Software Testing Technique
Chapter 5

Graph Coverage

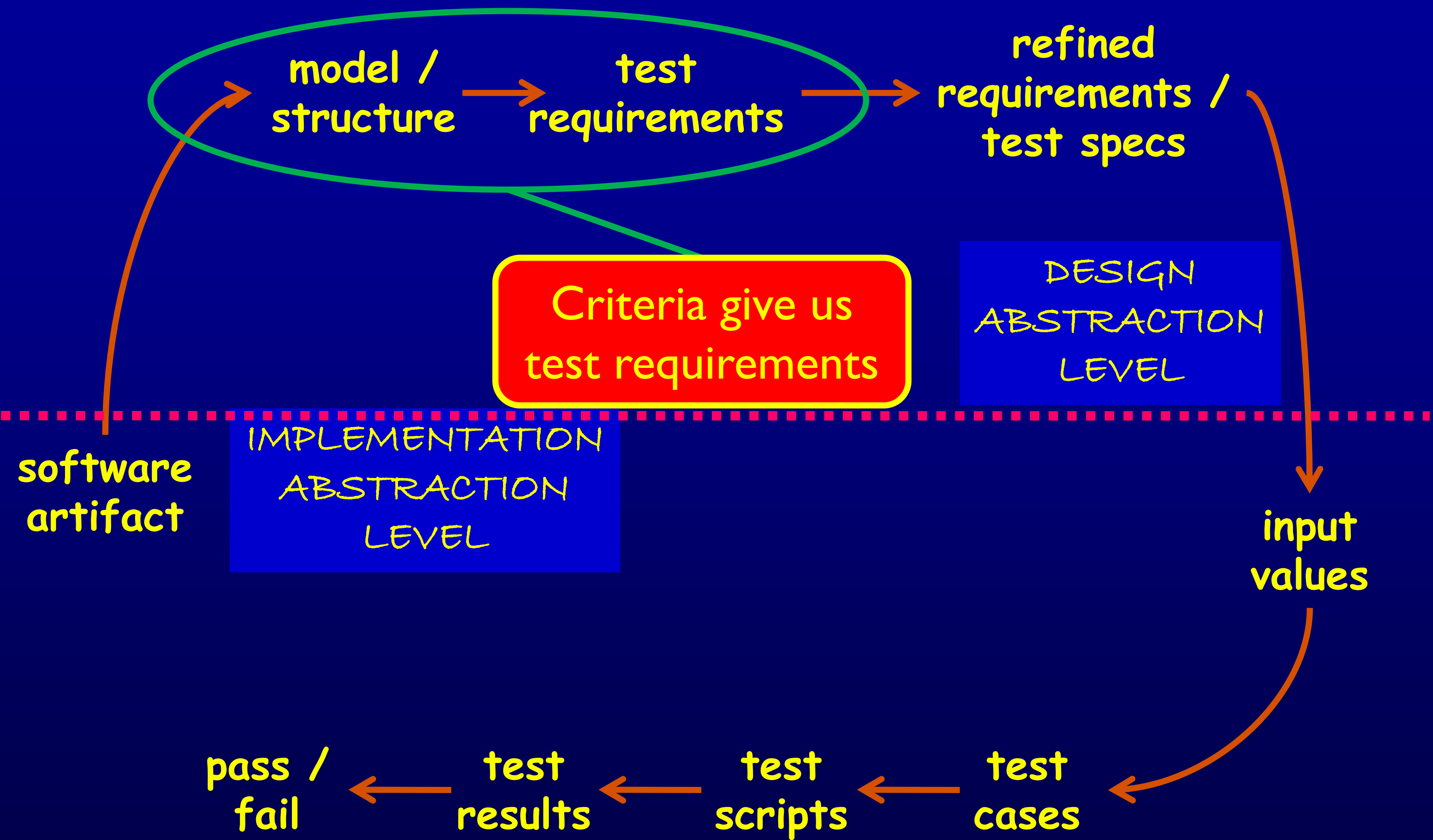
Zan Wang（王赞）
Office: 55-A413
Email: wangzan@tju.edu.cn

2022

Schedule

- **Session 9, Graph coverage. March 21**
- **Session 10, Logic coverage. March 23**
- **Session 11, Blackbox testing. March 28**
- **Session 12, Test Automation and Selenium. March 30**
- **Session 13, *Lab 3, Selenium I*. April 4**
- **Session 14, *Lab 4, Selenium II*. April 6**
- **Session 15, Load Testing and *Lab 5, Jmeter I*. April 11**
- **Session 16, *Lab 6, Jmeter2*. April 13.**

Model-Driven Test Design



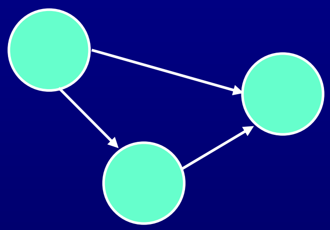
Criteria Based on Structures

Structures : Four ways to model software

1. Input Domain Characterization (sets)

A: {0, 1, >1}
B: {600, 700, 800}
C: {swe, cs, isa, infs}

2. Graphs



3. Logical Expressions

(not X or not Y) and A and B

4. Syntactic Structures (grammars)

```
if (x > y)
   z = x - y;
else
  z = 2 * x;
```

MORE DETAIL: INTRODUCTION TO SOFTWARE TESTING(J.OFFUTT) CHAPTER 7

Outline

- **Introduction to Graph Coverage**
- **Graph Coverage Criteria**
- **Graph Coverage for Source Code**
  - **Control Flow Graph Coverage**
  - **Data Flow Graph Coverage**

INTRODUCTION TO GRAPH COVERAGE

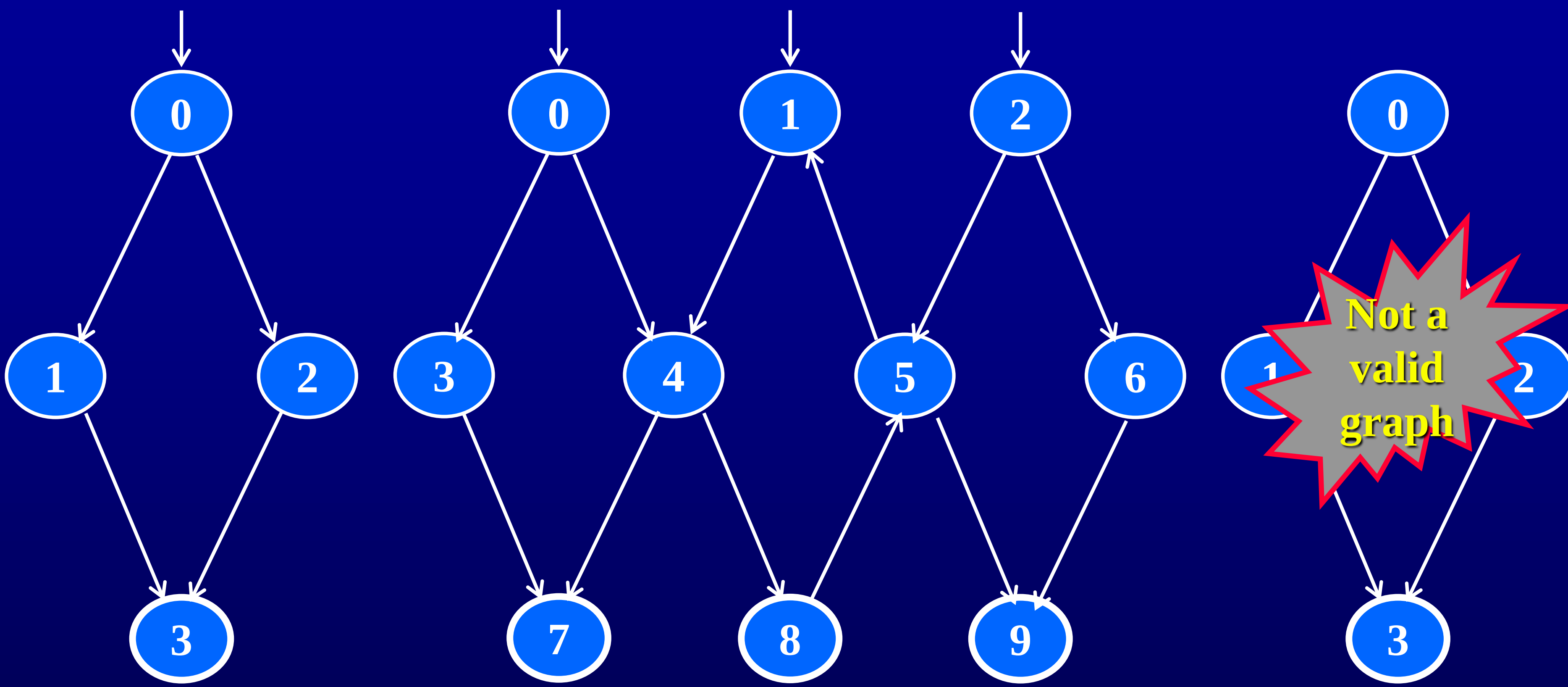
Covering Graphs

- Graphs are the most **commonly** used structure for testing
- Graphs can come from **many sources**
  - Control flow graphs
  - Design structure
  - FSMs and state charts
  - 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

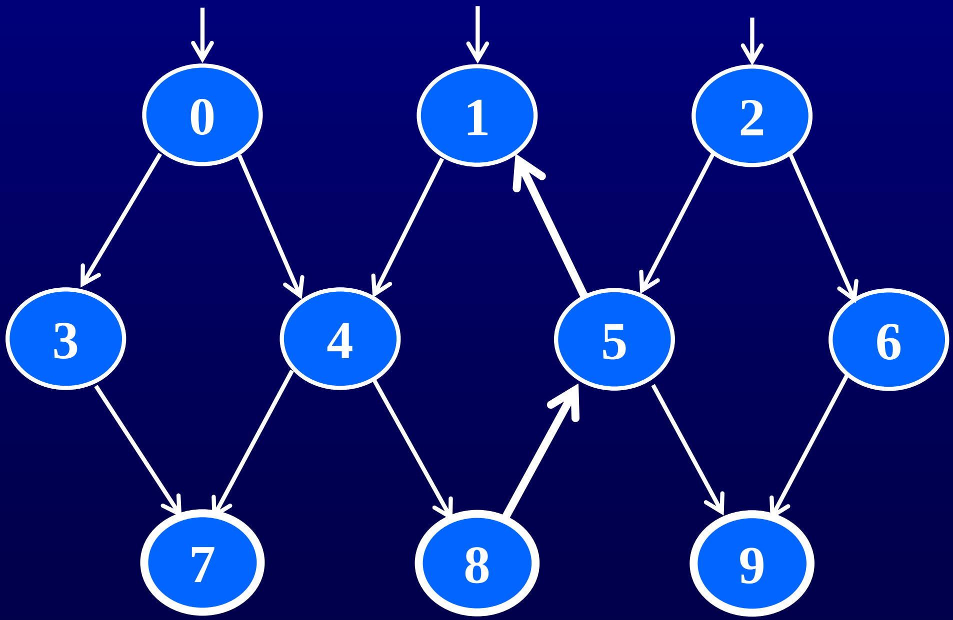


$N_0 = \{ 0 \}$ $N_0 = \{ 0, 1, 2 \}$ $N_0 = \{ \}$

$N_f = \{ 3 \}$ $N_f = \{ 7, 8, 9 \}$ $N_f = \{ 3 \}$

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*



A Few Paths

[0, 3, 7]

[1, 4, 8, 5, 1]

[2, 6, 9]

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

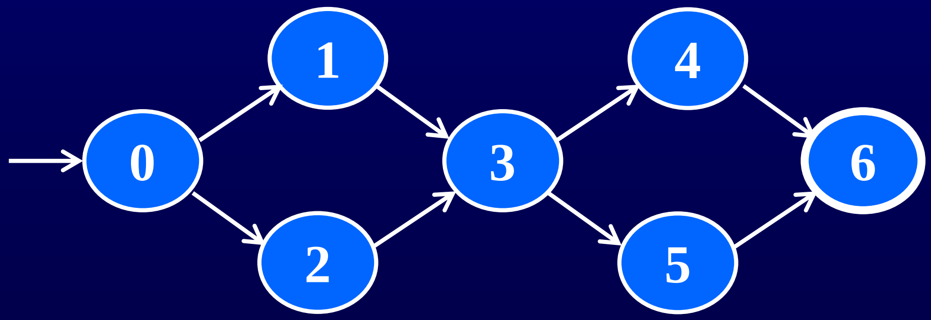
Reach ({0, 2}) = G

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 <u>any</u> tests

- **SESE graphs** : All test paths start at a single node and end at another node
  - Single-entry, single-exit
  - N_0 and N_f 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*

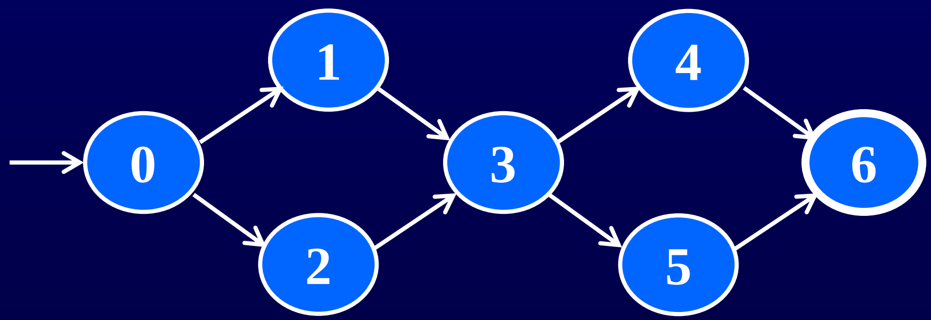
  A test path *p* ***visits*** edge *e* if *e* is in *p*
- **Tour** : A test path *p* ***tours*** subpath *q* if *q* is a subpath of *p*

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**

GRAPH COVERAGE CRITERIA

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**

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* if 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

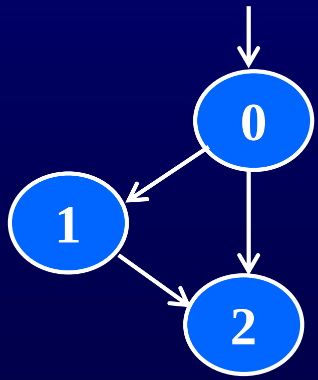
Node Coverage (NC) : TR contains each reachable node in G.

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 phrase "*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)**

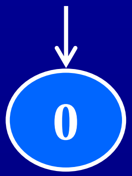


<u>Node Coverage</u> : TR = { 0, 1, 2 }
Test Path = [0, 1, 2]

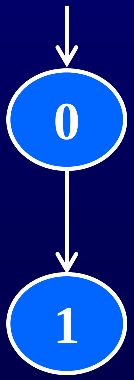
<u>Edge Coverage</u> : TR = { (0,1), (0, 2), (1, 2) }
Test Paths = [0, 1, 2]
[0, 2]

Paths of Length 1 and 0

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



- It may be seem trivial, 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 phrase "**length up to 2**" is used to include graphs that have less than 2 edges

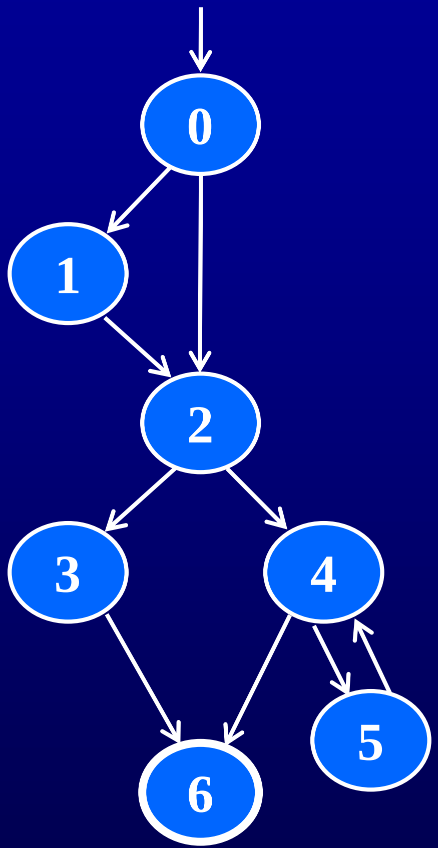
- The logical extension is to require **all paths** …

> **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

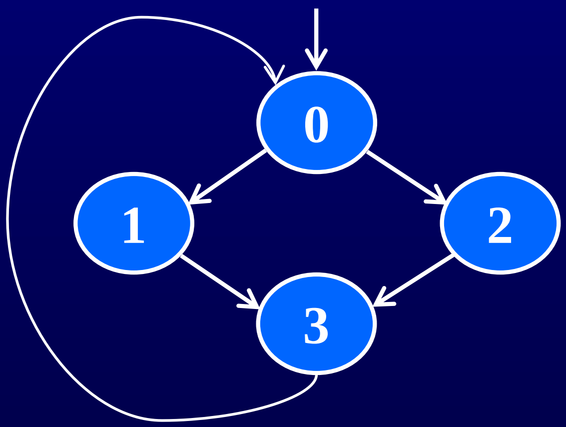
Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 6] [0, 1, 2, 4, 5, 4, 6] [0, 1, 2, 4, 5, 4, 5, 4, 6] [0, 1, 2, 4, 5, 4, 5, 4, 5, 4, 6] …

Loops in Graphs

- **If a graph contains a loop, it has an infinite number of paths**
- **Thus, CPC is not feasible**
- **SPC is not satisfactory because the results are subjective 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**

Simple Paths and Prime Paths

- **Simple Path** : ***A path from node n_i to n_j is simple if no node appears more than once, except possibly the first and last nodes are the same***
  - No internal loops
  - 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

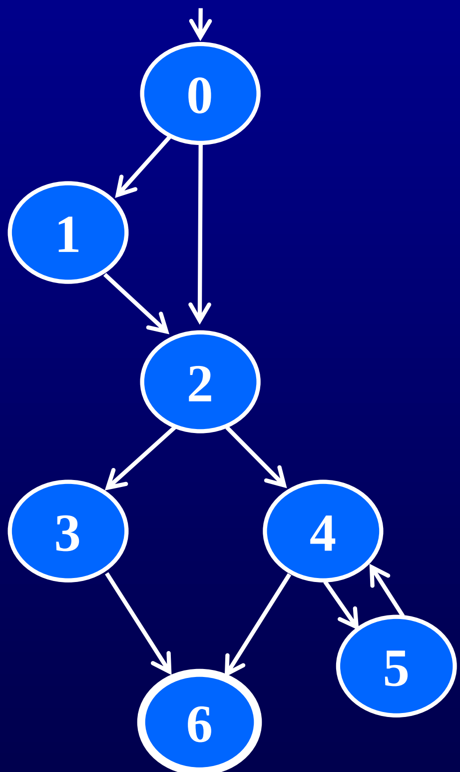
- A simple, elegant and finite criterion that requires loops to be executed as well as skipped

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
- Note : PPC does NOT subsume EPC
  - If a node *n* has an edge to itself, EPC will require [*n*, *n*, *m*]
  - [*n*, *n*, *m*] is not prime

Prime Path Example

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



<u>Prime Paths</u>

[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

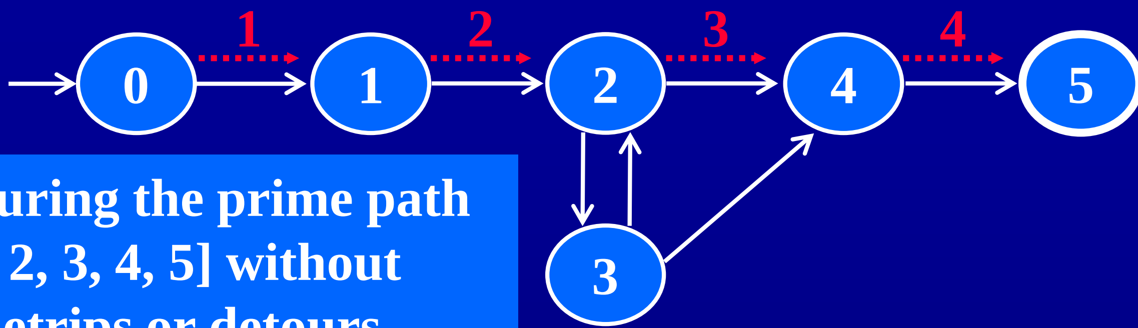
- **Prime paths do not have internal loops … test paths <u>might</u>**

- **<u>Tour</u> :** ***A test path p tours subpath q if q is a subpath of p***

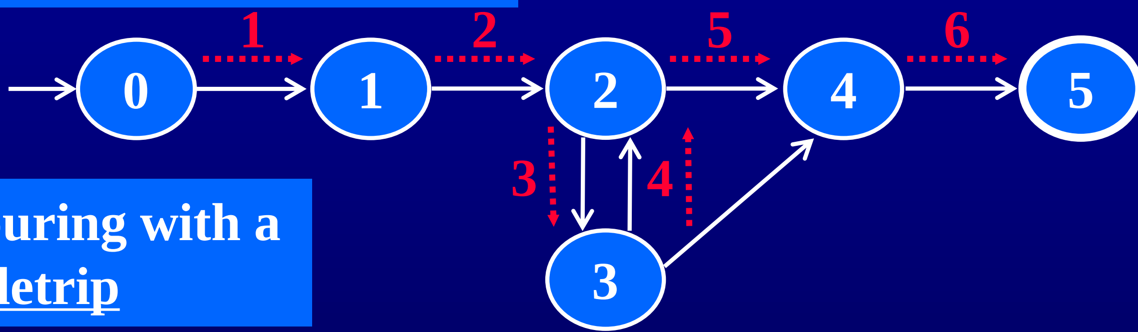
- **<u>Tour With Sidetrips</u> :** ***A test path p tours subpath q with <u>sidetrips</u> if every <u>edge</u> in q is also in p in <u>the same order</u>***
  - **The tour can include a sidetrip, as long as it comes back to the same node**

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

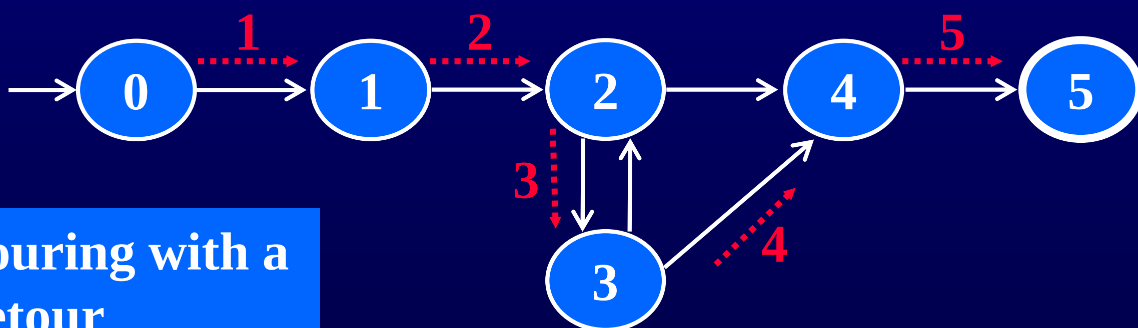
Sidetrips and Detours Example



Touring the prime path [1, 2, 3, 4, 5] without <u>sidetrips or detours</u>



Touring with a <u>sidetrip</u>



Touring with a <u>detour</u>

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 $X < 0$)

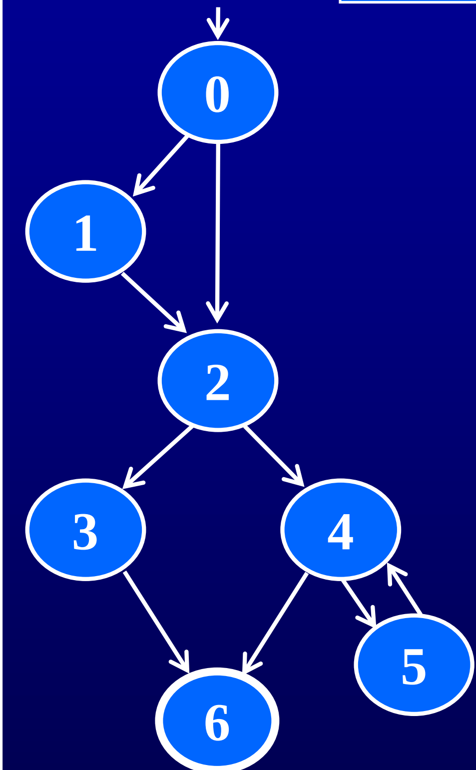
- Most test criteria have some infeasible test requirements
- It is usually undecidable whether all test requirements are feasible
- When sidetrips are not allowed, many structural criteria have more infeasible test requirements
- However, always allowing sidetrips weakens the test criteria

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

Simple paths



‘!’ means path
es

ns path
les

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] !

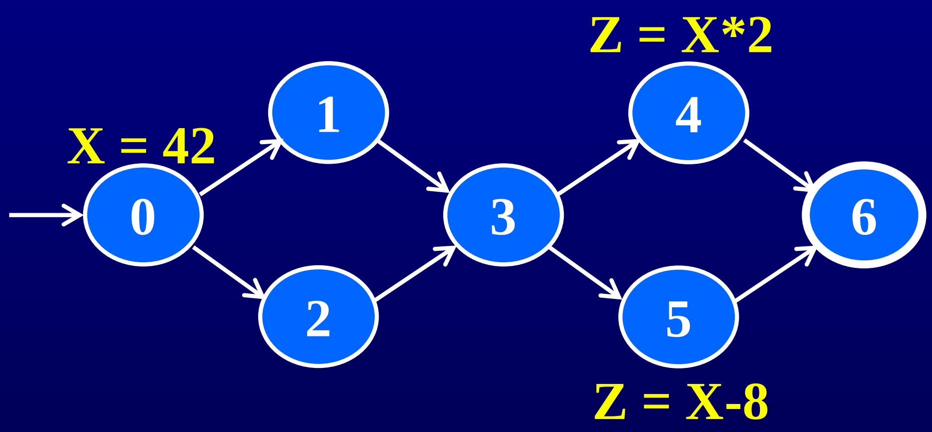
Len 4
[0, 1, 2, 3, 6] !
[0, 1, 2, 4, 6] !
[0, 1, 2, 4, 5] !

Prime Paths

Data Flow Criteria

Goal: Try to ensure that values are computed and used correctly

- **Definition (def) : A location where a value for a variable is stored into memory**
- **Use : A location where a variable's value is accessed**



Defs: def (0) = {X}
def (4) = {Z}
def (5) = {Z}
Uses: use (4) = {X}
use (5) = {X}

The values given in defs should reach at least one, some, or all possible uses

DU Pairs and DU Paths

- **def (n) or def (e) : The set of variables that are defined by node n or edge e**
- **use (n) or use (e) : The set of variables that are used by node n or edge e**

- **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.

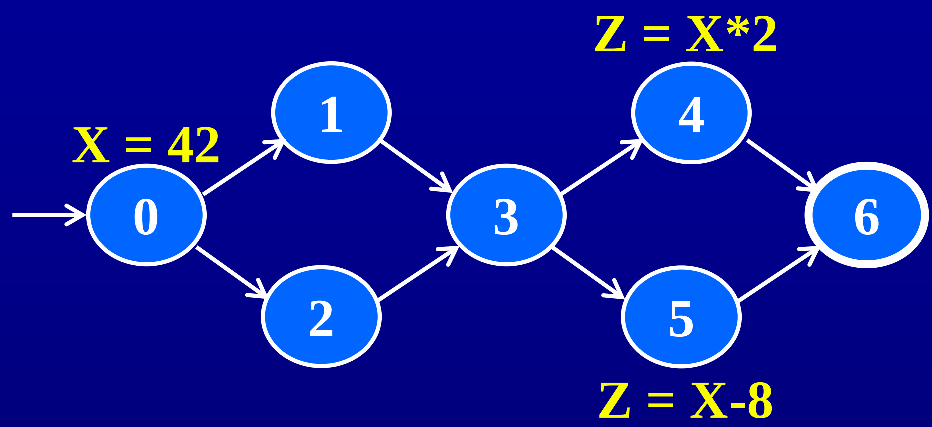
- **Then we make sure that every def reaches all possible uses**

<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

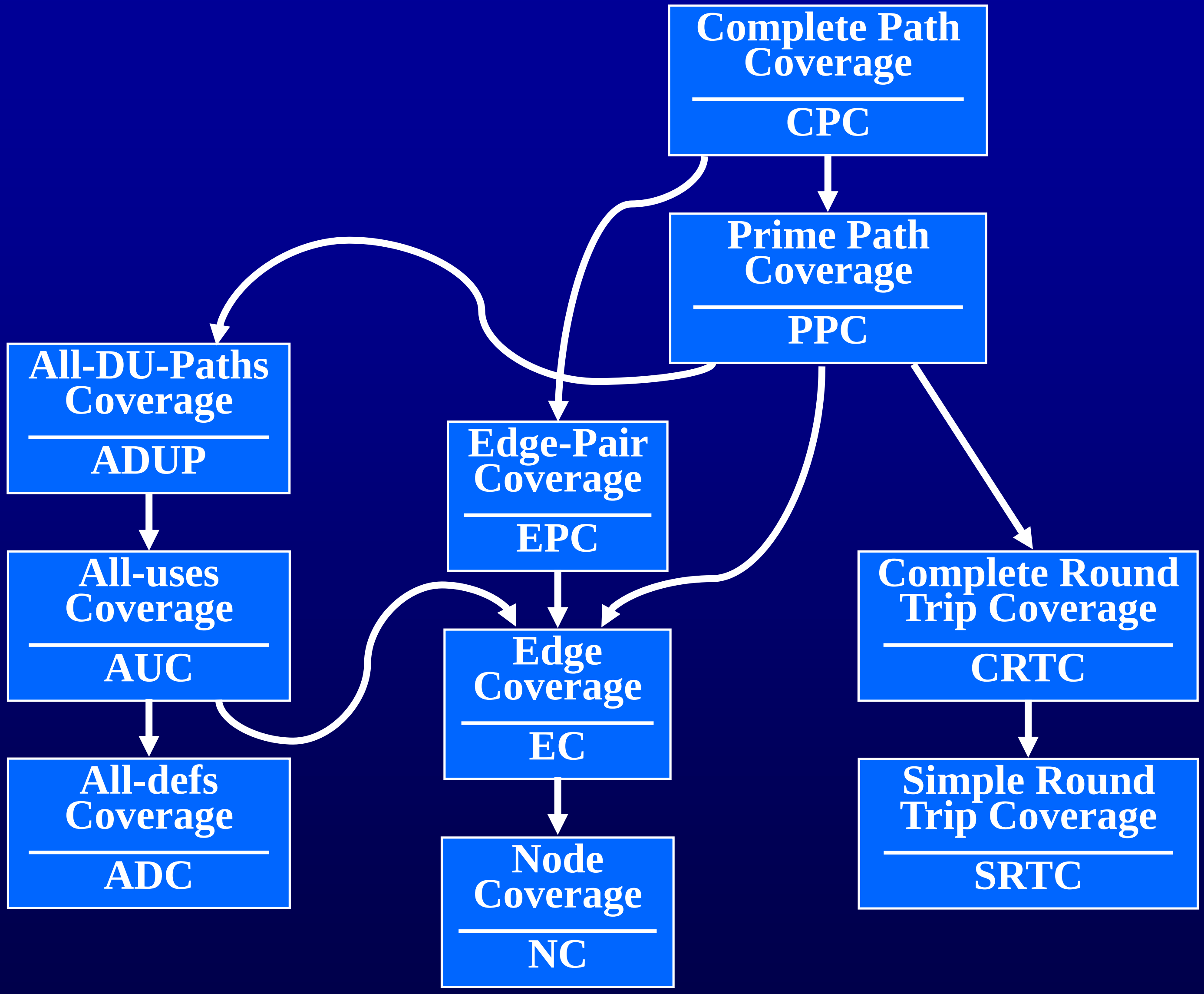


All-defs for *X*
[0, 1, 3, 4]

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



GRAPH COVERAGE FOR SOURCE CODE

CONTROL FLOW GRAPH COVERAGE

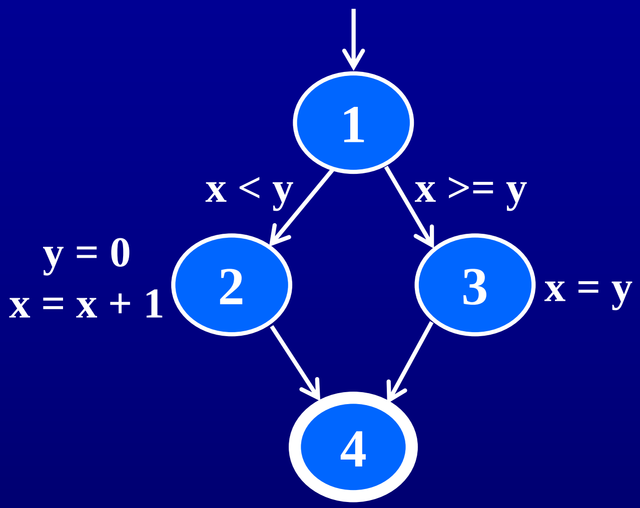
Control Flow Graphs

- 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)

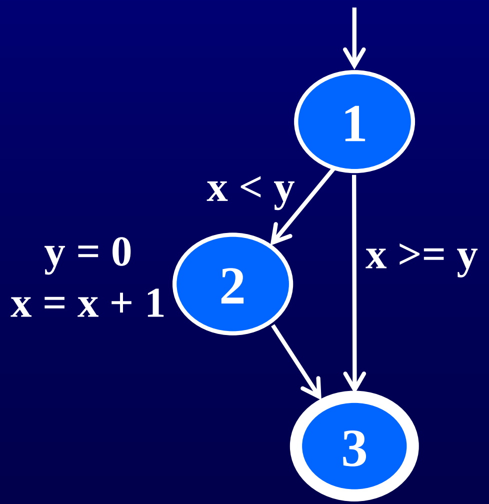
- CFGs are sometimes annotated with extra information
  - branch predicates
  - defs
  - uses
- Rules for translating statements into graphs …

CFG : The if Statement

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

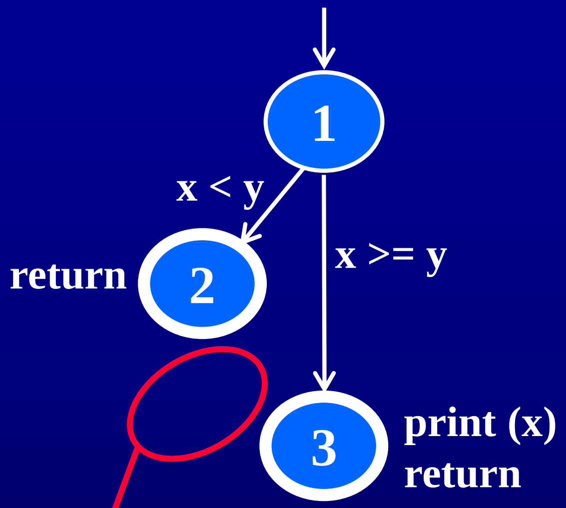



```
if (x < y)
{
  y = 0;
  x = x + 1;
}
```



CFG : The if-Return Statement

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



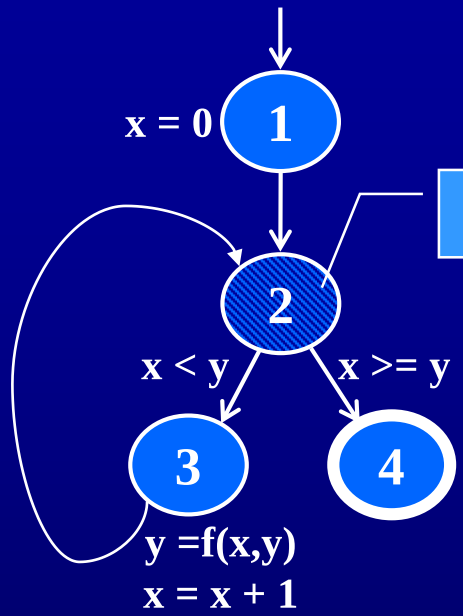
No edge from node 2 to 3.
The return nodes must be distinct.

Loops

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

CFG : while and for Loops

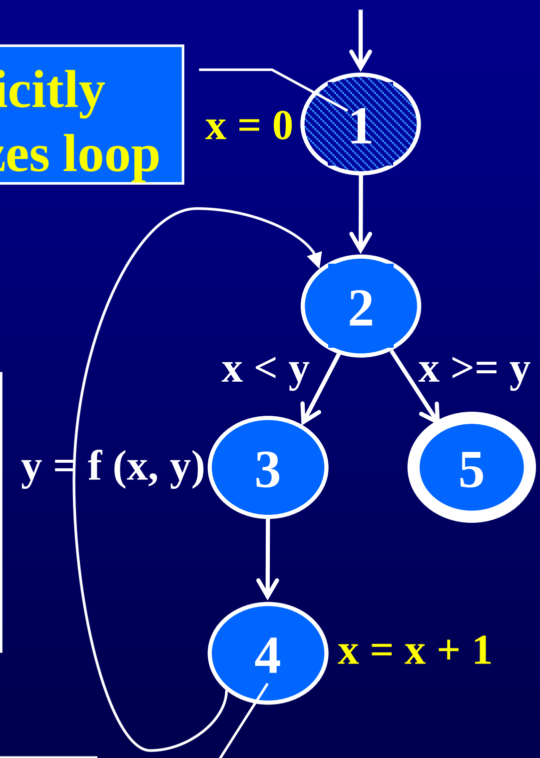
```
x = 0;
while (x < y)
{
   y = f (x, y);
   x = x + 1;
}
```



dummy **node**

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

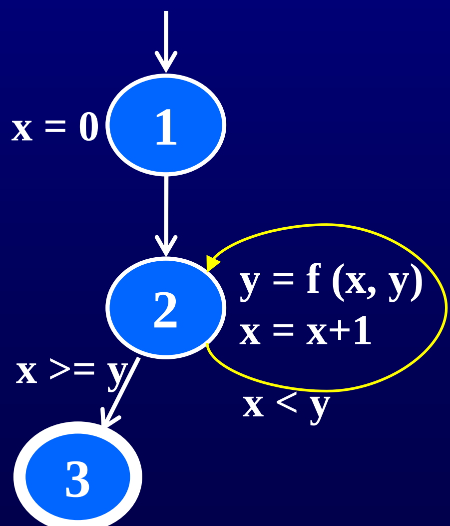
implicitly initializes loop



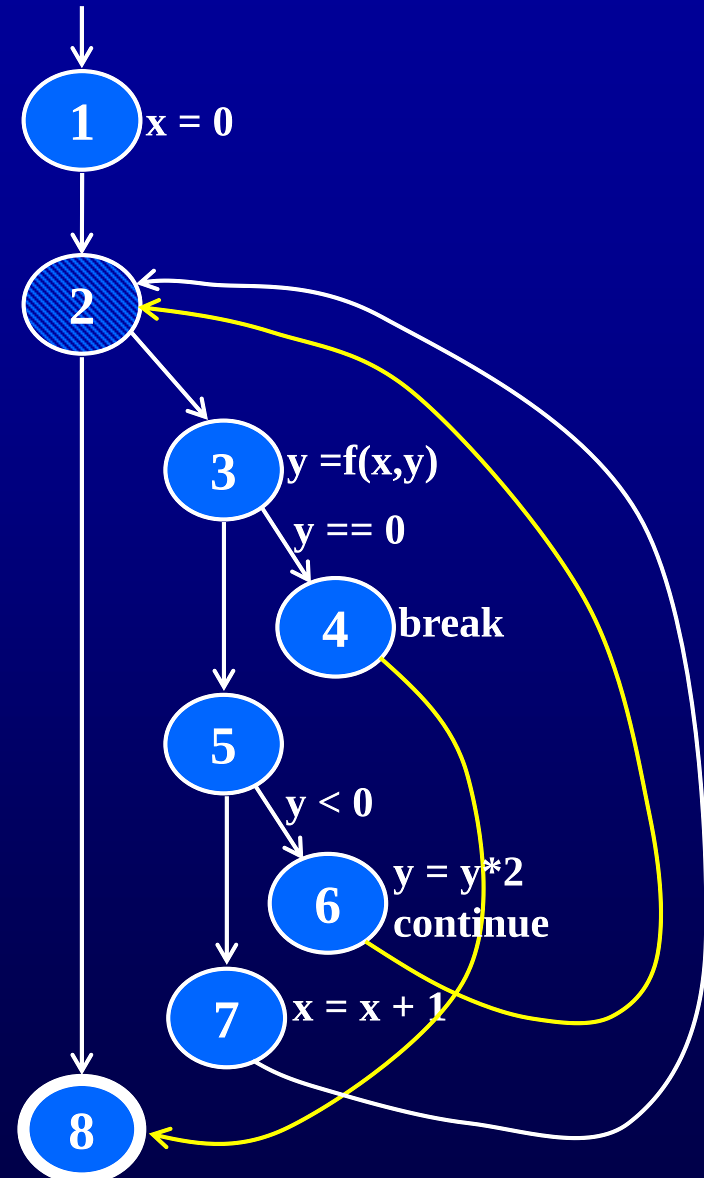
implicitly increments loop

CFG : do Loop, break and continue

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

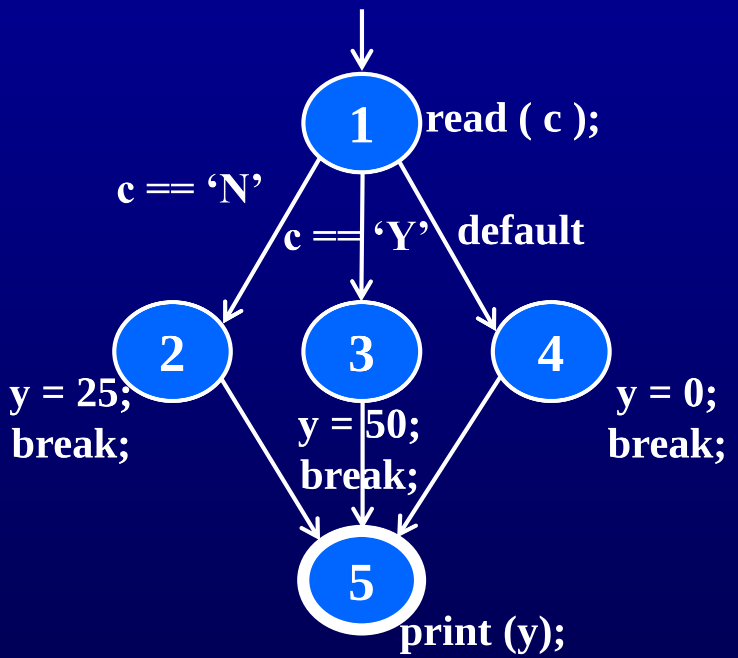



```
x = 0;
while (x < y)
{
  y = f (x, y);
  if (y == 0)
  {
    break;
  } else if (y < 0)
  {
    y = y*2;
    continue;
  }
  x = x + 1;
}
print (y);
```



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);
```

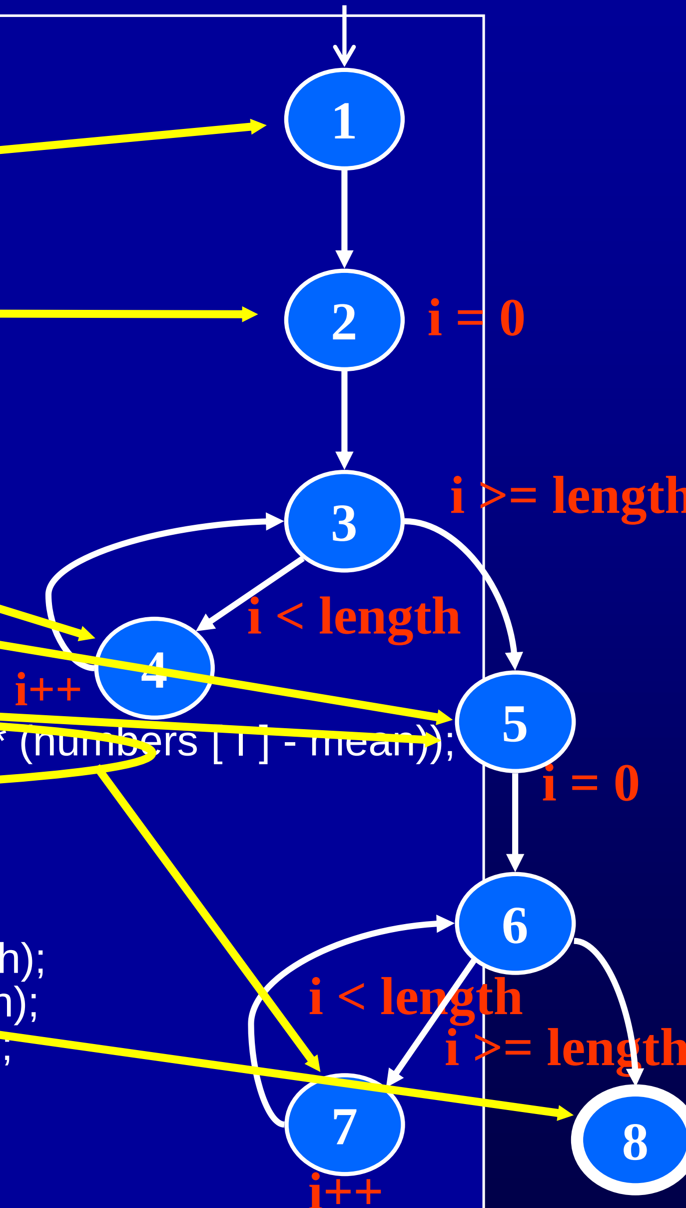


Example Control Flow – Stats

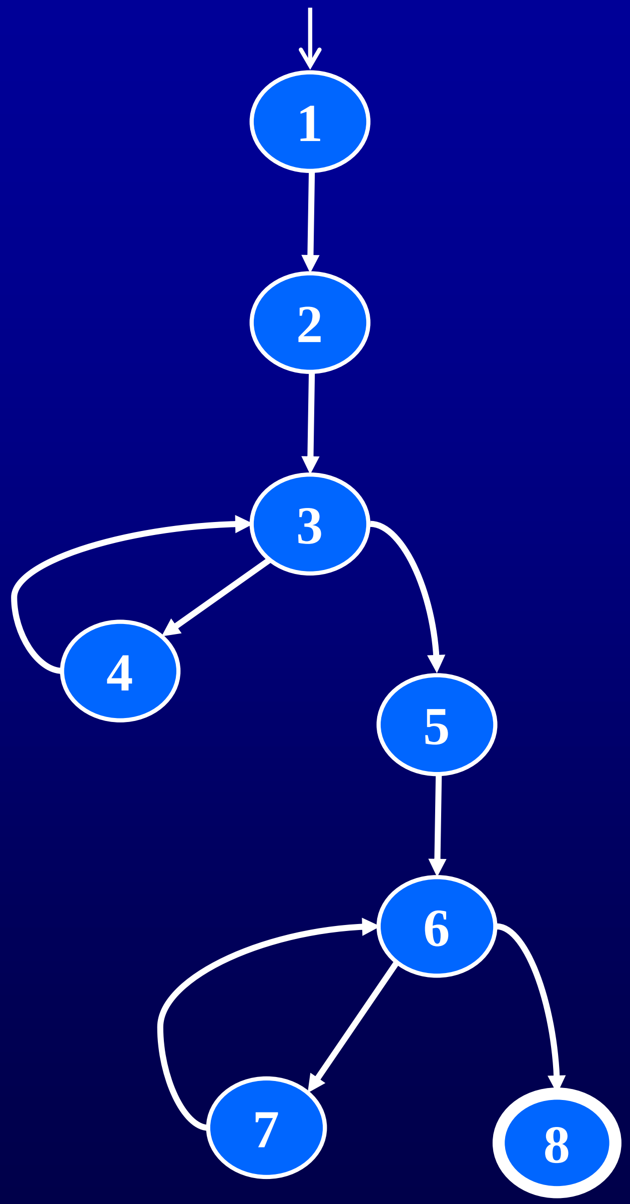
```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0.0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2];
    mean = sum / (double) length;

    varsum = 0.0;
    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 = 0.0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2];
    mean = sum / (double) length;

    varsum = 0.0;
    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);
}
```

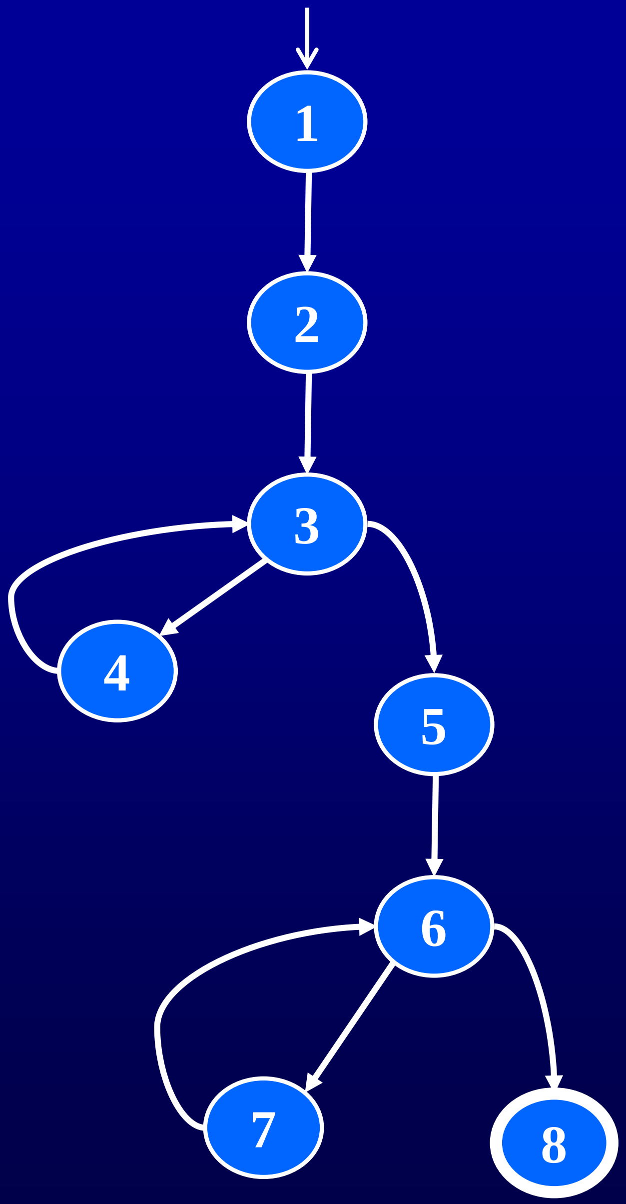


Control Flow TRs and Test Paths – EC



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]	

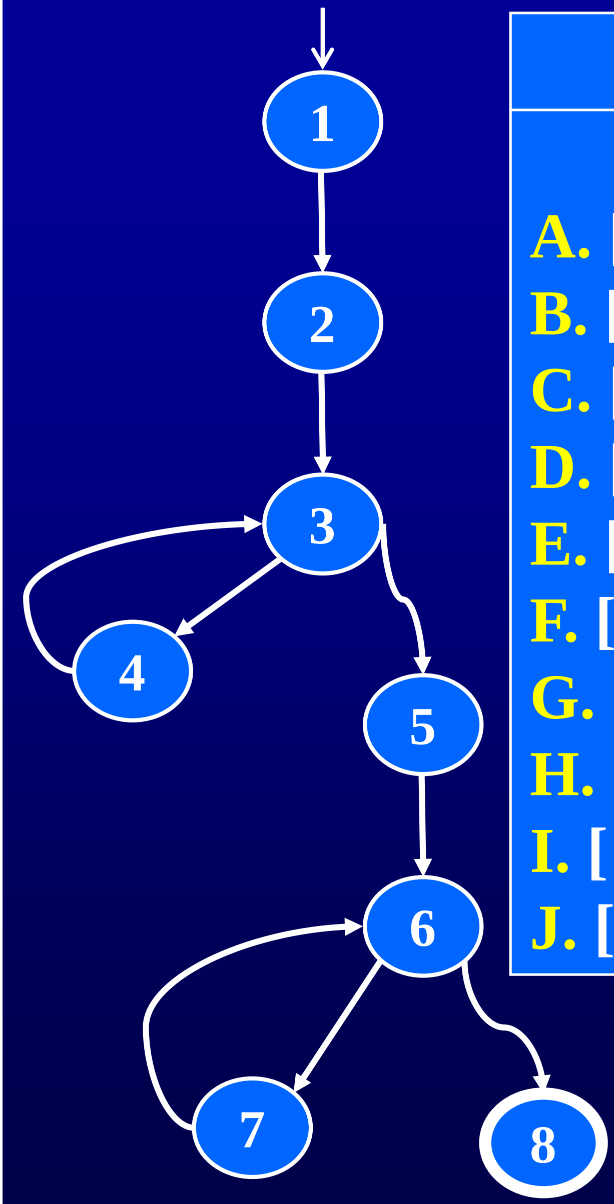
Control Flow TRs and Test Paths – EPC



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



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	

GRAPH COVERAGE FOR SOURCE CODE

DATAFLOW GRAPH COVERAGE

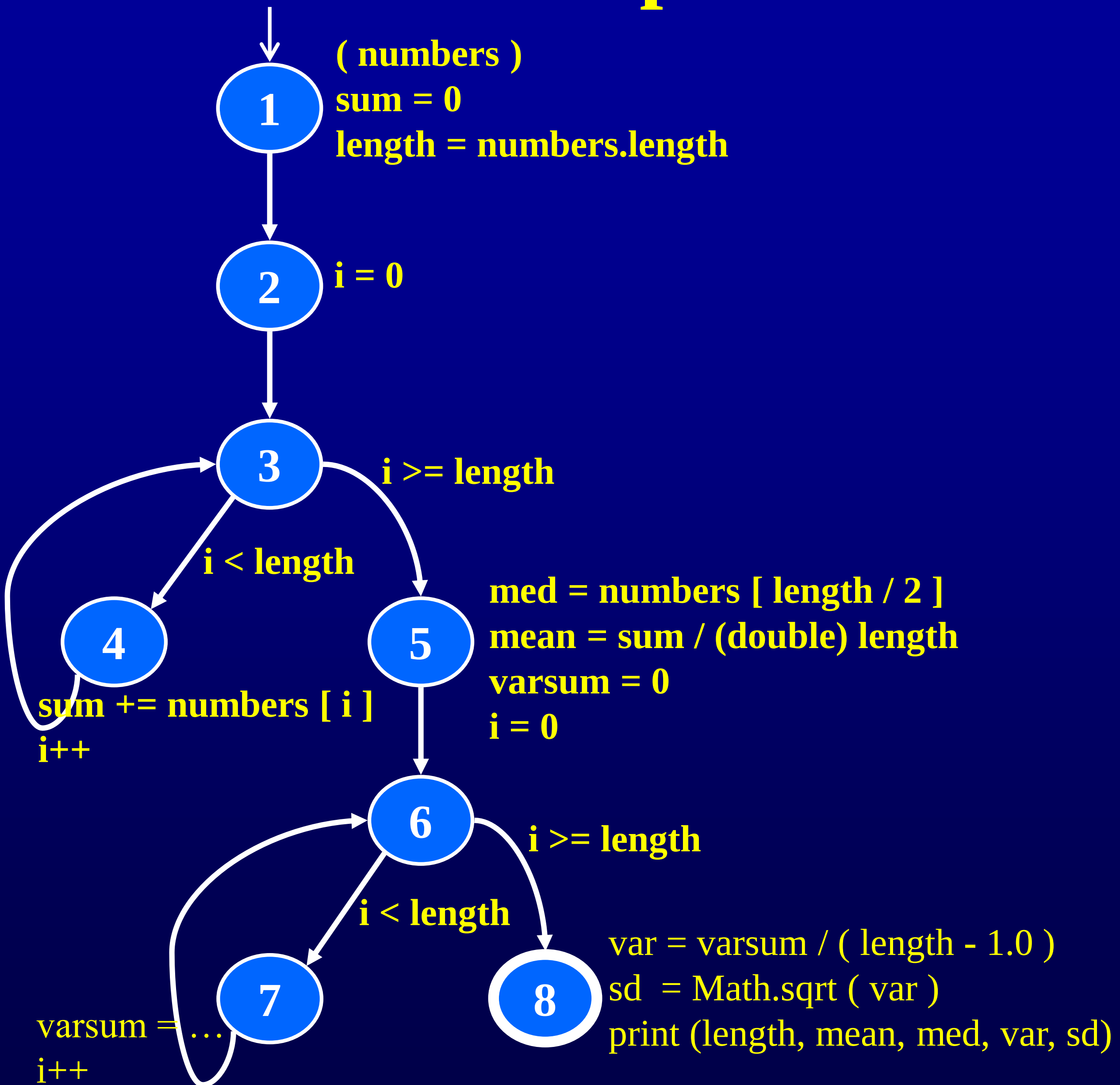
Data Flow Coverage for Source

- <u>def</u> : 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
- <u>use</u> : 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 <u>same node</u>, then it is only a DU-pair if the def occurs <u>after</u> the use and the node is in a loop

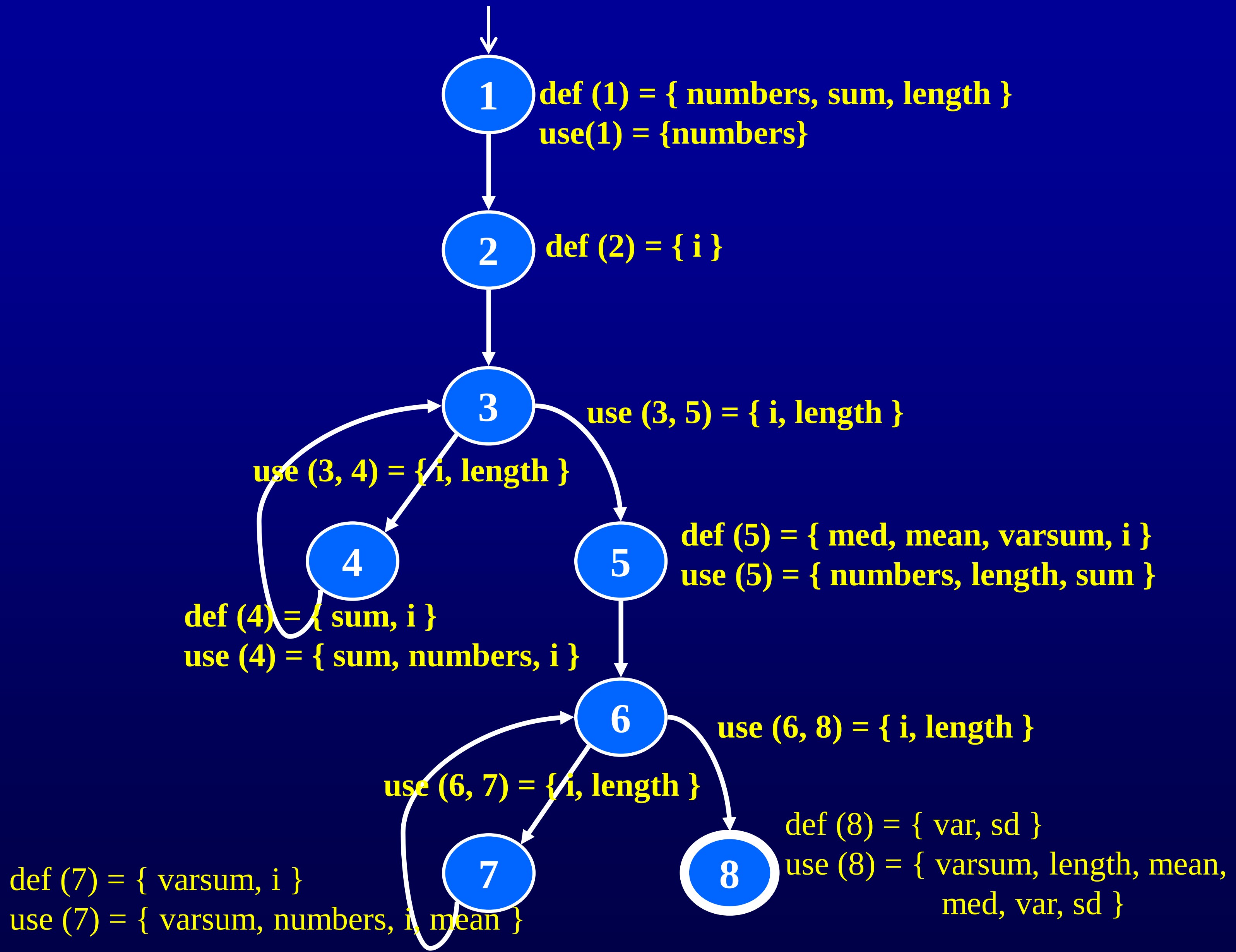
Example Data Flow – Stats

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

    sum = 0.0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2 ];
    mean = sum / (double) length;

    varsum = 0.0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1 );
    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



Defs and Uses Tables for Stats

Node	Def	Use
1	{ numbers, sum, length }	{ numbers }
2	{ i }	
3		
4	{ sum, i }	{ numbers, i, sum }
5	{ med, mean, 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 <u>before</u> uses, do not count as DU pairs

defs <u>after</u> 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 only 12 unique DU paths.

[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]

4 expect a loop not to be "entered"

6 require at least one iteration of a loop

2 require at least <u>two</u> iterations of a loop

Test Cases and Test Paths

Test Case : numbers = (44) ; length = 1

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 Case : numbers = (2, 10, 15) ; length = 3

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 index out of bounds exception…

med = numbers [length / 2];

mean = sum / (double) length;

A fault was found

Summary

- 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 and the book uses **basic blocks**
  - Coverage is the same, although the **bookkeeping** will differ

Homework 3

- 书上110页的图7-25的程序 patternIndex()
  - 画出控制流图
  - 找出图中的主路径
  - 找到测试用例，可以覆盖所有的主路径。