Graph Coverage Criteria

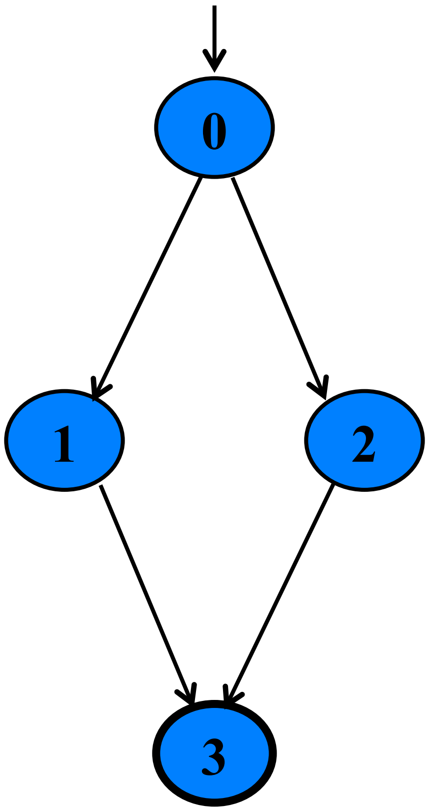
Covering Graphs

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

Definition of a Testable Graph

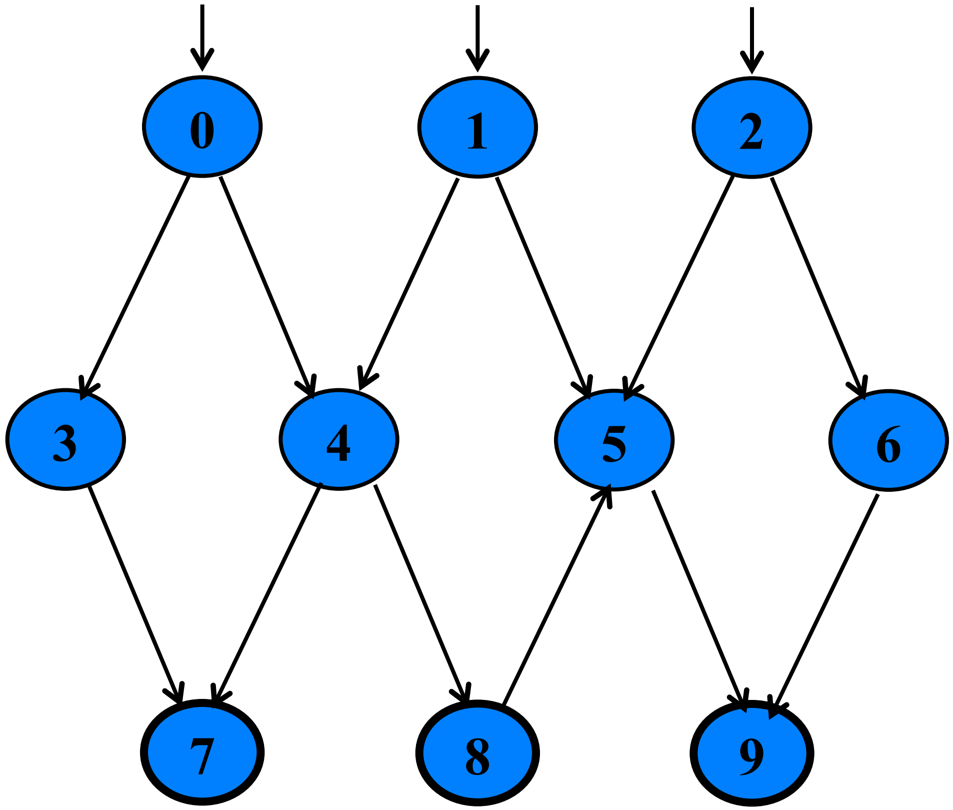
- A set N of nodes, N is not empty
- A set N_0 of initial nodes, N_0 is not empty
- A set N_f of final nodes, N_f is not empty
- A set E of edges, each edge from one node to another
  - (n_i , n_j), i is predecessor, j is successor

Three Example Graphs



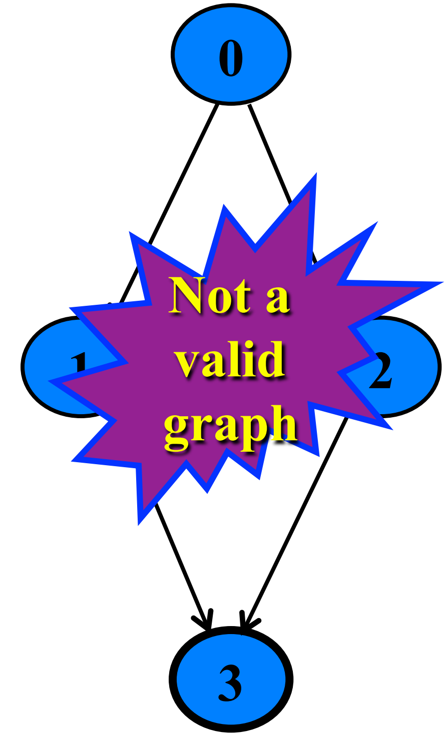
$N_0 = \{ 0 \}$

$N_f = \{ 3 \}$



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

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

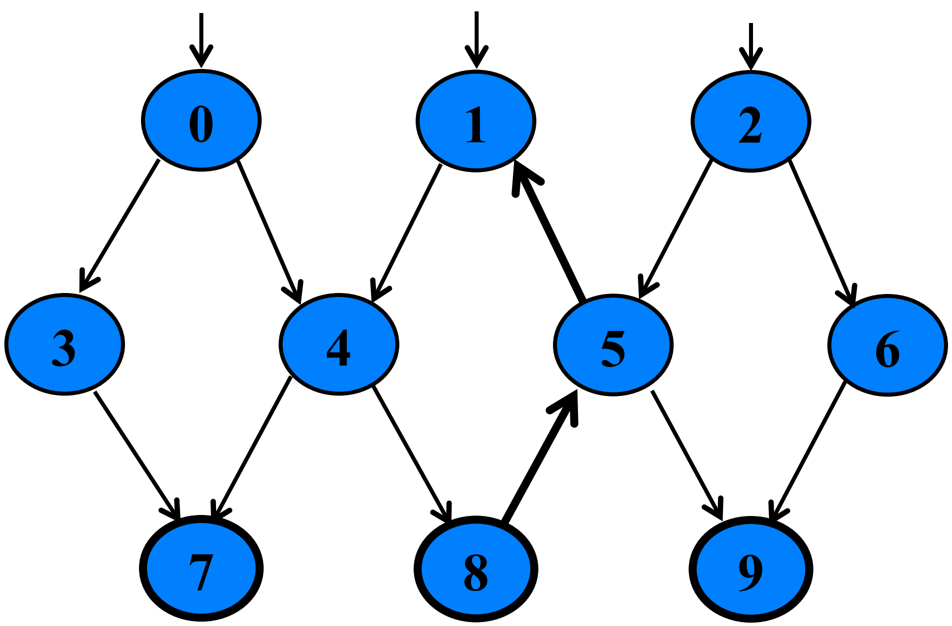


$N_0 = \{ \}$

$N_f = \{ 3 \}$

Paths in Graphs

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



<u>A Few Paths</u>

[0, 3, 7]

[1, 4, 8, 5, 1]

[2, 6, 9]

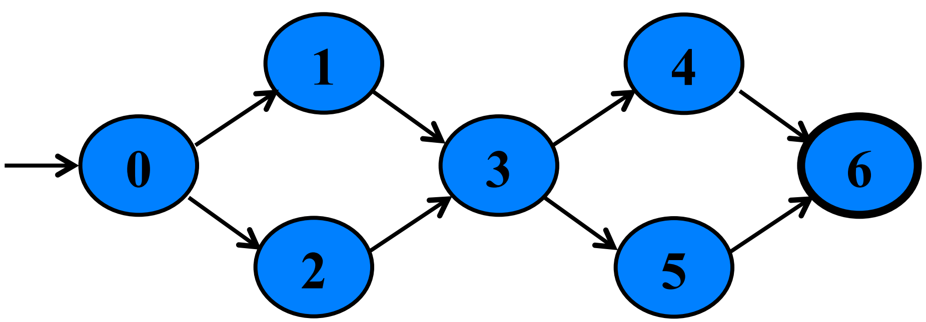
Reach (0) = { 0, 3, 4, 7, 8, 5, 1, 9 }

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



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

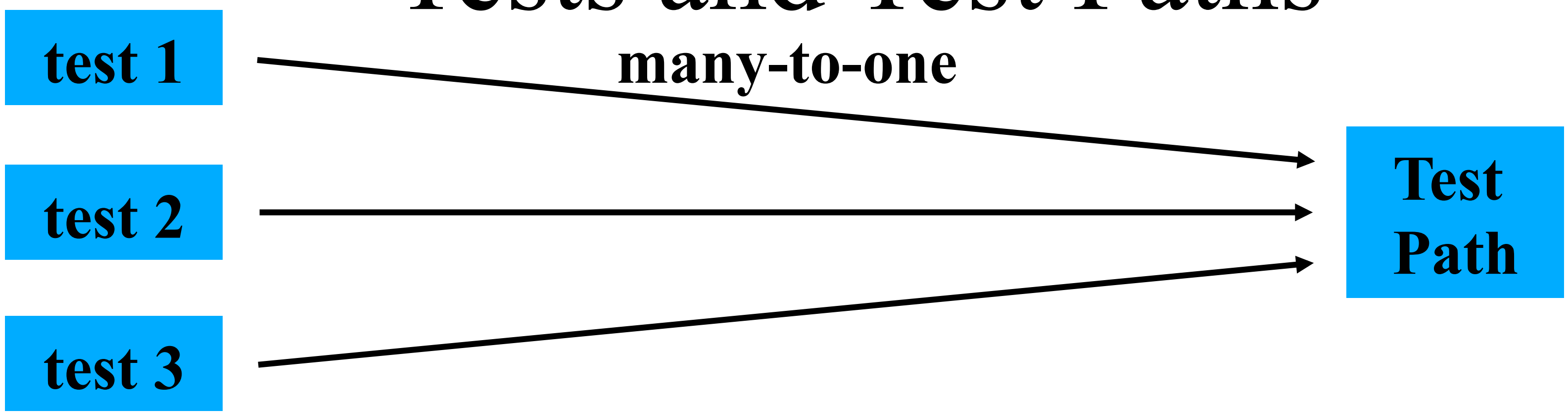
Visits edges (0, 1), (1, 3), (3, 4), (4, 6)

Tours subpaths [0, 1, 3], [1, 3, 4], [3, 4, 6], [0, 1, 3, 4], [1, 3, 4, 6]

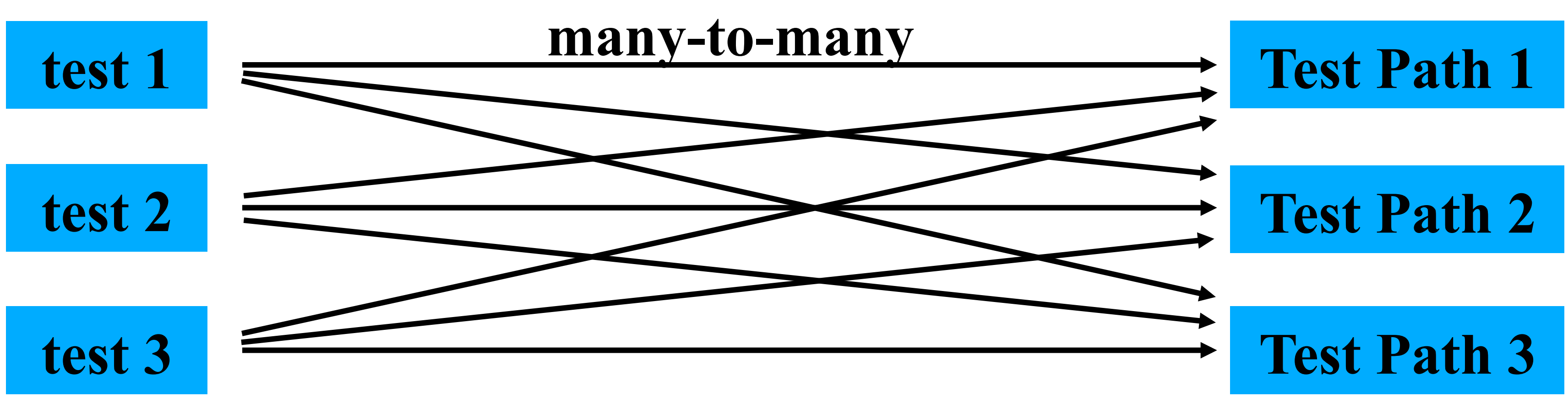
Tests and Test Paths

- 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

What to test

- **Test Requirements (TR) : Describe properties of test paths**
- **Test Criterion : Rules that define test requirements**
- **Satisfaction : *Given a set TR of test requirements fo on C, a set of tests T satisfies C on a graph if and only i ry test requirement in TR, there is a test path in path(T) that meets the test requirement tr***

How to stop?

- **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**

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.

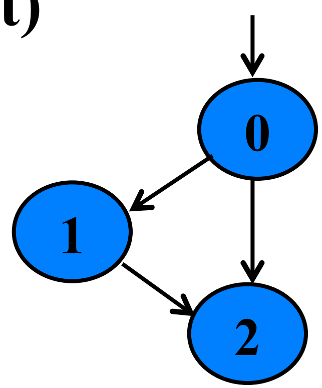
Node and Edge Coverage

- Edge coverage is slightly stronger than node coverage

<u>Edge Coverage (EC)</u> : 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)**

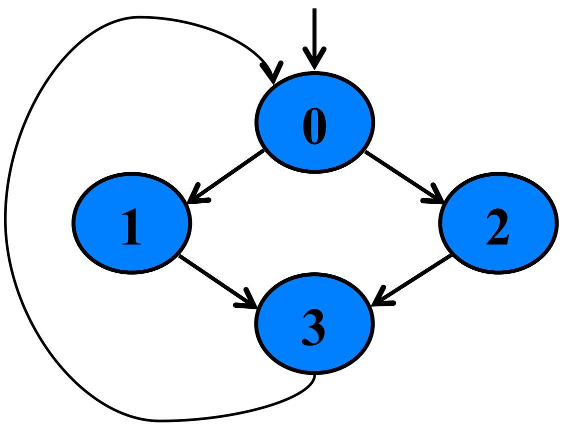


<u>Node Coverage</u> : TR = { 0, 1, 2 }
Test Path = [0, 1, 2]

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

Simple Paths and Prime Paths

- Simple Path : *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
- Prime Path : *A simple path that does not appear as a proper subpath of any other simple path*



Simple Paths : [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]

Prime Paths : [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

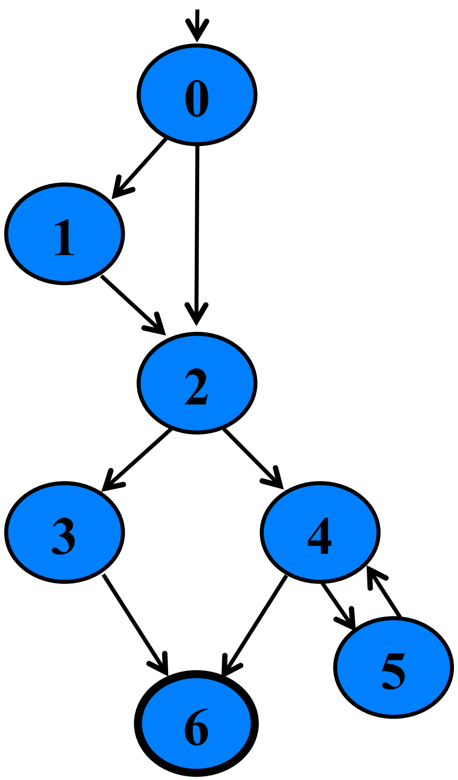
- A simple, elegant and finite criterion that requires loops to be executed as well as skipped

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

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

Prime Path Example

- The 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

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 or node coverage**

Basis Path Coverage

- Independent ***test path***: any test path that introduces at least one new node.
- Basis path set:
  - A minimum set of distinct independent test paths
  - All other paths can be composed out of the combinations of the set

Basis Coverage (BC) : TR contains each independent path in the Basis Path Set.

Cyclomatic Complexity Number

Determine the cyclomatic complexity of the graph.

cyclomatic complexity: A quantitative measure of the logical complexity of the code.

Cyclomatic complexity provides an upper bound on the number of paths that need to be tested in the code.

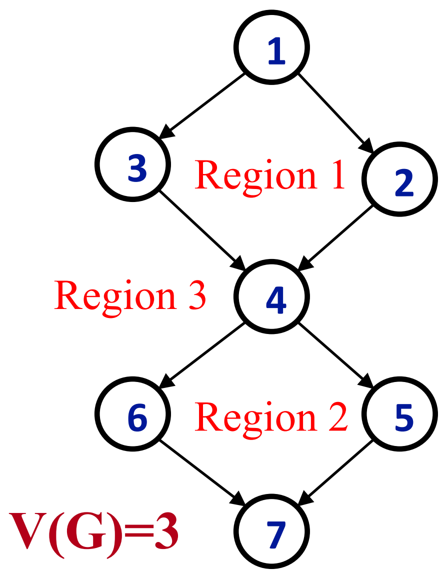
Ways to compute cyclomatic complexity V(G):

- ☞ V(G) = the number of regions (areas bounded by nodes and edges—area outside the graph is also a region)
- ☞ V(G) = the number of edges - the number of nodes + 2
- ☞ V(G) = the number of (simple) predicate nodes + 1

BASIS PATH TESTING: INDEPENDENT PATHS

Procedure: example()

1. If c1
2. f1()
3. Else
4. f2()
5. Endif
6. If c2
7. f3()
8. Else
9. f4()
10. Endif
11. End



Basis set 1 :

1. 1 2 4 5 7
2. 1 3 4 5 7
3. 1 3 4 6 7

The basis set is not unique!

How many independent paths are there in the basis set?

Recall: An independent path introduces at least one new set of statements or a new condition (i.e., it traverses at least one new edge).

☞ V(G) is just an **upper bound** on the number of independent paths.

How to compute basis set?

- Very simple
  - Go from the starting node to the end node.
  - When the road forks, just take one direction.
  - Repeat the tour and take the other direction.
  - Remember the size of the set should be always equal or smaller than the cyclomatic number

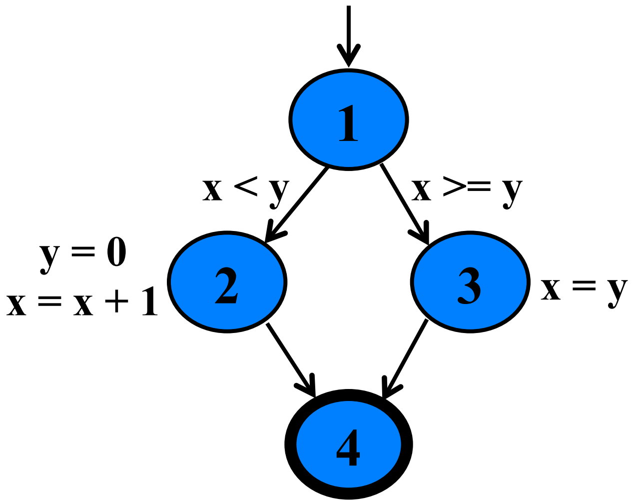
Control Flow Graphs

- 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)

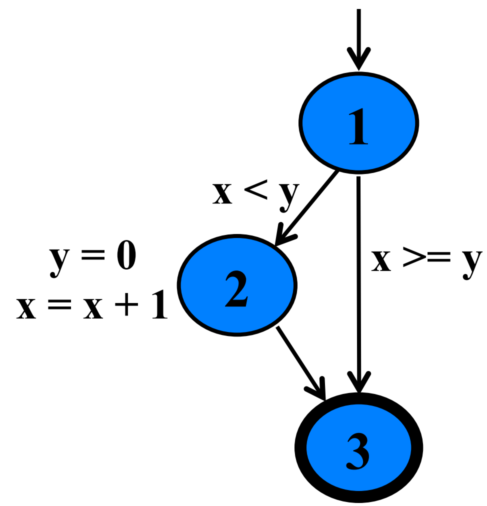
- CFGs are sometimes annotated with extra information
  - branch predicates
  - defs
  - uses
- Rules for translating statements into graphs …

CFG : The *if* Statement

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

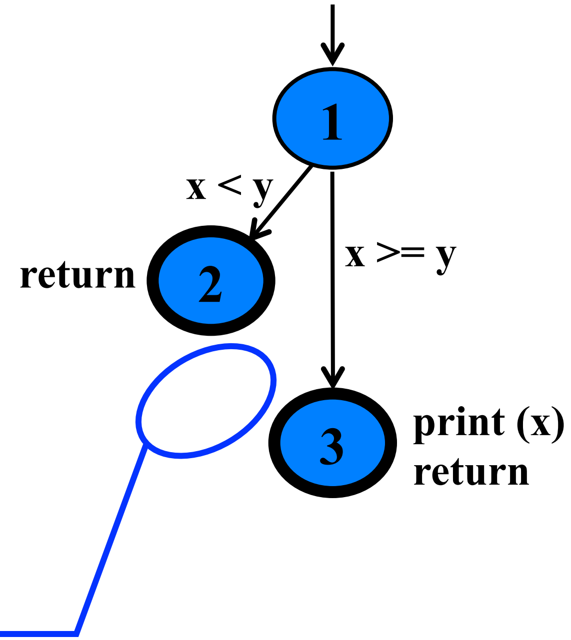



```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```



CFG : The *if-return* Statement

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



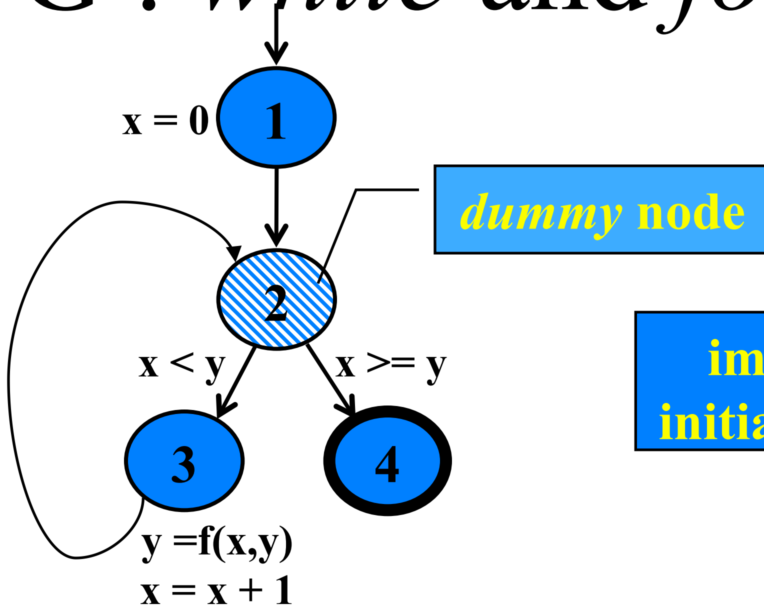
No edge from node 2 to 3.
The return nodes must be distinct.

Loops

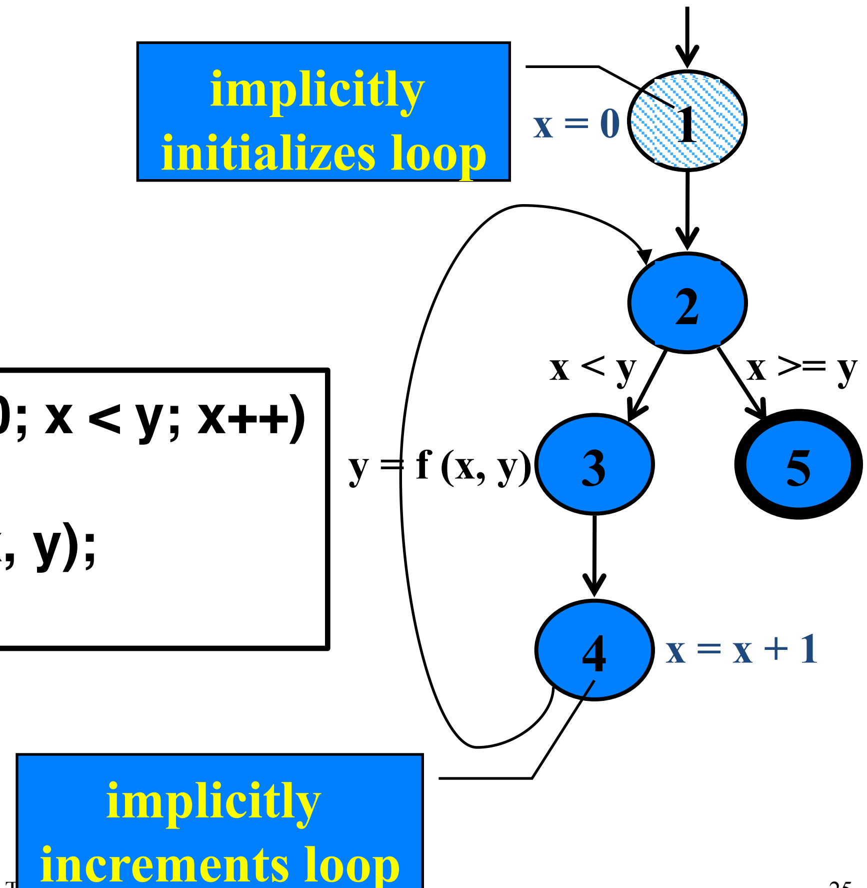
- Loops require "*extra*" nodes to be added
- Nodes that <u>do not</u> represent statements or basic blocks

CFG : *while* and *for* Loops

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

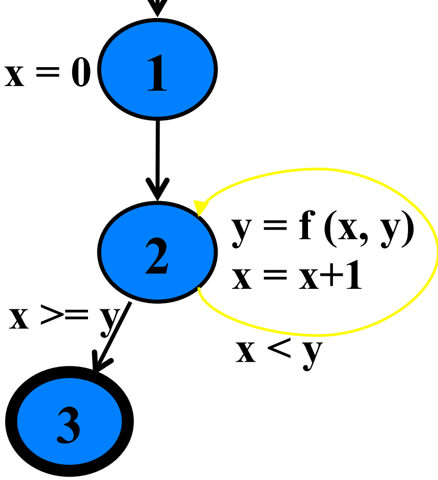



```
for (x = 0; x < y; x++)
{
   y = f (x, y);
}
```

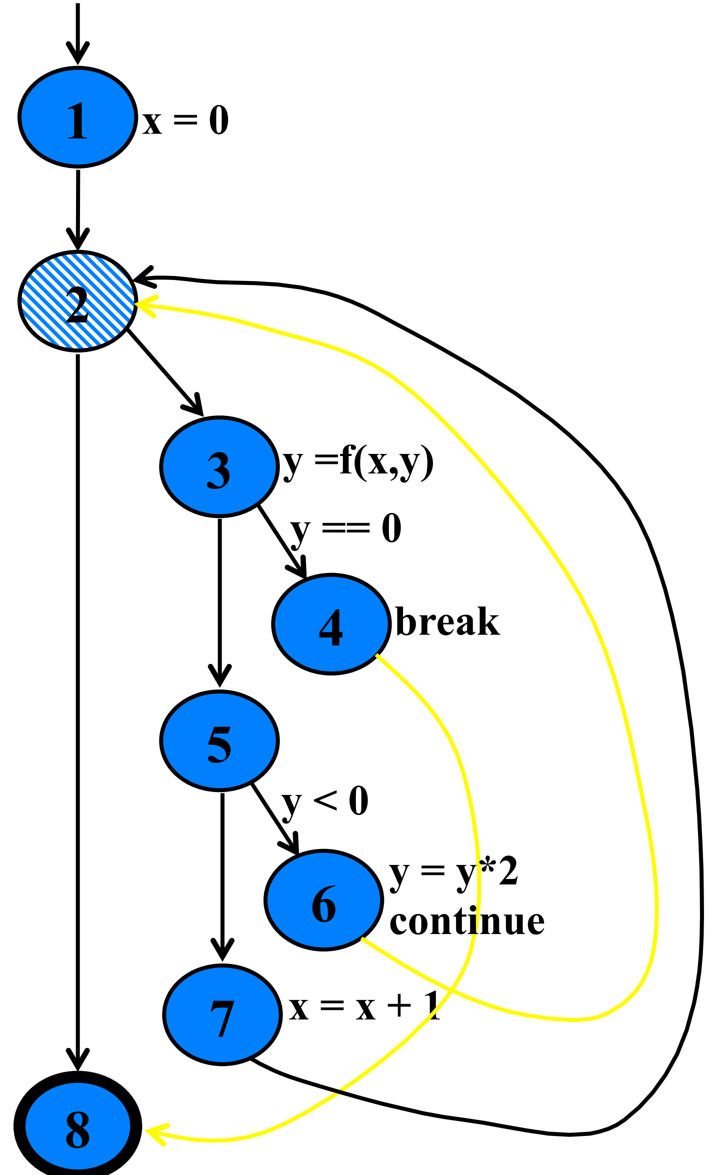


CFG : *do* Loop, *break* and *continue*

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

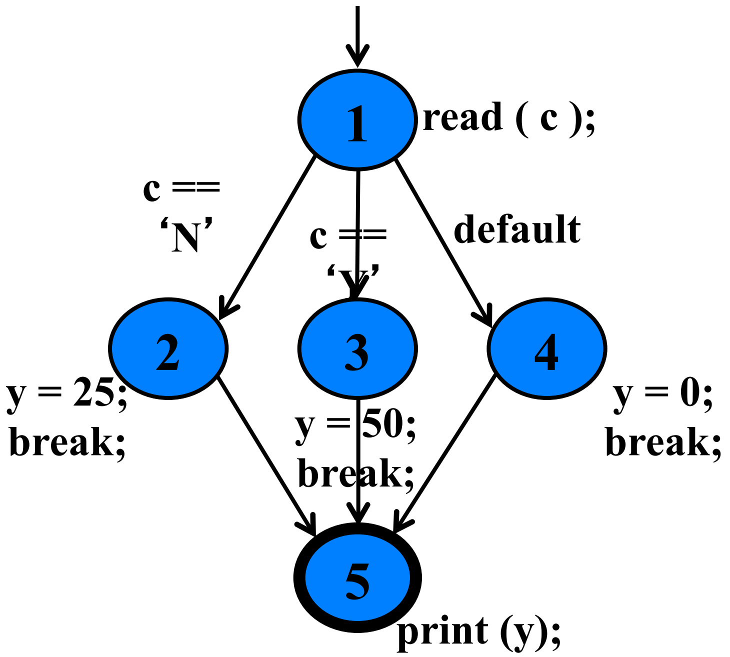



```
x = 0;
while (x < y)
{
   y = f (x, y);
   if (y == 0)
   {
      break;
   } else if y < 0)
   {
      y = y*2;
      continue;
   }
   x = x + 1;
}
print (y);
```



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);
```

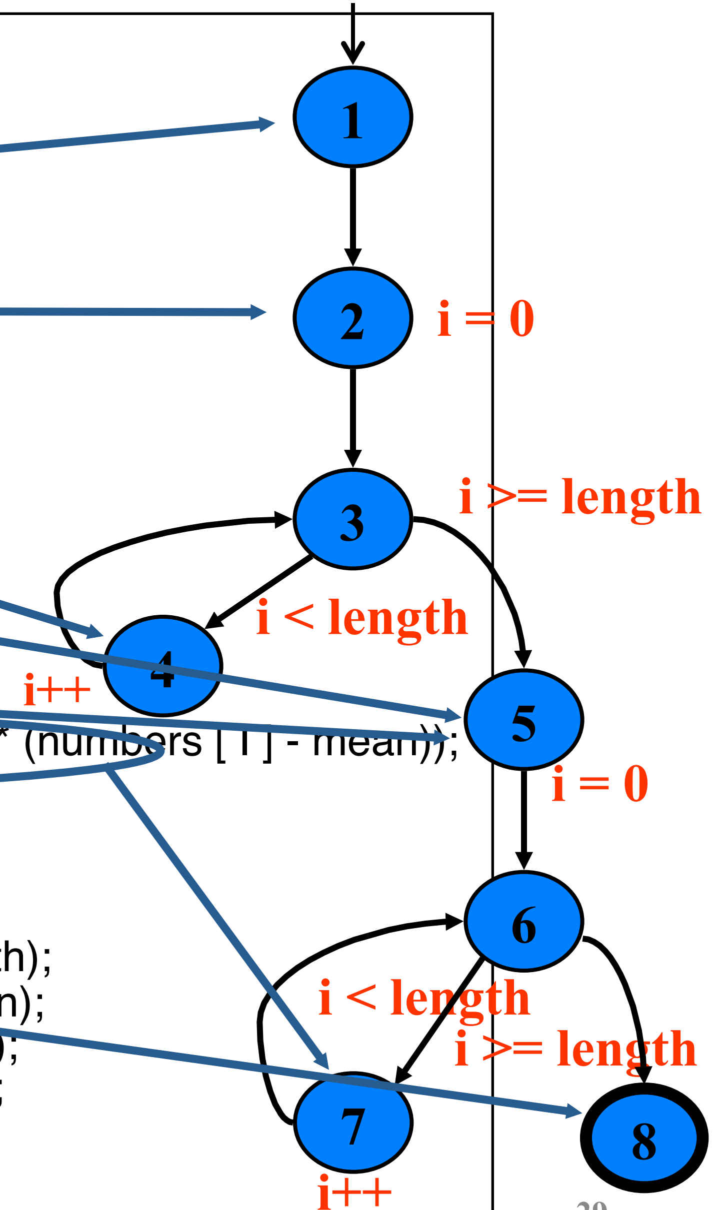


Example Control Flow – Stats

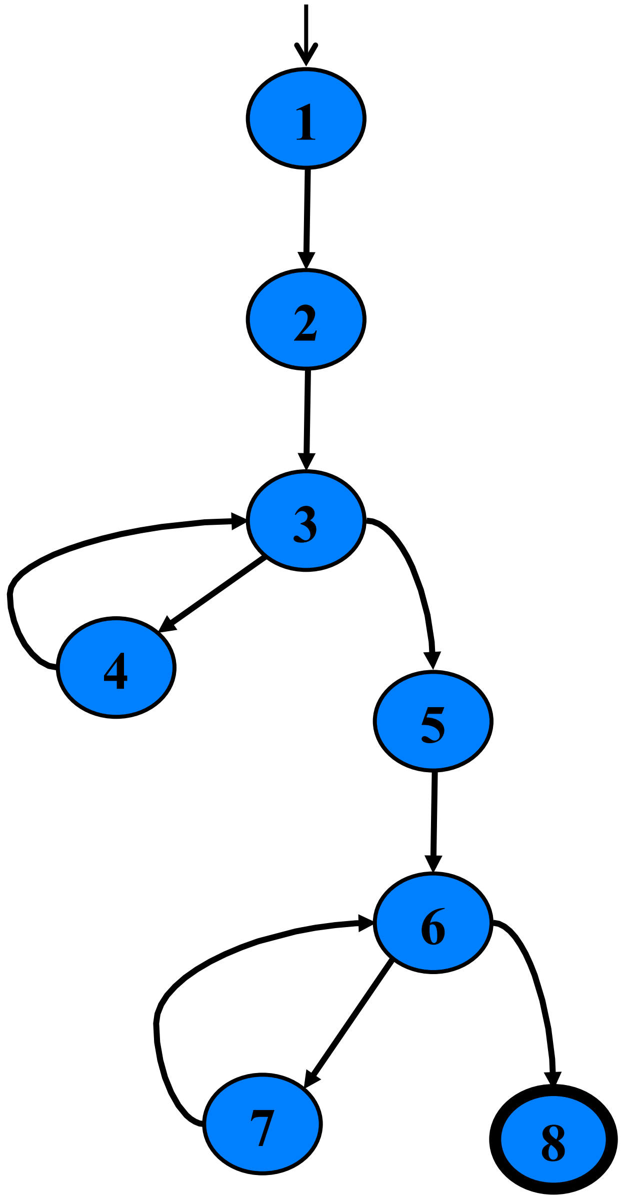
```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    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);
}
```

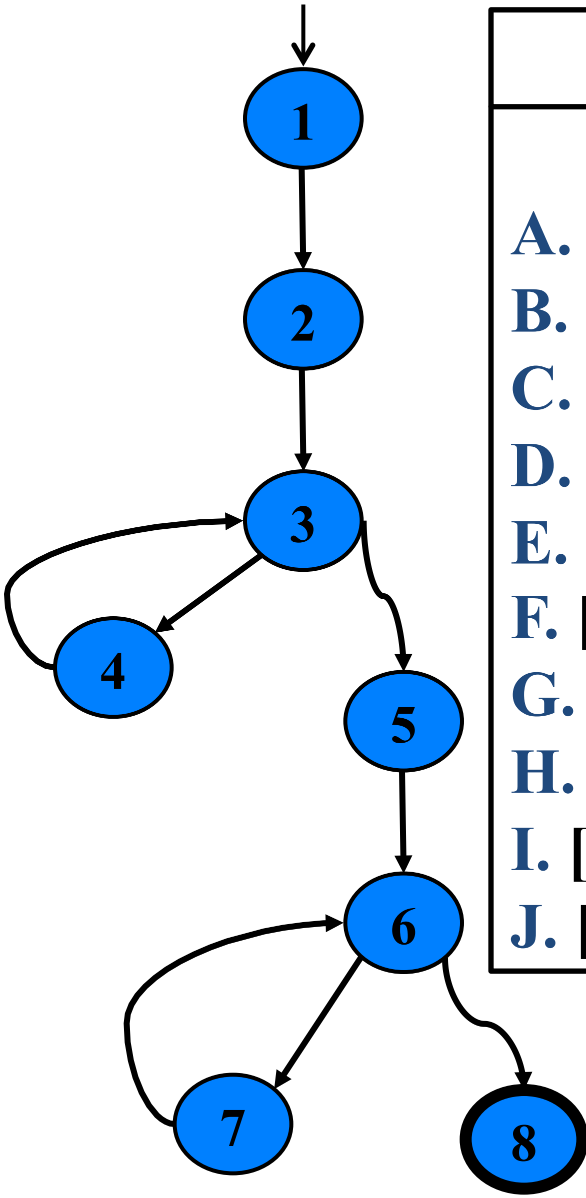


Control Flow TRs and Test Paths – EC



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 – PPC



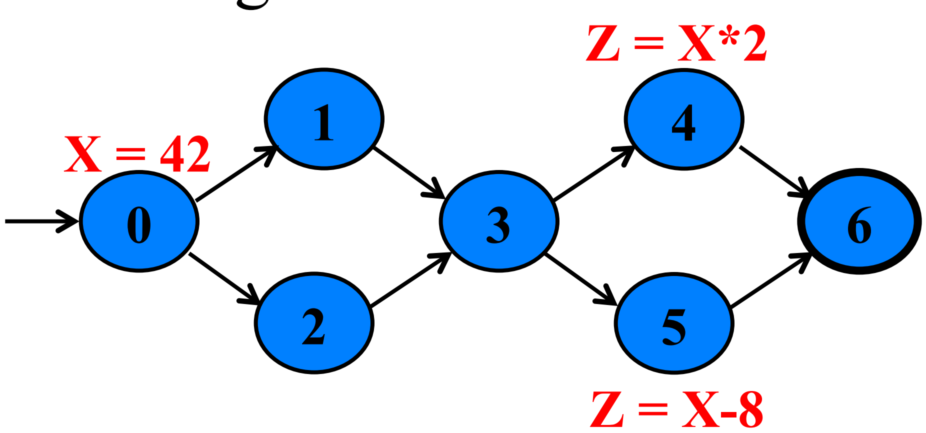
Prime Path Coverage

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. [3, 4, 5, 6, 7]	
H. [3, 4, 5, 6, 8]	
I. [1, 2, 3, 5, 6, 7]	
J. [1, 2, 3, 5, 6, 8]	

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*
- 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.

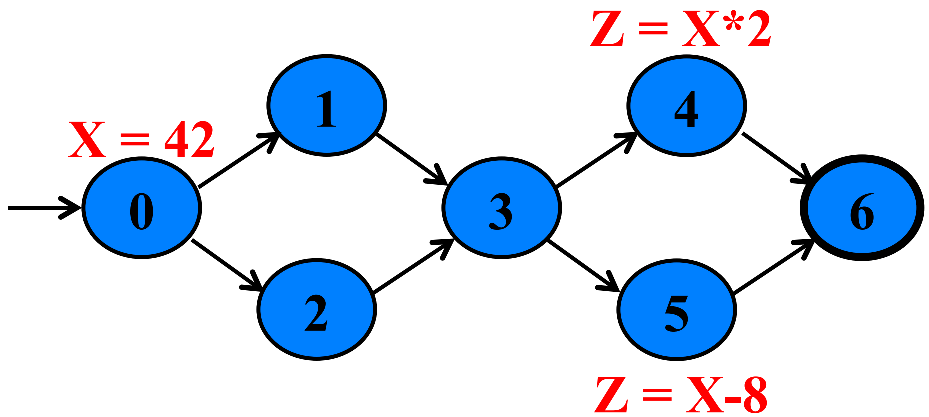
- **Then we make sure that every def reaches all possible uses**

<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]