalStocks = totalStocks + stocks
12    totalBarrels = totalBarrels + barrels
13    Input(locks)
14  EndWhile
15  Output("Locks sold: ", totalLocks)
16  Output("Stocks sold: ", totalStocks)
17  Output("Barrels sold: ", totalBarrels)
18  sales = lockPrice*totalLocks +
    stockPrice*totalStocks  + barrelPrice * totalBarrels
19  Output("Total sales: ", sales)
20  If (sales > 1800.0)
21    Then
22      commission = 0.10 * 1000.0
23      commission = commission + 0.15 * 800.0
24      commission = commission + 0.20*(sales-1800.0)
25  Else If (sales > 1000.0)
26    Then
27      commission = 0.10 * 1000.0
28      commission = commission + 0.15*(sales-1000.0)
29    Else commission = 0.10 * sales
30    EndIf
31  EndIf
32  Output("Commission is $", commission)
33  End Commission
```

Commission Program Graph



Adopted with permission from Software Testing: A Craftsman's Approach, by Paul Jorgensen, CRC PRESS, third edition.