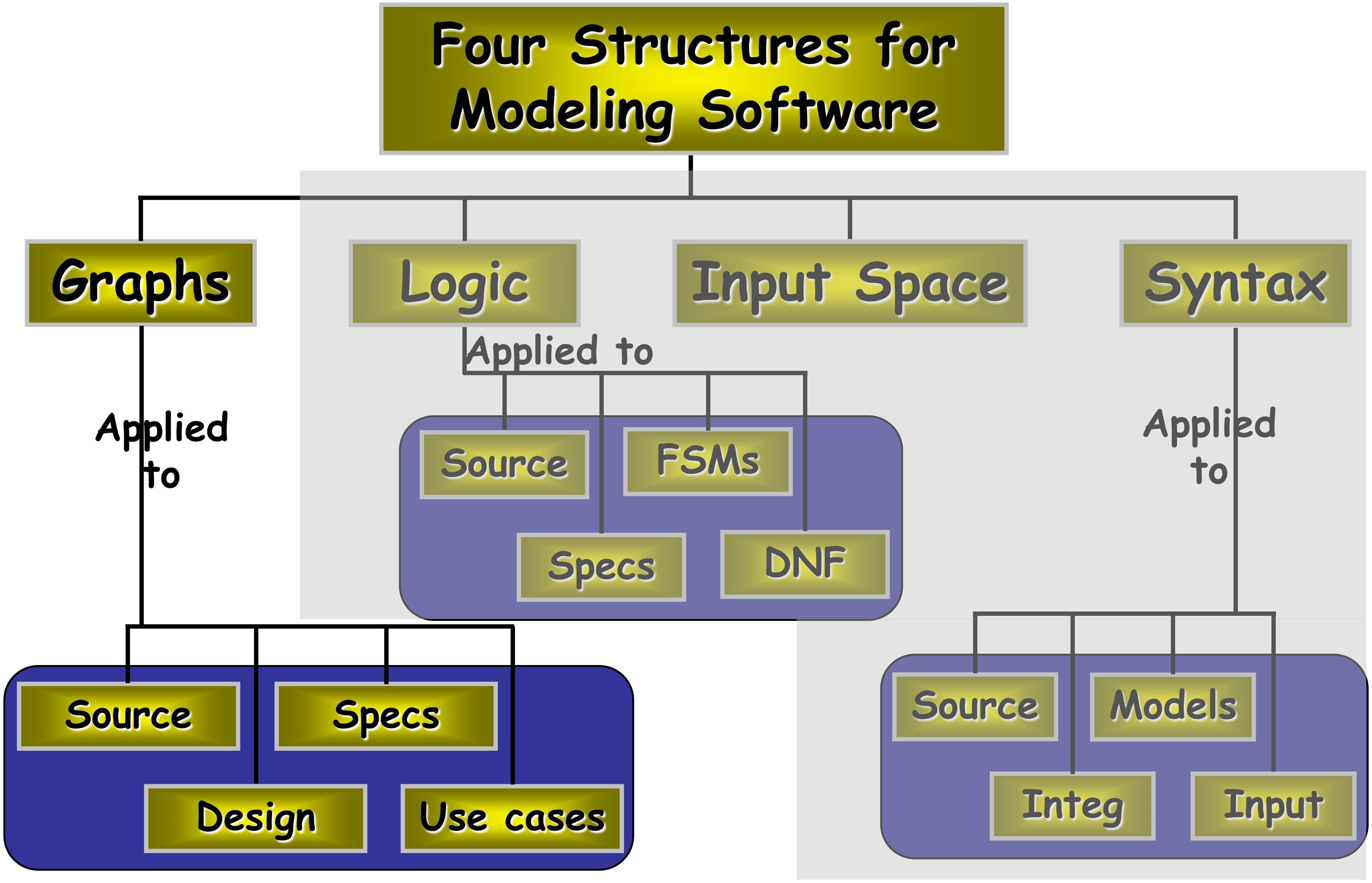
Software Engineering Testing

Graph Coverage

Objective

- Provide an overview of graph coverage
  - ➢ beginning with theoretical aspects of graphs
  - ➢ then looking at practical applications by demonstrating how graphs can be obtained from various software artifacts
- Introduce graph coverage for source code
- Discuss graph coverage for UML Use Cases

Overview - Graph Coverage Criteria



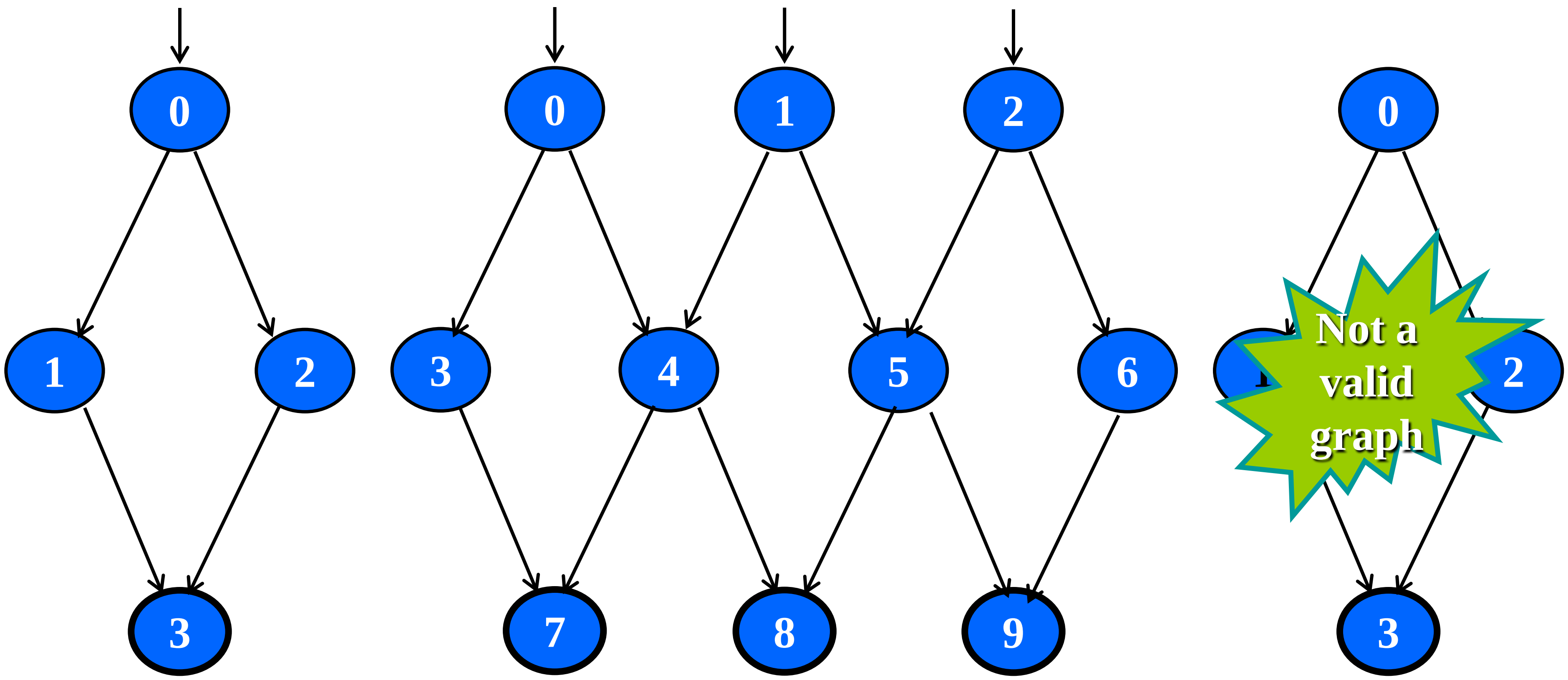
Covering Graphs

- Graphs are the most commonly used structure for testing
- Graphs can come from many sources
  - Control flow graphs
  - Design structure
  - Finite State Machines (FSMs) and statecharts
  - Use cases
- Tests usually are intended to “cover” the graph in some way

Definition of a Graph

- A set N of <u>nodes</u>, N is not empty
- A set N_0 of <u>initial nodes</u>, N_0 is not empty
- A set N_f of <u>final nodes</u>, N_f is not empty
- A set E of <u>edges</u>, each edge from one node to another
  - (n_i , n_j), i is predecessor, j is successor

Three Example Graphs



$N_0 = \{ 0 \}$

$N_f = \{ 3 \}$

$N_0 = \{ 0, 1, 2 \}$

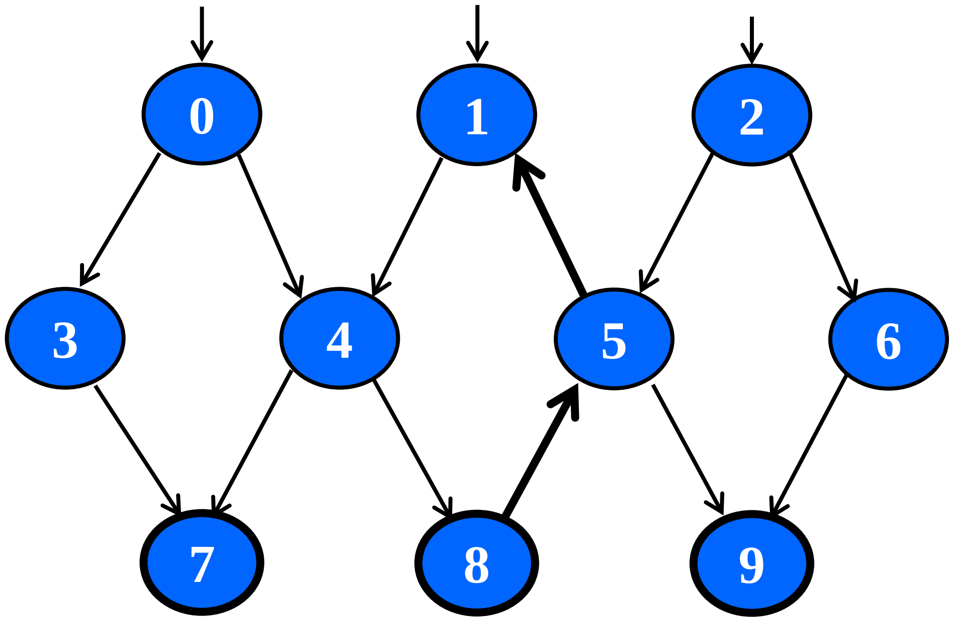
$N_f = \{ 7, 8, 9 \}$

$N_0 = \{ \}$

$N_f = \{ 3 \}$

Paths in Graphs

- Path : A sequence of nodes – $[n_1, n_2, \ldots, n_M]$
  - Each pair of nodes is an edge
- Length : The number of edges
  - A single node is a path of length 0
- Subpath : A subsequence of nodes in *p* is a subpath of *p*
- Reach (*n*) : Subgraph that can be reached from *n*



Paths

[0, 3, 7]

[1, 4, 8, 5, 1]

[2, 6, 9]

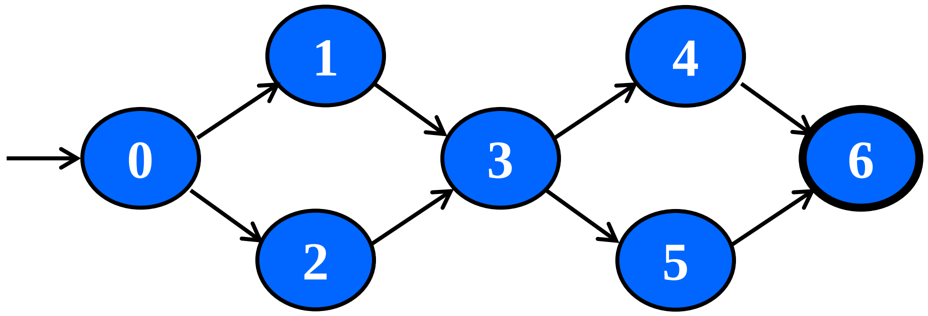
Reach (0) = { 0, 3, 4, 7, 8, 5, 1, 9 }

Reach ({0, 2}) = G

Reach([2,6]) = {2, 6, 9}

Test Paths and SESEs

- Test Path : A path that starts at an initial node and ends at a final node
- Test paths represent execution of test cases
  - Some test paths can be executed by many tests
  - Some test paths cannot be executed by any tests
- SESE graphs : All test paths start at a single node and end at another node
  - Single-entry, single-exit
  - N0 and Nf have exactly one node


$$\begin{tikzcd}
 & 1 \arrow[dr] & & 4 \arrow[dr] & \\
\arrow[r] & 0 \arrow[u] \arrow[d] & 3 \arrow[ur] \arrow[dr] & & 6 \\
 & 2 \arrow[ur] & & 5 \arrow[ur] &
\end{tikzcd}$$

Double-diamond graph

Four test paths

[0, 1, 3, 4, 6]

[0, 1, 3, 5, 6]

[0, 2, 3, 4, 6]

[0, 2, 3, 5, 6]

Visiting and Touring

- Visit : A test path *p* *visits* node *n* if *n* is in *p*

  A test path *p* *visits* edge *e* if *e* is in *p*
- Tour : A test path *p* *tours* subpath *q* if *q* is a subpath of *p*

Path [0, 1, 3, 4, 6]

Visits nodes 0, 1, 3, 4, 6

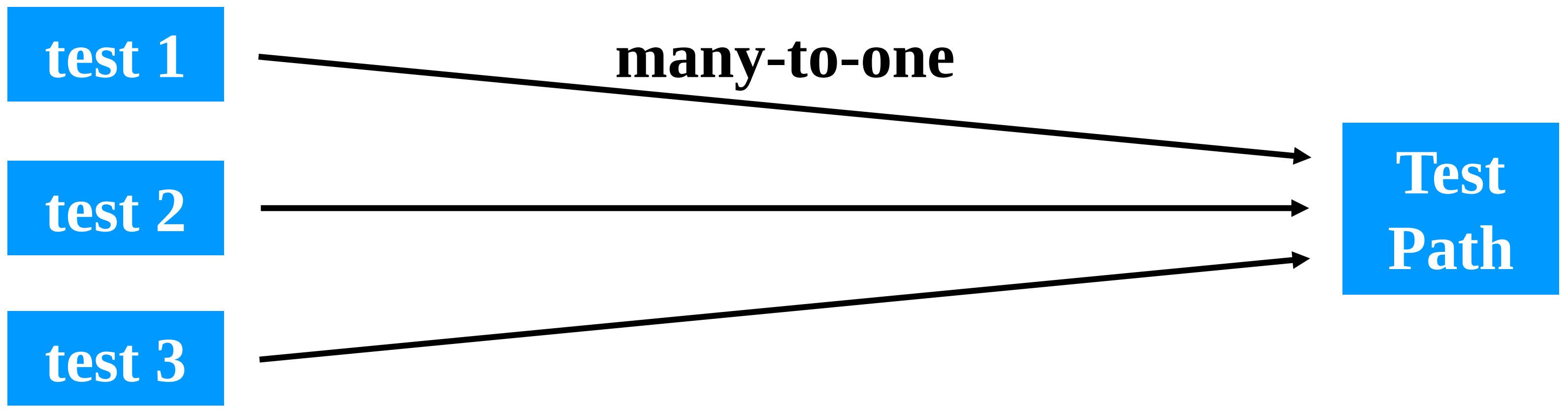
Visits edges (0, 1), (1, 3), (3, 4), (4, 6)

Tours subpaths (0, 1, 3), (1, 3, 4), (3, 4, 6), (0, 1, 3, 4), (1, 3, 4, 6)

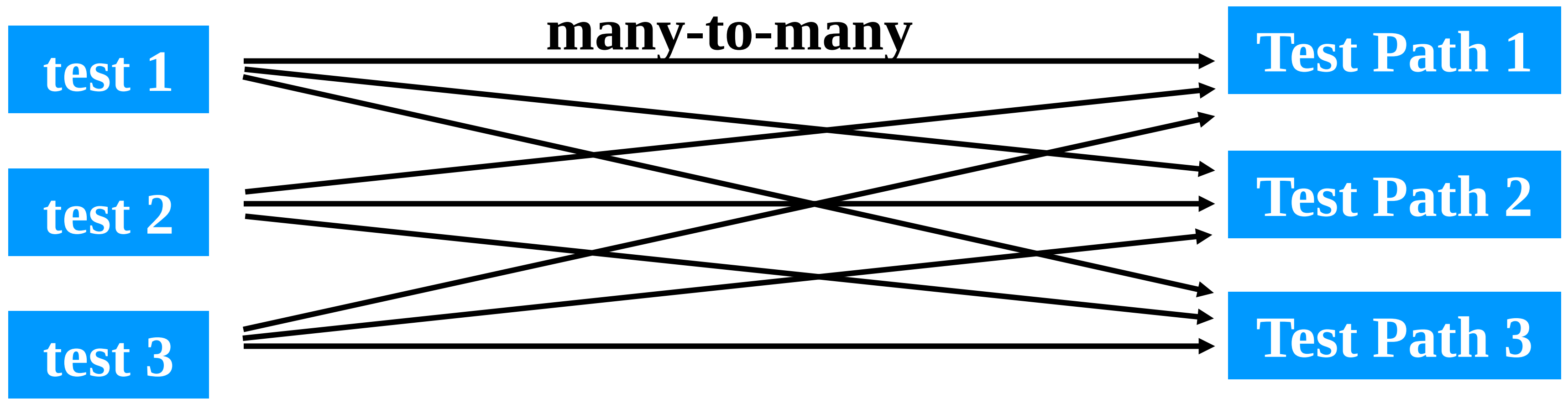
Tests and Test Paths

- path (*t*) : The test path executed by test *t*
- path (*T*) : The set of test paths executed by the set of tests *T*
- Each test executes one and only one test path
- A location in a graph (node or edge) can be reached from another location if there is a sequence of edges from the first location to the second
  - *Syntactic reach* : A subpath exists in the graph
  - *Semantic reach* : A test exists that can execute that subpath

Tests and Test Paths



Deterministic software – a test always executes the same test path



Non-deterministic software – a test can execute different test paths

Testing and Covering Graphs

- We use graphs in testing as follows :
  - Developing a model of the software as a graph
  - Requiring tests to visit or tour specific sets of nodes, edges or subpaths
- Test Requirements (TR) : Describe properties of test paths
- Test Criterion : Rules that define test requirements
- Satisfaction : Given a set TR of test requirements for a criterion C, a set of tests T satisfies C on a graph if and only if for every test requirement in TR, there is a test path in path(T) that meets the test requirement tr
- Structural Coverage Criteria : Defined on a graph just in terms of nodes and edges
- Data Flow Coverage Criteria : Requires a graph to be annotated with references to variables

Structural Coverage Criteria

Node and Edge Coverage

- The first (and simplest) two criteria require that each node and edge in a graph be executed

Node Coverage (NC) : Test set *T* satisfies node coverage on graph *G* iff for every syntactically reachable node *n* in *N*, there is some path *p* in *path(T)* such that *p* visits *n*.

- This statement is a bit cumbersome, so we abbreviate it in terms of the set of test requirements

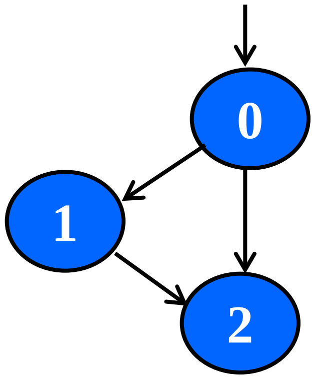
Node Coverage (NC) : TR contains each reachable node in G.

Node and Edge Coverage

- Edge coverage is slightly stronger than node coverage

Edge Coverage (EC) : TR contains each reachable path of length up to 1, inclusive, in G.

- **The "length up to 1" allows for graphs with one node and no edges**
- **NC and EC are only different when there is an edge and another subpath between a pair of nodes (as in an "if-else" statement)**

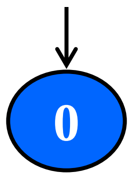


Node Coverage : TR = { 0, 1, 2 }
Test Path = [0, 1, 2]

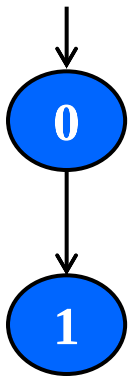
Edge Coverage : TR = { (0,1), (0, 2), (1, 2) }
Test Paths = [0, 1, 2]
[0, 2]

Paths of Length 1 and 0

- A graph with only one node will not have any edges



- **It may be boring, but formally, Edge Coverage needs to require Node Coverage on this graph**
- **Otherwise, Edge Coverage will not subsume Node Coverage**
  - **So we define "length up to 1" instead of simply "length 1"**
- **We have the same issue with graphs that only have one edge – for Edge Pair Coverage …**



Covering Multiple Edges

- Edge-pair coverage requires pairs of edges, or subpaths of length 2

Edge-Pair Coverage (EPC) : TR contains each reachable path of length up to 2, inclusive, in G.

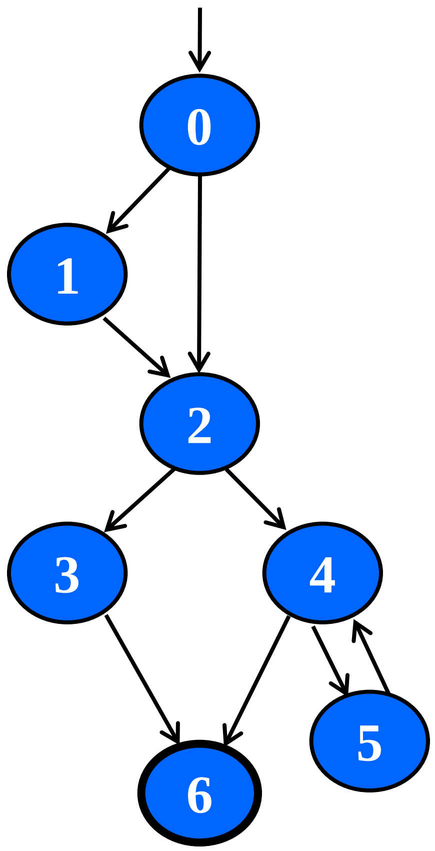
- **The "length up to 2" is used to include graphs that have less than 2 edges**
- **The logical extension is to require all paths …**

Complete Path Coverage (CPC) : TR contains all paths in G.

- **Unfortunately, this is impossible if the graph has a loop, so a weak compromise is to make the tester decide which paths:**

Specified Path Coverage (SPC) : TR contains a set S of test paths, where S is supplied as a parameter.

Structural Coverage Example



Node Coverage

TR = { 0, 1, 2, 3, 4, 5, 6 }

Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 5, 4, 6]

Edge Coverage

TR = { (0,1), (0,2), (1,2), (2,3), (2,4), (3,6), (4,5), (4,6), (5,4) }

Test Paths: [0, 1, 2, 3, 6] [0, 2, 4, 5, 4, 6]

Edge-Pair Coverage

TR = { [0,1,2], [0,2,3], [0,2,4], [1,2,3], [1,2,4], [2,3,6], [2,4,5], [2,4,6], [4,5,4], [5,4,5], [5,4,6] }

Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 6] [0, 2, 3, 6] [0, 2, 4, 5, 4, 5, 4, 6]

Complete Path Coverage

Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 6] [0, 1, 2, 4, 5, 4, 6] [0, 1, 2, 4, 5, 4, 5, 4, 6] [0, 1, 2, 4, 5, 4, 5, 4, 5, 4, 6] …

Loops in Graphs

- If a graph contains a loop, it has an <u>infinite</u> number of paths
- Thus, CPC is <u>not feasible</u>
- SPC is not satisfactory because the results are <u>subjective</u> and vary with the tester
- Attempts to “deal with” loops:
  - 1970s : Execute cycles once ([4, 5, 4] in previous example, informal)
  - 1980s : Execute each loop, exactly once (formalized)
  - 1990s : Execute loops 0 times, once, more than once (informal description)
  - 2000s : Prime paths

Cyclomatic Complexity

- An upper bound on the number of paths through the graph
- Defines the number of tests required to ensure that all statements are executed at least once

$$CC = \#edges - \#nodes + 2$$

- or more formally:

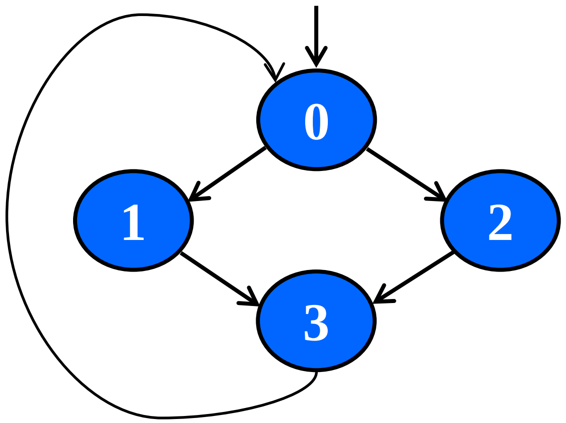
$$CC = E – N + 2$$

where, E = the number of edges and N = the number of nodes

- Approach - Determine the set of linearly independent paths and then
- Prepare test cases:
  - to force execution along each path

Simple Paths and Prime Paths

- <u>Simple Path</u> : *A path from node ni to nj is simple if no node appears more than once, except possibly the first and last nodes are the same*
  - No internal loops
  - Includes all other subpaths
  - A loop is a simple path
- <u>Prime Path</u> : *A simple path that does not appear as a proper subpath of any other simple path*



<u>Simple Paths</u> : [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1], [0, 1, 3], [0, 2, 3], [1, 3, 0], [2, 3, 0], [3, 0, 1], [3, 0, 2], [0, 1], [0, 2], [1, 3], [2, 3], [3, 0], [0], [1], [2], [3]

<u>Prime Paths</u> : [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1]

Prime Path Coverage

- A simple, elegant and finite criterion that requires loops to be executed as well as skipped

Prime Path Coverage (PPC) : TR contains each prime path in G.

- **Will tour all paths of length 0, 1, …**
- **That is, it subsumes node, edge, and edge-pair coverage**

Round Trips

- Round-Trip Path : *A prime path that starts and ends at the same node*

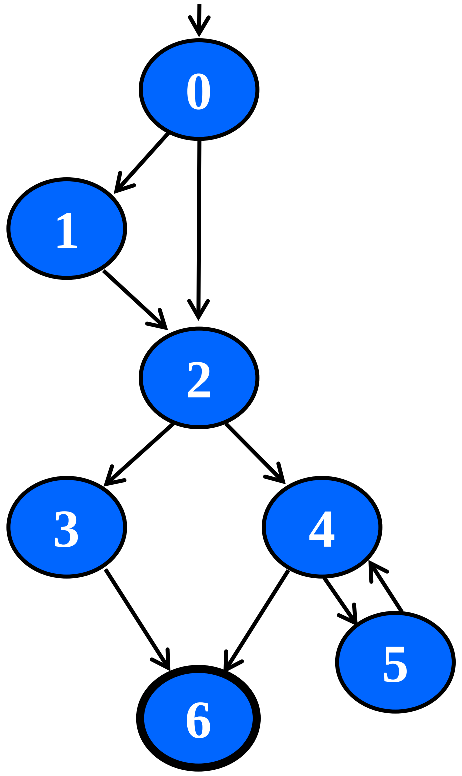
Simple Round Trip Coverage (SRTC) : TR contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path.

Complete Round Trip Coverage (CRTC) : TR contains all round-trip paths for each reachable node in G.

- **These criteria omit nodes and edges that are not in round trips**
- **That is, they do not subsume edge-pair, edge, or node coverage**

Prime Path Example

- The previous example has 38 simple paths
- Only nine *prime paths*



Prime Paths

[0, 1, 2, 3, 6]
[0, 1, 2, 4, 5]
[0, 1, 2, 4, 6]
[0, 2, 3, 6]
[0, 2, 4, 5]
[0, 2, 4, 6]
[5, 4, 6]
[4, 5, 4]
[5, 4, 5]

Execute loop 0 times

Execute loop once

Execute loop more than once

Touring, Sidetrips and Detours

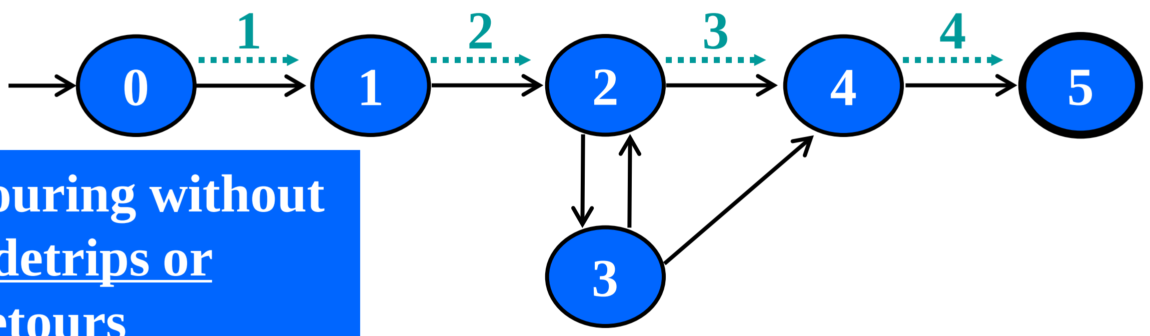
- Prime paths do not have internal loops … test paths might

- Tour : *A test path p tours subpath q if q is a subpath of p*

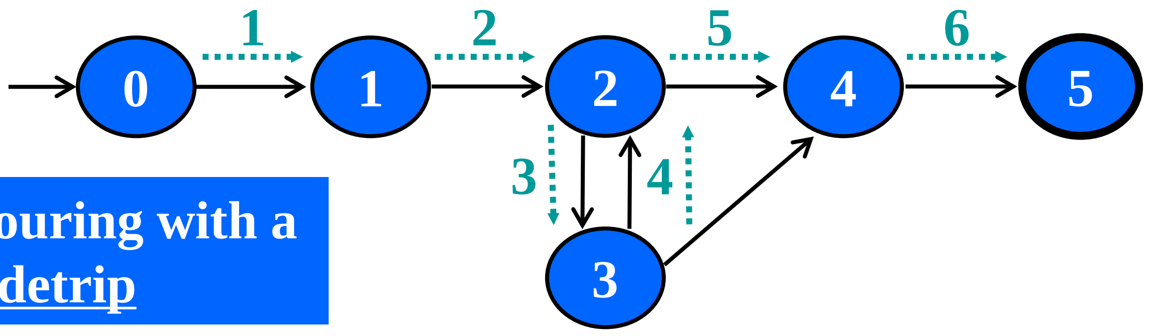
- Tour With Sidetrips : *A test path p tours subpath q with sidetrips iff every edge in q is also in p in the same order*
  - The tour can include a sidetrip, as long as it comes back to the same node

- Tour With Detours : *A test path p tours subpath q with detours iff every node in q is also in p in the same order*
  - The tour can include a detour from node *ni*, as long as it comes back to the prime path at a successor of *ni*

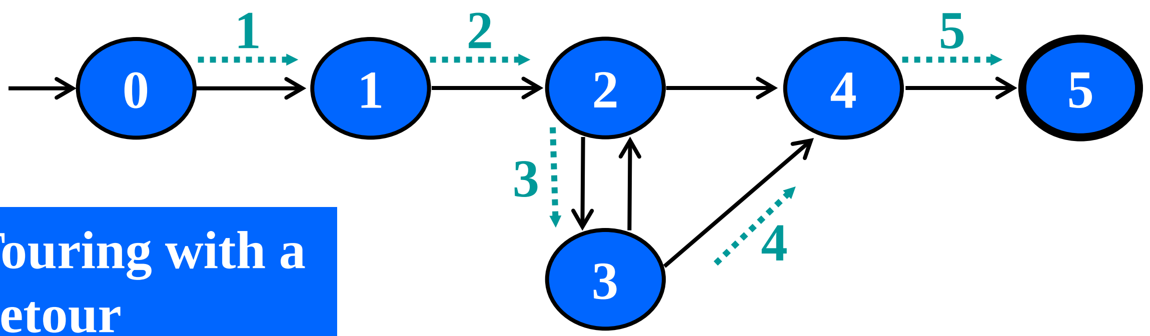
Sidetrips and Detours Example



Touring without sidetrips or detours



Touring with a sidetrip



Touring with a detour

Infeasible Test Requirements

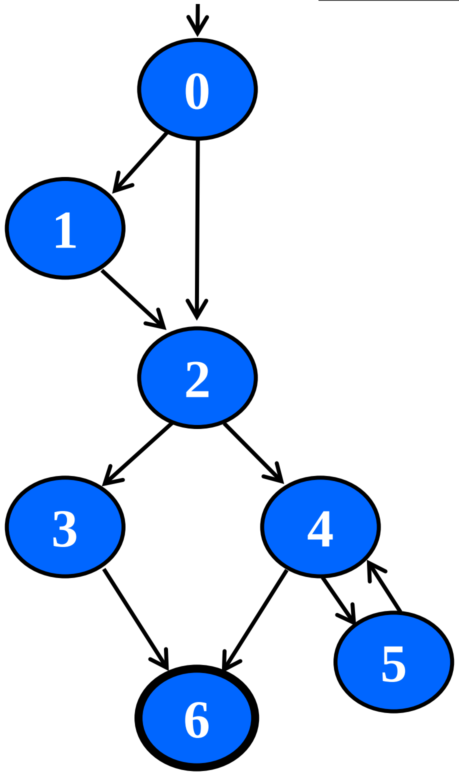
- An infeasible test requirement cannot be satisfied
  - Unreachable statement (dead code)
  - A subpath that can only be executed if a contradiction occurs (*X > 0* and *X < 0*)
- **Most test criteria have some infeasible test requirements**
- **It is usually undecidable whether all test requirements are feasible**
- **When sidetrips are not allowed, many structural criteria have more infeasible test requirements**
- **However, always allowing sidetrips weakens the test criteria**

Practical recommendation – Best Effort Touring

– **Satisfy as many test requirements as possible without sidetrips**

– **Allow sidetrips to try to satisfy unsatisfied test requirements**

Simple & Prime Path Example



‘!’ means path terminates

‘*’ means path cycles

Simple paths

Len 0
[0]
[1]
[2]
[3]
[4]
[5]
[6] !

Len 1
[0, 1]
[0, 2]
[1, 2]
[2, 3]
[2, 4]
[3, 6] !
[4, 6] !
[4, 5]
[5, 4]

Len 2
[0, 1, 2]
[0, 2, 3]
[0, 2, 4]
[1, 2, 3]
[1, 2, 4]
[2, 3, 6] !
[2, 4, 6] !
[2, 4, 5] !
[4, 5, 4] *
[5, 4, 6] !
[5, 4, 5] *

Len 3
[0, 1, 2, 3]
[0, 1, 2, 4]
[0, 2, 3, 6] !
[0, 2, 4, 6] !
[0, 2, 4, 5] !
[1, 2, 3, 6] !
[1, 2, 4, 5] !
[1, 2, 4, 6] !

Len 4
[0, 1, 2, 3, 6] !
[0, 1, 2, 4, 6] !
[0, 1, 2, 4, 5] !

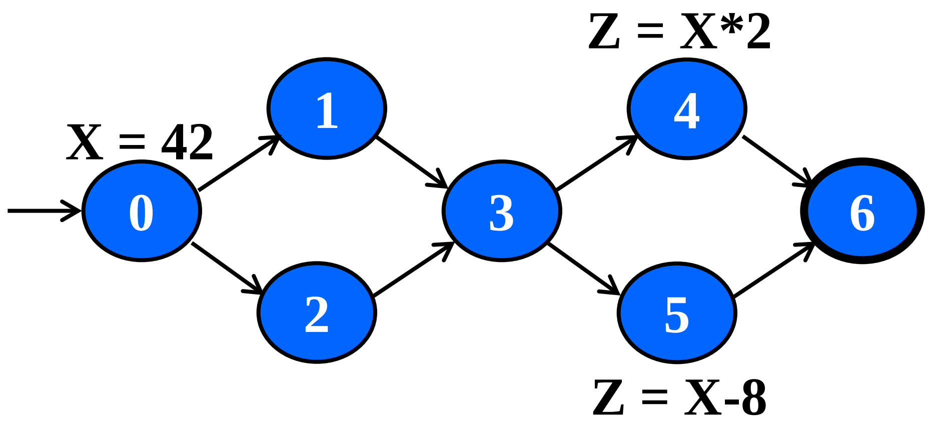
Prime Paths

Data Flow Coverage Criteria

Data Flow Criteria

<u>Goal</u>: Try to ensure that values are computed and used correctly

- <u>Definition (def)</u> : A location where a value for a variable is stored into memory
- <u>Use</u> : A location where a variable's value is accessed
- <u>def (n) or def (e)</u> : The set of variables that are defined by node n or edge e
- <u>use (n) or use (e)</u> : The set of variables that are used by node n or edge e



<u>Defs</u>: def (0) = {X}

def (4) = {Z}

def (5) = {Z}

<u>Uses</u>: use (4) = {X}

use (5) = {X}

DU Pairs and DU Paths

- DU pair : A pair of locations (l_i, l_j) such that a variable *v* is defined at l_i and used at l_j
- Def-clear : A path from l_i to l_j is *def-clear* with respect to variable *v* if *v* is not given another value on any of the nodes or edges in the path
- Reach : If there is a def-clear path from l_i to l_j with respect to *v*, the def of *v* at l_i reaches the use at l_j
- du-path : A simple subpath that is def-clear with respect to *v* from a def of *v* to a use of *v*
- du (n_i, n_j, *v*) – the set of du-paths from n_i to n_j
- du (n_i, *v*) – the set of du-paths that start at n_i

Touring DU-Paths

- A test path *p* *du-tours* subpath *d* with respect to *v* if *p* tours *d* and the subpath taken is def-clear with respect to *v*
- Sidetrips can be used, just as with previous touring
- Three criteria
  - Use every def
  - Get to every use
  - Follow all du-paths

Data Flow Test Criteria

- **First, we make sure every def reaches a use**

<u>All-defs coverage (ADC)</u> : For each set of du-paths *S* = *du* (*n*, *v*), TR contains at least one path *d* in *S*.

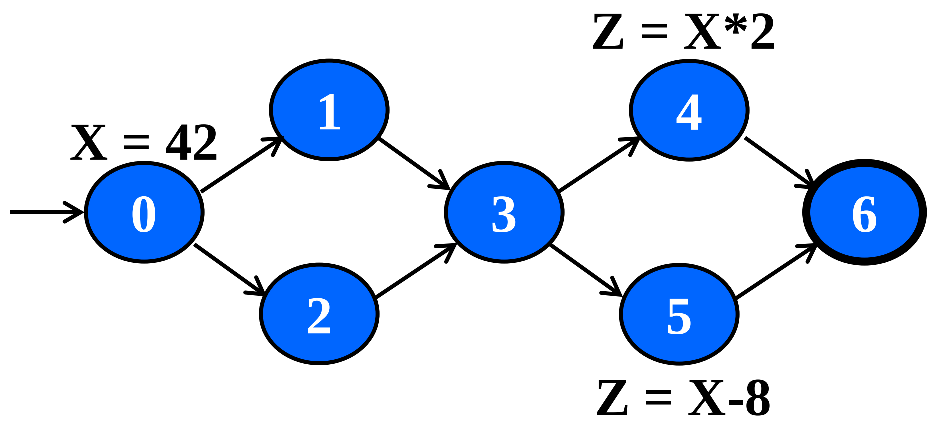
- **Then we make sure that every def reaches all possible uses**

<u>All-uses coverage (AUC)</u> : For each set of du-paths to uses *S* = *du* (n_i, n_j, *v*), TR contains at least one path *d* in *S*.

- **Finally, we cover all the paths between defs and uses**

<u>All-du-paths coverage (ADUPC)</u> : For each set *S* = *du* (*ni*, *nj*, *v*), TR contains every path *d* in *S*.

Data Flow Testing Example

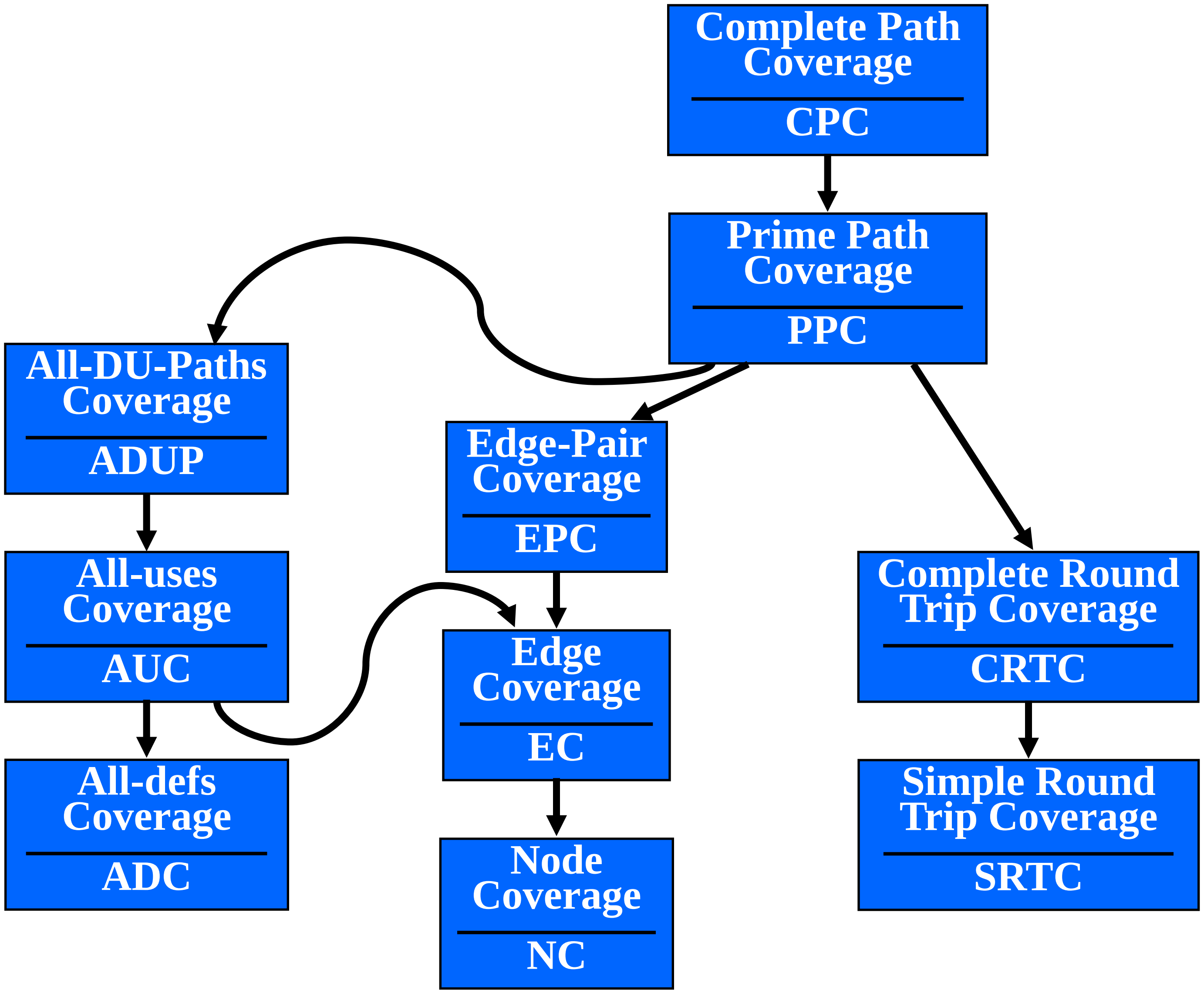


All-defs for *X*
[0, 1, 3, 4]

All-uses for *X*
[0, 1, 3, 4]
[0, 1, 3, 5]

All-du-paths for *X*
[0, 1, 3, 4]
[0, 2, 3, 4]
[0, 1, 3, 5]
[0, 2, 3, 5]

Graph Coverage Criteria Subsumption



Overview: Graph Coverage for Source Code

Overview Graph Coverage for Source Code

- The most common application of graph criteria is to program source
- Graph : Usually the control flow graph (CFG)
- Node coverage : Execute every statement
- Edge coverage : Execute every branch
- Loops : Looping structures such as for loops, while loops, etc.
- Data flow coverage : Augment the CFG
  - defs are statements that assign values to variables
  - uses are statements that use variables

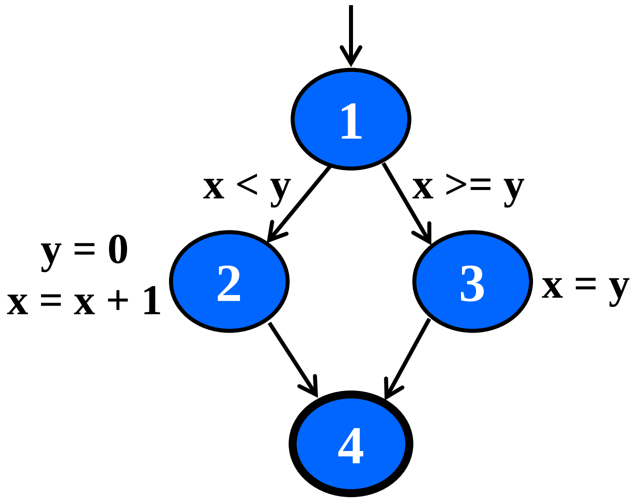
Control Flow Graphs

- A CFG models all executions of a method by describing control structures
- Nodes : Statements or sequences of statements (basic blocks)
- Edges : Transfers of control
- Basic Block : A sequence of statements such that if the first statement is executed, all statements will be (no branches)

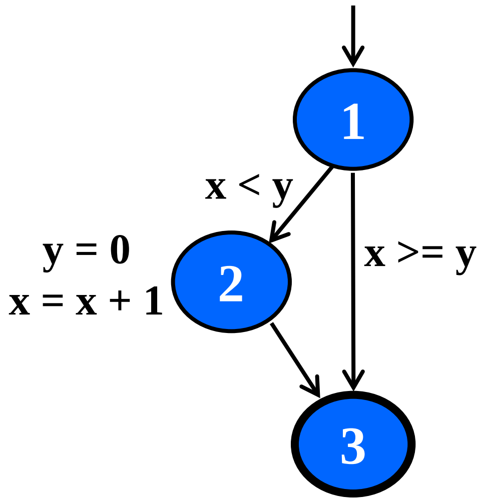
- CFGs are sometimes annotated with extra information
  - branch predicates
  - defs
  - uses
- Rules for translating statements into graphs …

CFG : The if Statement

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
```

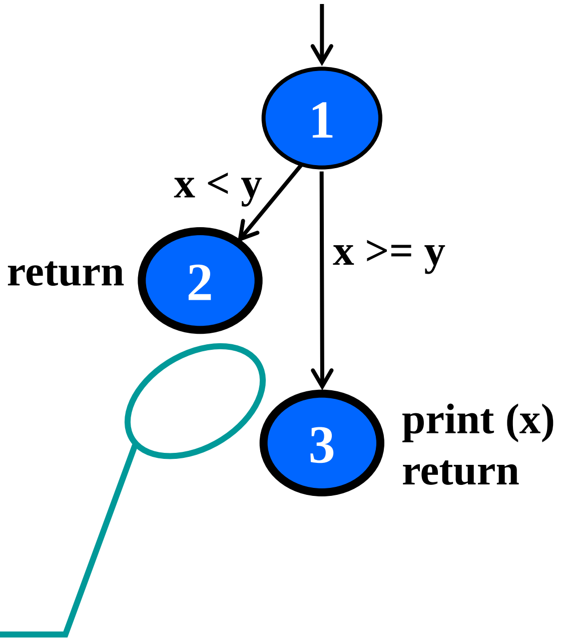


```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```



CFG : The if-Return Statement

```
if (x < y)
{
   return;
}
print (x);
return;
```



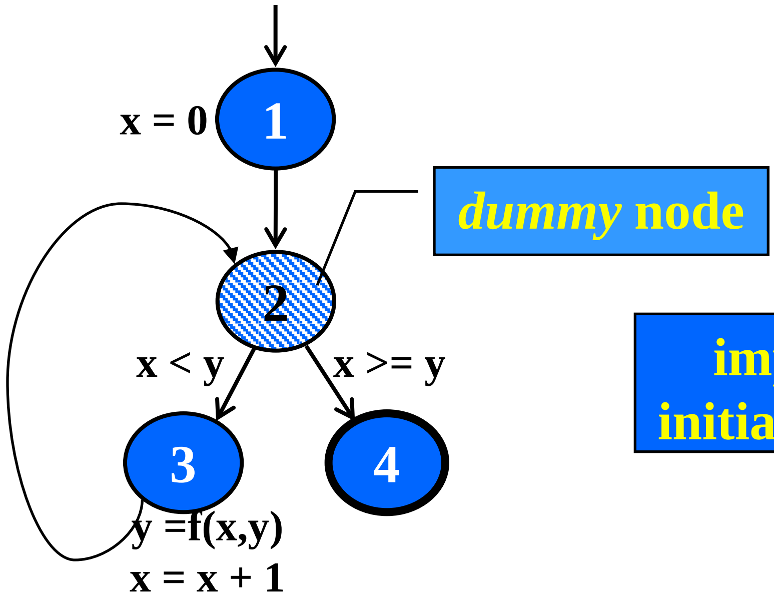
No edge from node 2 to 3.
The return nodes must be distinct.

Loops

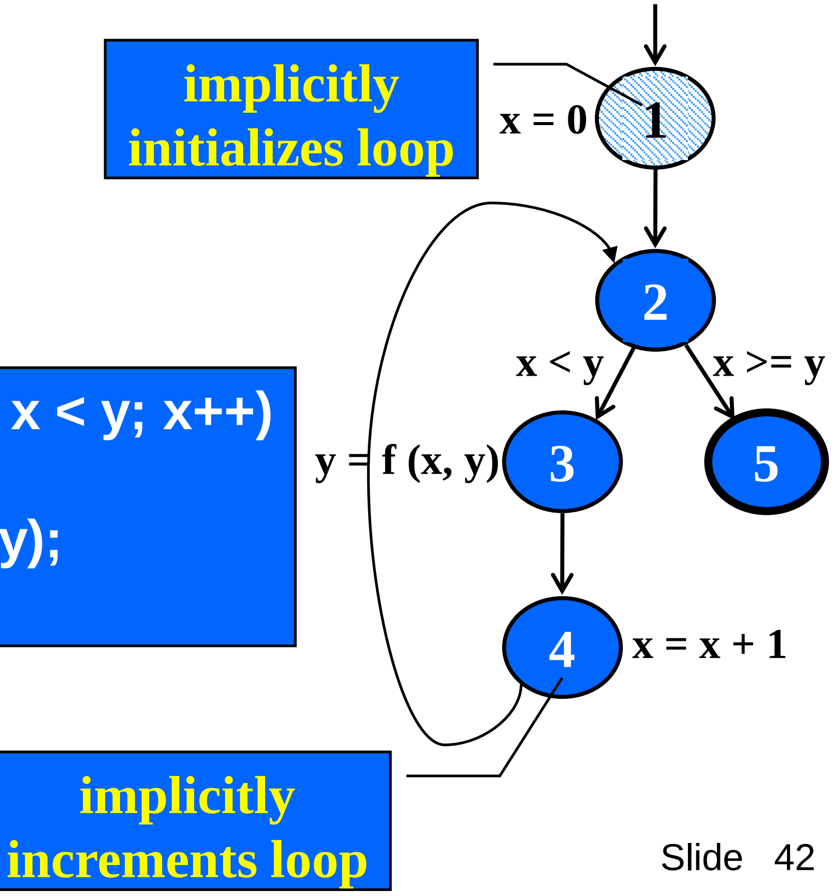
- Loops require “*extra*” nodes to be added
- Nodes that <u>do not</u> represent statements or basic blocks

CFG : while and for Loops

```
x = 0;
while (x < y)
{
   y = f (x, y);
   x = x + 1;
}
```

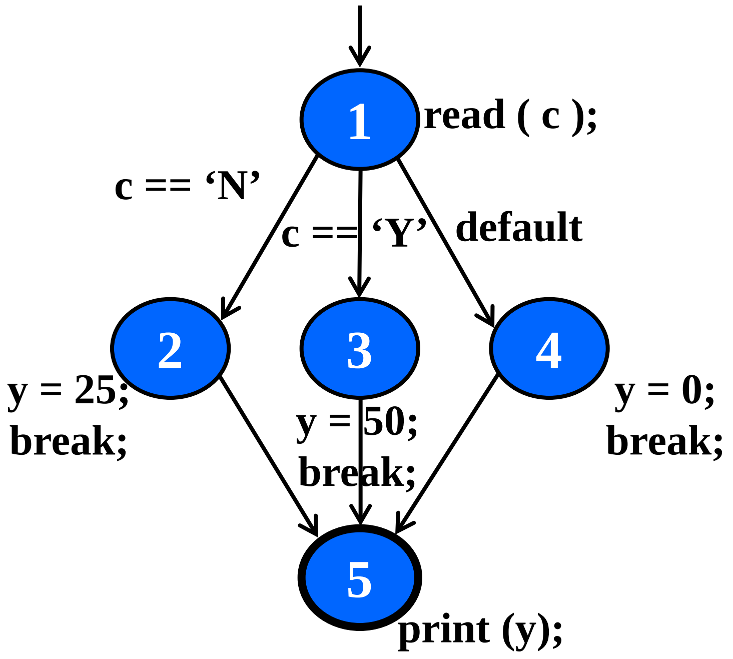


```
for (x = 0; x < y; x++)
{
   y = f (x, y);
}
```



CFG : The case (switch) Structure

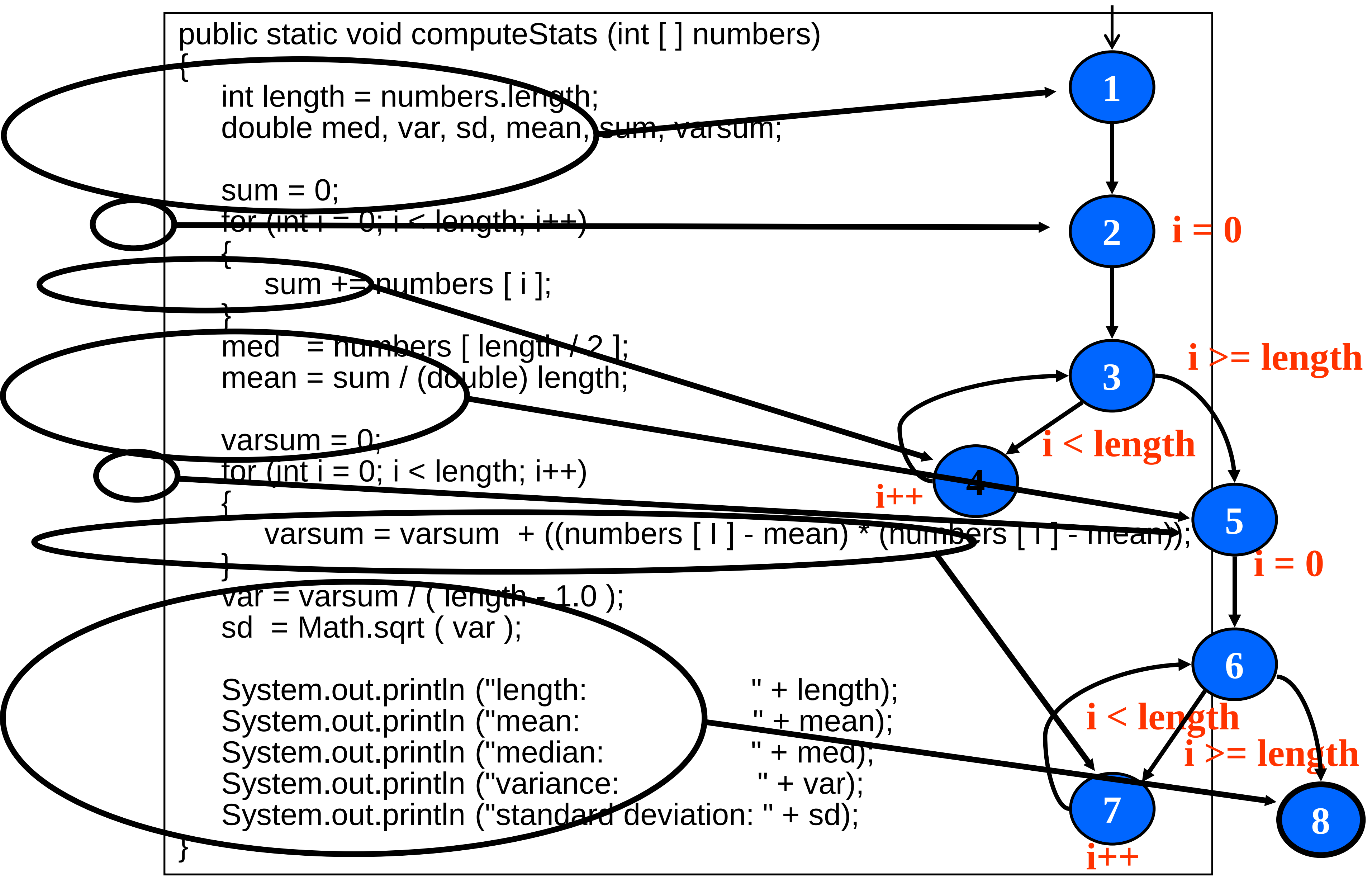
```
read ( c) ;
switch ( c )
{
   case ‘N’:
      y = 25;
      break;
   case ‘Y’:
      y = 50;
      break;
   default:
      y = 0;
      break;
}
print (y);
```



Example Control Flow – Stats

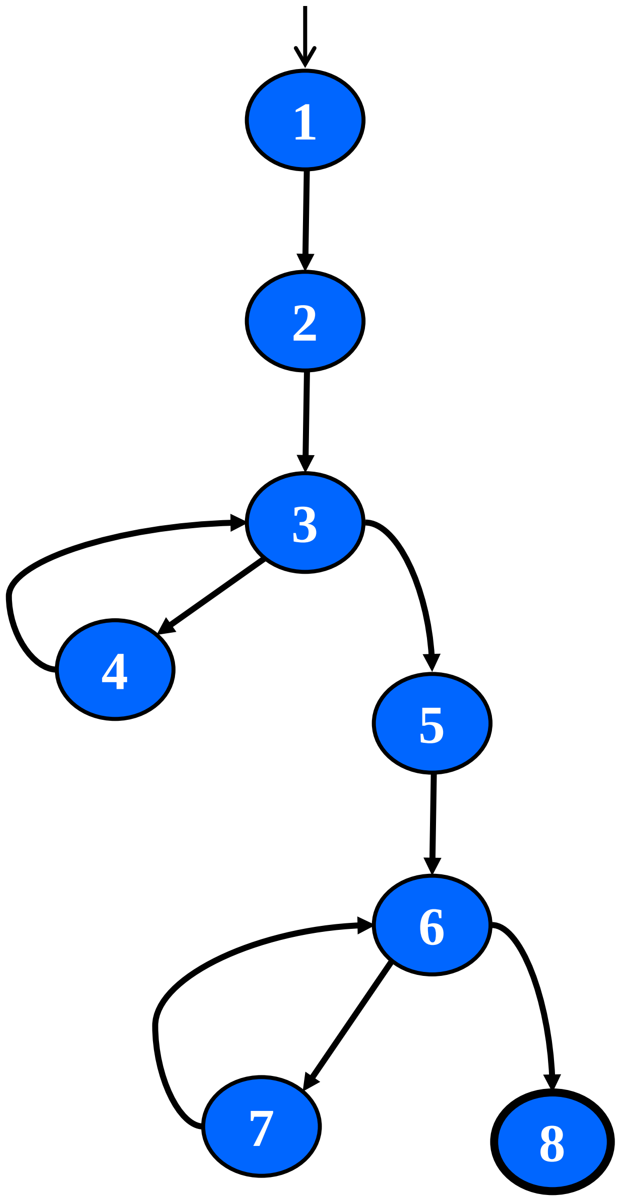
```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2 ];
    mean = sum / (double) length;

    varsum = 0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ I ] - mean) * (numbers [ I ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                   " + length);
    System.out.println ("mean:                     " + mean);
    System.out.println ("median:                   " + med);
    System.out.println ("variance:                 " + var);
    System.out.println ("standard deviation: " + sd);
}
```

Control Flow Graph for Stats



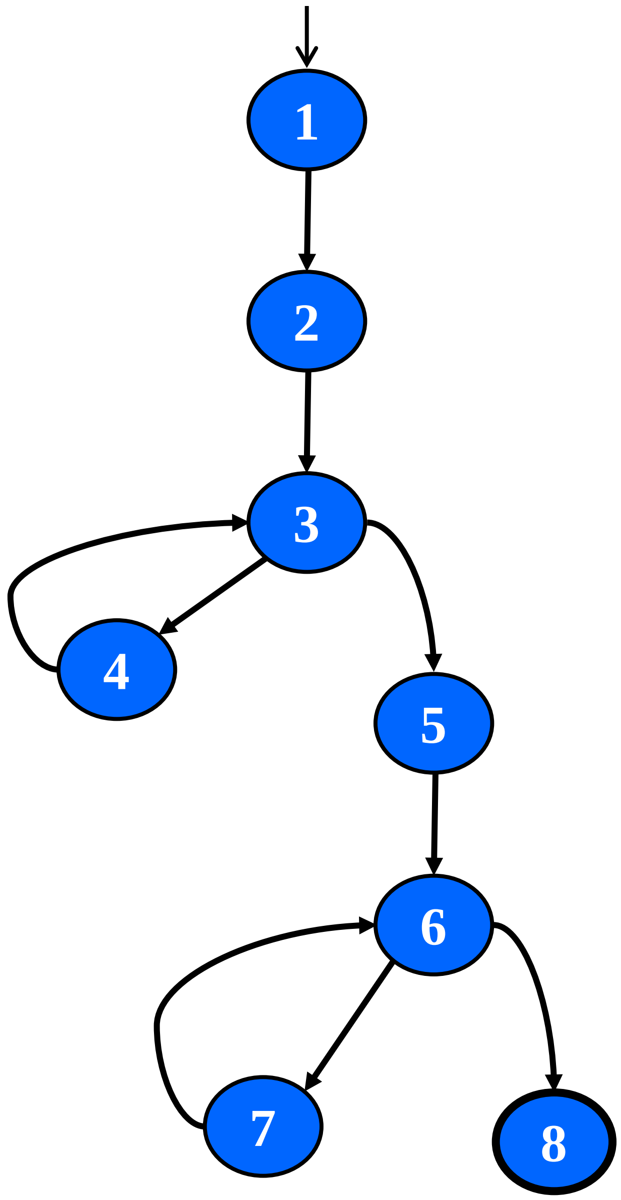
```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2 ];
    mean = sum / (double) length;

    varsum = 0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ I ] - mean) * (numbers [ I ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                    " + length);
    System.out.println ("mean:                      " + mean);
    System.out.println ("median:                    " + med);
    System.out.println ("variance:                  " + var);
    System.out.println ("standard deviation: " + sd);
}
```

Control Flow TRs and Test Paths – EC



Edge Coverage	
TR	**Test Path**
A. [1, 2]	[1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [2, 3]	
C. [3, 4]	
D. [3, 5]	
E. [4, 3]	
F. [5, 6]	
G. [6, 7]	
H. [6, 8]	
I. [7, 6]	

Control Flow TRs and Test Paths – EPC

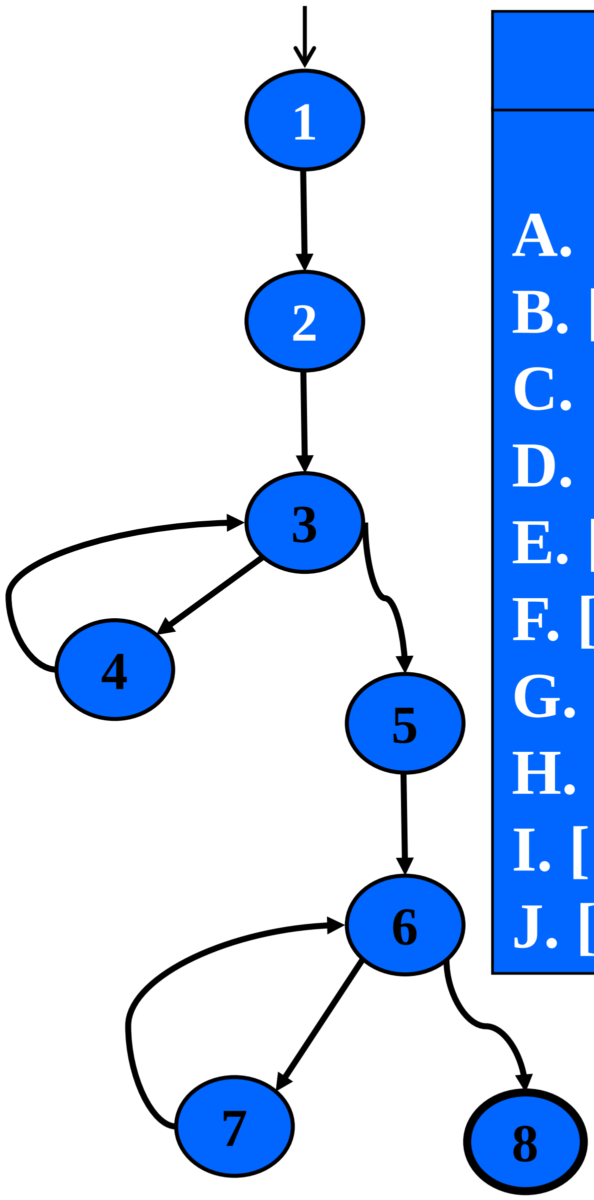

$$\begin{tikzcd}
 & \arrow[d] & \\
 & 1 \arrow[d] & \\
 & 2 \arrow[d] & \\
 & 3 \arrow[dl] \arrow[dr] & \\
4 \arrow[ur, bend left] & & 5 \arrow[d] \\
 & & 6 \arrow[dl] \arrow[dr] \\
 & 7 \arrow[ur, bend left] & & 8
\end{tikzcd}$$

Edge-Pair Coverage

TR	Test Paths
A. [1, 2, 3]	i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [2, 3, 4]	ii. [1, 2, 3, 5, 6, 8]
C. [2, 3, 5]	iii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]
D. [3, 4, 3]	
E. [3, 5, 6]	
F. [4, 3, 5]	
G. [5, 6, 7]	
H. [5, 6, 8]	
I. [6, 7, 6]	
J. [7, 6, 8]	
K. [4, 3, 4]	
L. [7, 6, 7]	

TP	TRs toured	*sidetrips*
i	A, B, D, E, F, G, I, J	C, H
ii	A, C, E, H	
iii	A, B, D, E, F, G, I, J, K, L	C, H

Control Flow TRs and Test Paths – PPC


$$\begin{tikzcd}
 & 1 \arrow[d] & & \\
 & 2 \arrow[d] & & \\
 & 3 \arrow[dl] \arrow[dr] & & \\
4 \arrow[ur, bend left] & & 5 \arrow[d] & \\
 & & 6 \arrow[dl] \arrow[dr] & \\
 & 7 \arrow[ur, bend left] & & 8
\end{tikzcd}$$

Prime Path Coverage

TR	Test Paths
A. [3, 4, 3]	i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [4, 3, 4]	ii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]
C. [7, 6, 7]	iii. [1, 2, 3, 4, 3, 5, 6, 8]
D. [7, 6, 8]	iv. [1, 2, 3, 5, 6, 7, 6, 8]
E. [6, 7, 6]	v. [1, 2, 3, 5, 6, 8]
F. [1, 2, 3, 4]	
G. [4, 3, 5, 6, 7]	
H. [4, 3, 5, 6, 8]	
I. [1, 2, 3, 5, 6, 7]	
J. [1, 2, 3, 5, 6, 8]	

TP	TRs toured	*sidetrips*
i	A, D, E, F, G	H, I, J
ii	A, B, C, D, E, F, G,	H, I, J
iii	A, F, H	J
iv	D, E, F, I	J
v	J	

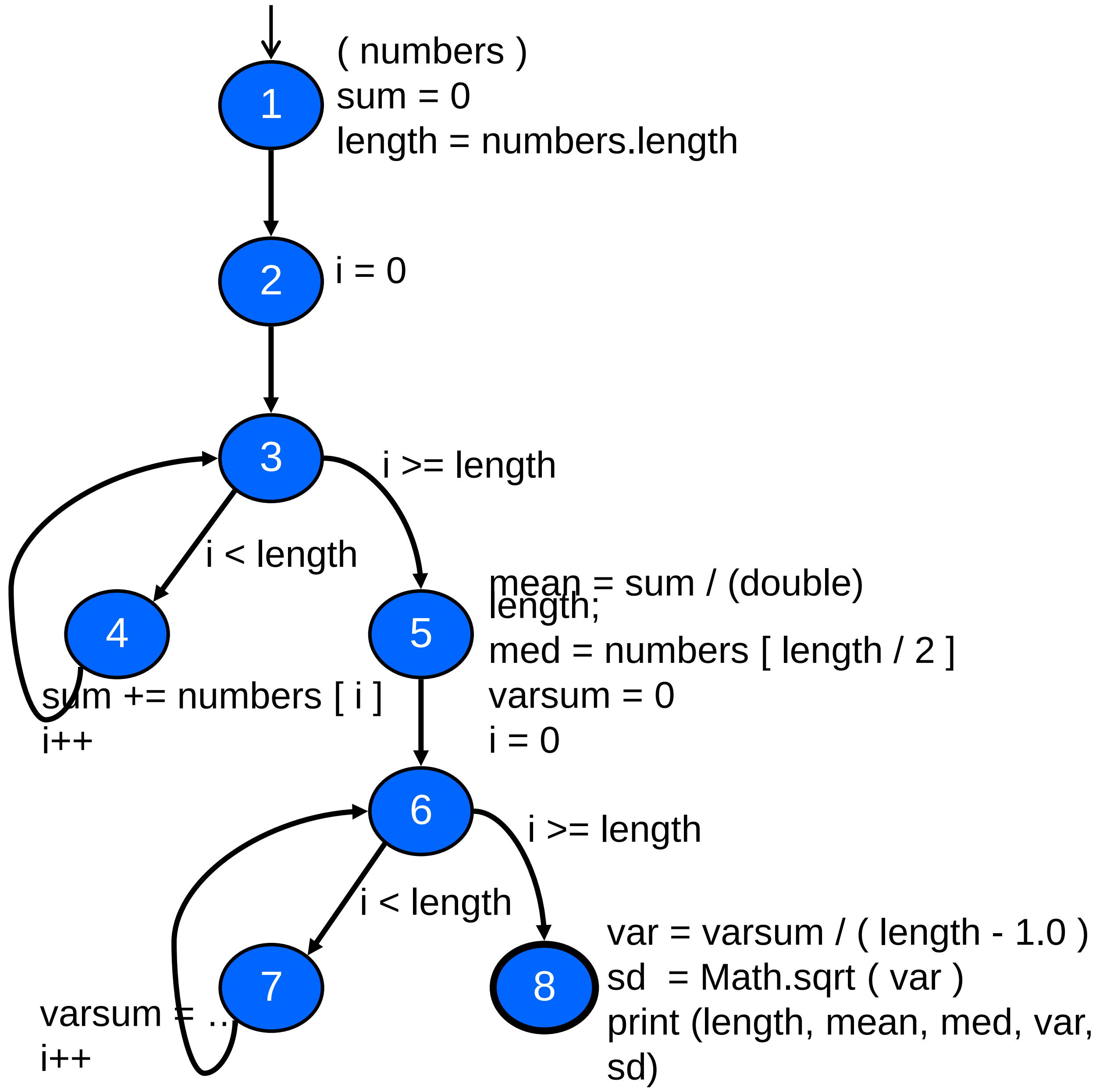
Data Flow Coverage for Source

- def : a location where a value is stored into memory
  - x appears on the left side of an assignment (x = 44;)
  - x is an actual parameter in a call and the method changes its value
  - x is a formal parameter of a method (implicit def when method starts)
  - x is an input to a program
- use : a location where variable's value is accessed
  - x appears on the right side of an assignment
  - x appears in a conditional test
  - x is an actual parameter to a method
  - x is an output of the program
  - x is an output of a method in a return statement
- If a def and a use appear on the same node, then it is only a DU-pair if the def occurs after the use and the node is in a loop

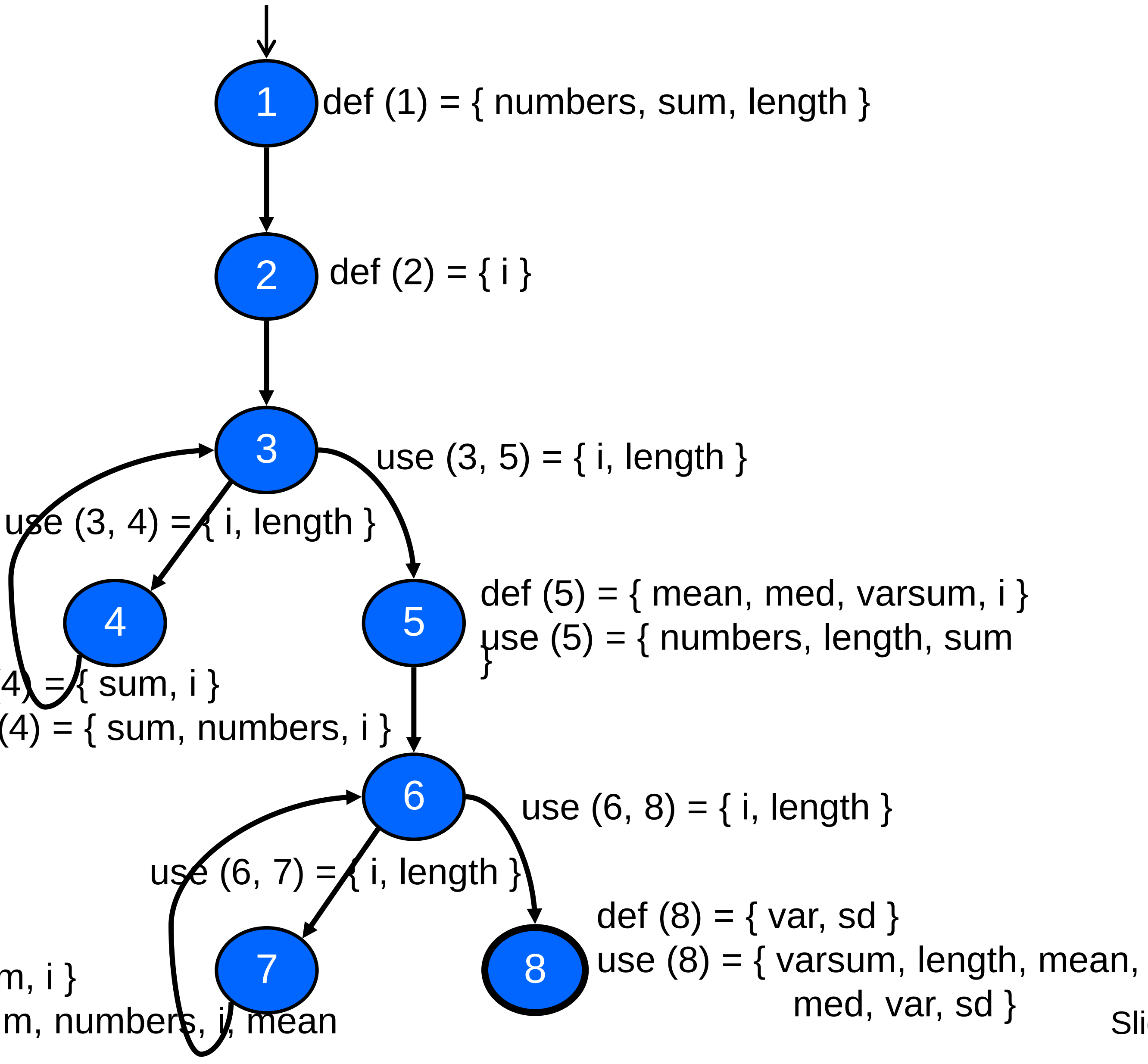
Example Data Flow – Stats

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    mean = sum / (double) length;
    med   = numbers [ length / 2 ];

    varsum = 0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:              " + length);
    System.out.println ("mean:                " + mean);
    System.out.println ("median:              " + med);
    System.out.println ("variance:            " + var);
    System.out.println ("standard deviation: " + sd);
}
```

Control Flow Graph for Stats



CFG for Stats – With Defs & Uses



1
def (1) = { numbers, sum, length }
2
def (2) = { i }
3
use (3, 5) = { i, length }
use (3, 4) = { i, length }
4
5
def (5) = { mean, med, varsum, i }
use (5) = { numbers, length, sum }
def (4) = { sum, i }
use (4) = { sum, numbers, i }
6
use (6, 8) = { i, length }
use (6, 7) = { i, length }
def (8) = { var, sd }
use (8) = { varsum, length, mean, med, var, sd }
7
8
def (7) = { varsum, i }
use (7) = { varsum, numbers, i, mean }

Defs and Uses Tables for Stats

Node	Def	Use
1	{ numbers, sum, length }	
2	{ i }	
3		
4	{ sum, i }	{ numbers, i, sum }
5	{ mean, med, varsum, i }	{ numbers, length, sum }
6		
7	{ varsum, i }	{ varsum, numbers, i, mean }
8	{ var, sd }	{ varsum, length, var, mean, med, var, sd }

Edge	Use
(1, 2)	
(2, 3)	
(3, 4)	{ i, length }
(4, 3)	
(3, 5)	{ i, length }
(5, 6)	
(6, 7)	{ i, length }
(7, 6)	
(6, 8)	{ i, length }

DU Pairs for Stats

variable	DU Pairs
numbers	(1, 4) (1, 5) (1, 7)
length	(1, 5) (1, 8) (1, (3,4)) (1, (3,5)) (1, (6,7)) (1, (6,8))
med	(5, 8)
var	(8, 8)
sd	(8, 8)
mean	(5, 7) (5, 8)
sum	(1, 4) (1, 5) (4, 4) (4, 5)
varsum	(5, 7) (5, 8) (7, 7) (7, 8)
i	(2, 4) (2, (3,4)) (2, (3,5)) ~~(2, 7) (2, (6,7)) (2, (6,8))~~ (4, 4) (4, (3,4)) (4, (3,5)) ~~(4, 7) (4, (6,7)) (4, (6,8))~~ (5, 7) (5, (6,7)) (5, (6,8)) (7, 7) (7, (6,7)) (7, (6,8))

defs come before uses, do not count as DU pairs

defs after use in loop, these are valid DU pairs

No def-clear path … different scope for i

No path through graph from nodes 5 and 7 to 4 or 3

DU Paths for Stats

variable	DU Pairs	DU Paths
numbers	(1, 4) (1, 5) (1, 7)	[1, 2, 3, 4] [1, 2, 3, 5] [1, 2, 3, 5, 6, 7]
length	(1, 5) (1, 8) (1, (3,4)) (1, (3,5)) (1, (6,7)) (1, (6,8))	[1, 2, 3, 5] [1, 2, 3, 5, 6, 8] [1, 2, 3, 4] [1, 2, 3, 5] [1, 2, 3, 5, 6, 7] [1, 2, 3, 5, 6, 8]
med	(5, 8)	[5, 6, 8]
var	(8, 8)	No path needed
sd	(8, 8)	No path needed
sum	(1, 4) (1, 5) (4, 4) (4, 5)	[1, 2, 3, 4] [1, 2, 3, 5] [4, 3, 4] [4, 3, 5]

variable	DU Pairs	DU Paths
mean	(5, 7) (5, 8)	[5, 6, 7] [5, 6, 8]
varsum	(5, 7) (5, 8) (7, 7) (7, 8)	[5, 6, 7] [5, 6, 8] [7, 6, 7] [7, 6, 8]
i	(2, 4) (2, (3,4)) (2, (3,5)) (4, 4) (4, (3,4)) (4, (3,5)) (5, 7) (5, (6,7)) (5, (6,8)) (7, 7) (7, (6,7)) (7, (6,8))	[2, 3, 4] [2, 3, 4] [2, 3, 5] [4, 3, 4] [4, 3, 4] [4, 3, 5] [5, 6, 7] [5, 6, 7] [5, 6, 8] [7, 6, 7] [7, 6, 7] [7, 6, 8]

DU Paths for Stats – No Duplicates

There are 38 DU paths for Stats, but only 12

✦	[1, 2, 3, 4]	[4, 3, 4]	✸
★	[1, 2, 3, 5]	[4, 3, 5]	✦
★	[1, 2, 3, 5, 6, 7]	[5, 6, 7]	✦
★	[1, 2, 3, 5, 6, 8]	[5, 6, 8]	★
✦	[2, 3, 4]	[7, 6, 7]	✸
★	[2, 3, 5]	[7, 6, 8]	✦

★ 5 expect a loop not to be “entered”

✦ 5 require at least one iteration of a loop

✸ 2 require at least two iteration of a loop

More on Test Paths

Test Path : [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]

<u>Additional DU Paths covered (no sidetrips)</u>

[1, 2, 3, 4]   [2, 3, 4]   [4, 3, 5]   [5, 6, 7]   [7, 6, 8]

The five stars that require at least one iteration of a loop

Test Path : [1, 2, 3, 4, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 7, 6, 8]

<u>DU Paths covered (no sidetrips)</u>

[4, 3, 4]   [7, 6, 7]

The two stars that require at least two iterations of a loop

Other DU paths require arrays with length 0 to skip loops

But the method fails with divide by zero on the statement …

```
    mean = sum / (double) length;
```

A fault was found

Quick review

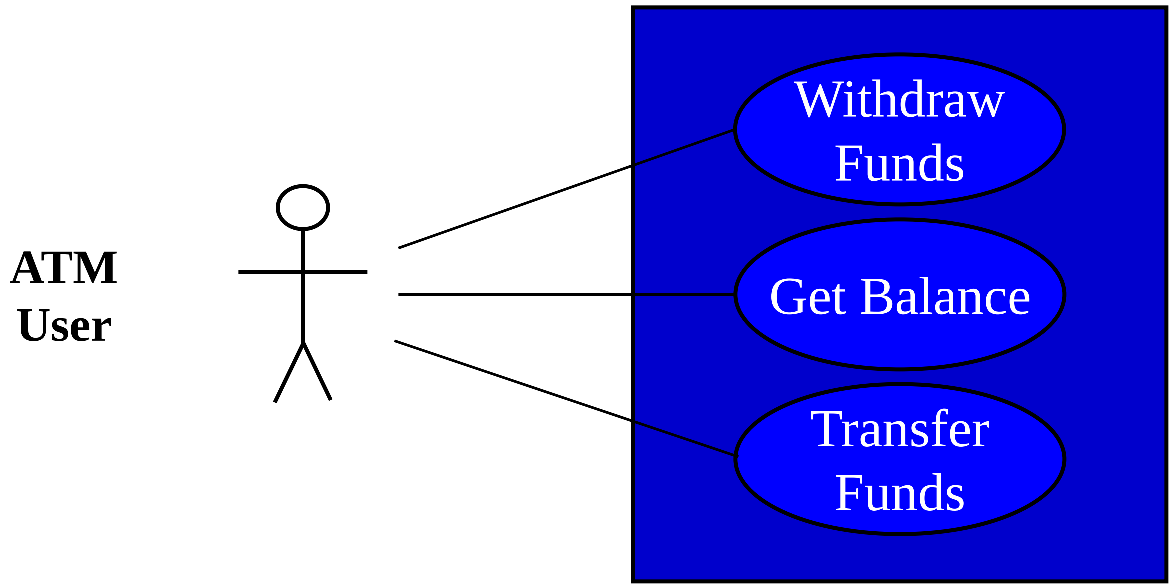
- Applying the graph test criteria to control flow graphs is relatively straightforward
  - Most of the developmental research work was done with CFGs
- A few subtle decisions must be made to translate control structures into the graph
- Some tools will assign each statement to a unique node
  - These slides uses basic blocks
  - Coverage is the same, although the bookkeeping will differ

Graph coverage for UML Use Cases

UML Use Cases

- UML use cases are often used to express software requirements
- They help express computer application workflow
- We won’t teach use cases, but show examples

Simple Use Case Example



- Actors : Humans or software components that use the software being modeled
- Use cases : Shown as circles or ovals
- Node Coverage : Try each use case once …

Use Case graphs, by themselves, are not useful for testing

Elaboration

- Use cases are commonly <u>elaborated</u> (or <u>documented</u>)

- Elaboration is first written <u>textually</u>
  - <u>Details</u> of operation
  - <u>Alternatives</u> model choices and conditions during execution

Elaboration of ATM Use Case

- Use Case Name : Withdraw Funds
- Summary : Customer uses a valid card to withdraw funds from a valid bank account.
- Actor : ATM Customer
- Precondition : ATM is displaying the idle welcome message
- Description :
  – Customer inserts an ATM Card into the ATM Card Reader.
  – If the system can recognize the card, it reads the card number.
  – System prompts the customer for a PIN.
  – Customer enters PIN.
  – System checks the card's expiration date and whether the card has been stolen or lost.
  – If the card is valid, the system checks if the entered PIN matches the card PIN.
  – If the PINs match, the system finds out what accounts the card can access.

Elaboration of ATM Use Case – (2/3)

- Description (continued) :
  - System displays customer accounts and prompts the customer to choose a type of transaction. There are three types of transactions, Withdraw Funds, Get Balance and Transfer Funds. (The previous eight steps are part of all three use cases; the following steps are unique to the Withdraw Funds use case.)
  - Customer selects Withdraw Funds, selects the account number, and enters the amount.
  - System checks that the account is valid, makes sure that customer has enough funds in the account, makes sure that the daily limit has not been exceeded, and checks that the ATM has enough funds.
  - If all four checks are successful, the system dispenses the cash.
  - System prints a receipt with a transaction number, the transaction type, the amount withdrawn, and the new account balance.
  - System ejects card.
  - System displays the idle welcome message.

Elaboration of ATM Use Case – (3/3)

- Alternatives :
  - If the system cannot recognize the card, it is ejected and the welcome message is displayed.
  - If the current date is past the card's expiration date, the card is confiscated and the welcome message is displayed.
  - If the card has been reported lost or stolen, it is confiscated and the welcome message is displayed.
  - If the customer entered PIN does not match the PIN for the card, the system prompts for a new PIN.
  - If the customer enters an incorrect PIN three times, the card is confiscated and the welcome message is displayed.
  - If the account number entered by the user is invalid, the system displays an error message, ejects the card and the welcome message is displayed.
  - If the request for withdraw exceeds the maximum allowable daily withdrawal amount, the system displays an apology message, ejects the card and the welcome message is displayed.
  - If the request for withdraw exceeds the amount of funds in the ATM, the system displays an apology message, ejects the card and the welcome message is displayed.
  - If the customer enters Cancel, the system cancels the transaction, ejects the card and the welcome message is displayed.
- Postcondition :
  - Funds have been withdrawn from the customer's account.

Wait A Minute …

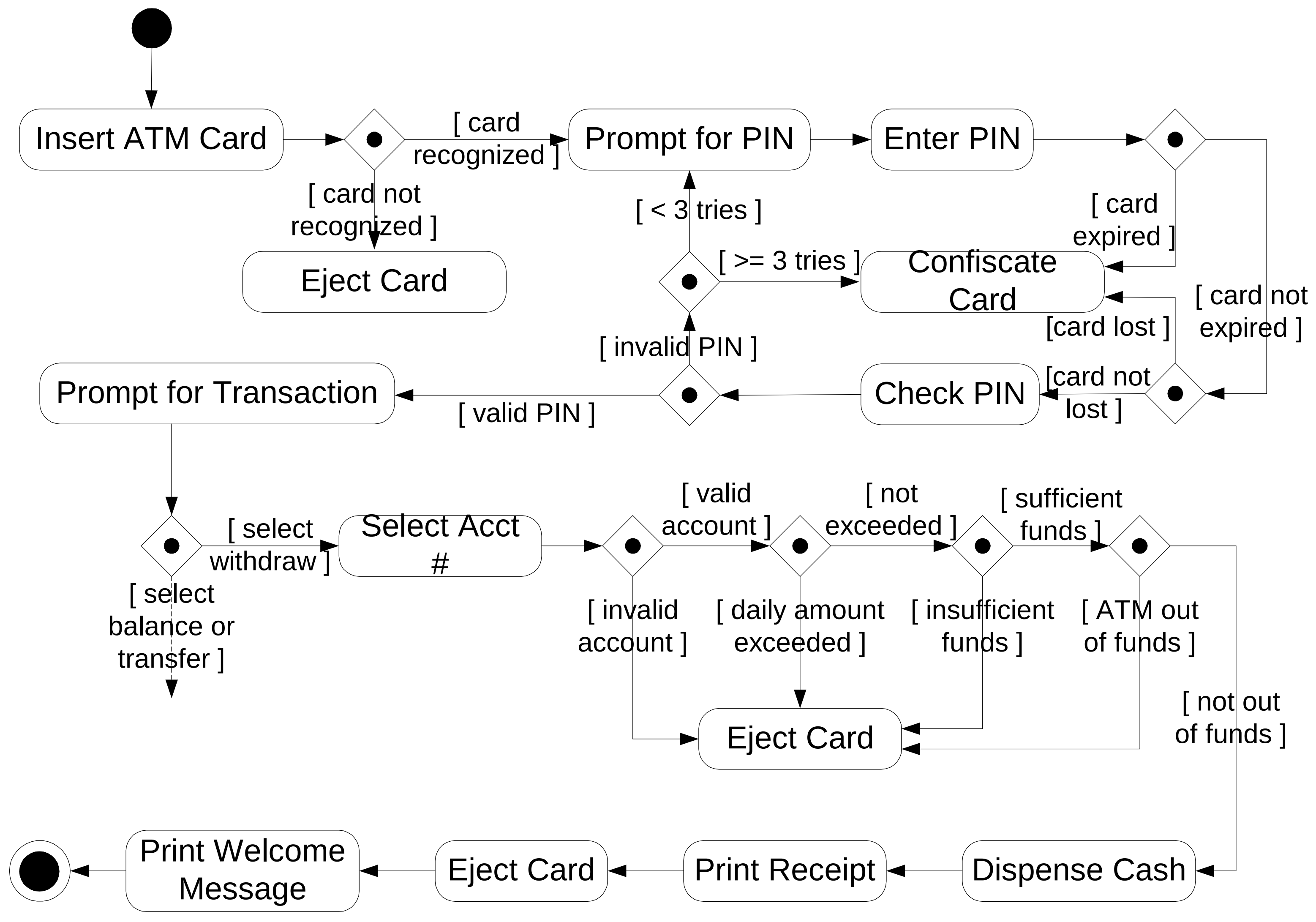
- What does this have to do with testing ?
- Specifically, what does this have to do with <u>graphs</u> ???
- Remember our admonition : <u>Find</u> a graph, then cover it!
- Beizer suggested "<u>Transaction Flow Graphs</u>" in his book
- UML has something very similar :

Activity Diagrams

Use Cases to Activity Diagrams

- Activity diagrams indicate flow among activities
- Activities should model user level steps
- Two kinds of nodes:
  - Action states
  - Sequential branches
- Use case descriptions become action state nodes in the activity diagram
- Alternatives are sequential branch nodes
- Flow among steps are edges
- Activity diagrams usually have some helpful characteristics:
  - Few loops
  - Simple predicates
  - No obvious DU pairs

ATM Withdraw Activity Graph



Covering Activity Graphs

- Node Coverage
  - Inputs to the software are derived from labels on nodes and predicates
  - Used to from test case values
- Edge Coverage
- Data flow techniques do not apply
- Scenario Testing
  - <u>Scenario</u> : A complete path through a use case activity graph
  - Should make <u>semantic</u> sense to the users
  - Number of paths often <u>finite</u>
  - If not, scenarios defined based on <u>domain knowledge</u>
  - Use “<u>specified path coverage</u>”, where the set S of paths is the set of scenarios
  - Note that specified path coverage does not necessarily subsume edge coverage, but scenarios <u>should be</u> defined so that it does

References

- Paul Ammann and Jeff Offutt, *Introduction to Software Testing*, Cambridge University Press, 2008