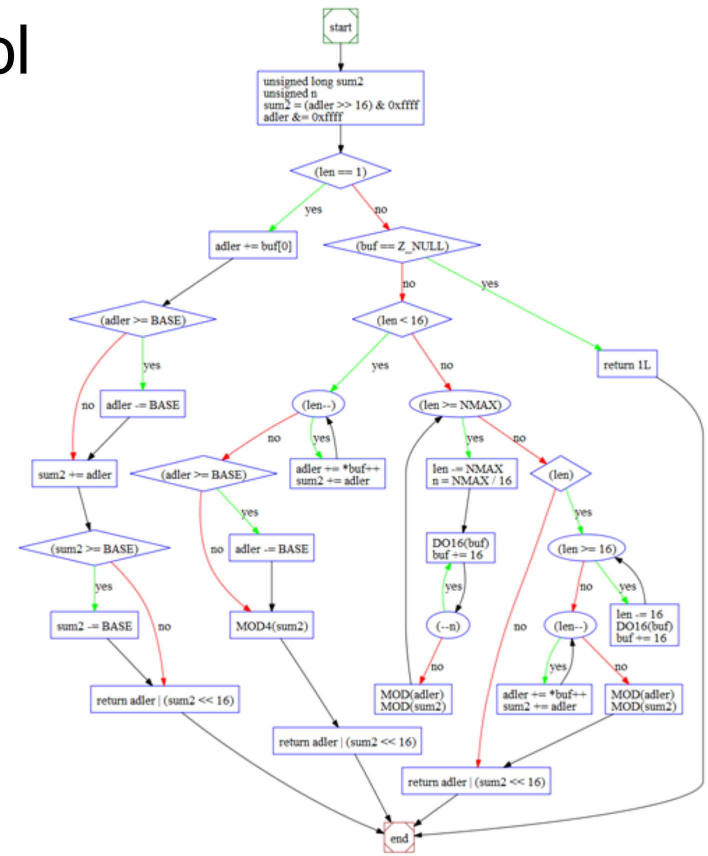
CS 575
Software Testing and Analysis

Control Flow Analysis

(c) Slides patially adopted from the slides of P. Amman & J. Offut
and of M. Pezze and M. Young

Control Flow

- Goal: Quantify flow of control in a program
  - sequencing of activities
- Basic control structures:
  - Sequence
  - Selection
  - Iteration
- Advanced control structures:
  - Procedure/function/agent call
  - Recursion (self-call)
  - Interrupt
  - Concurrence



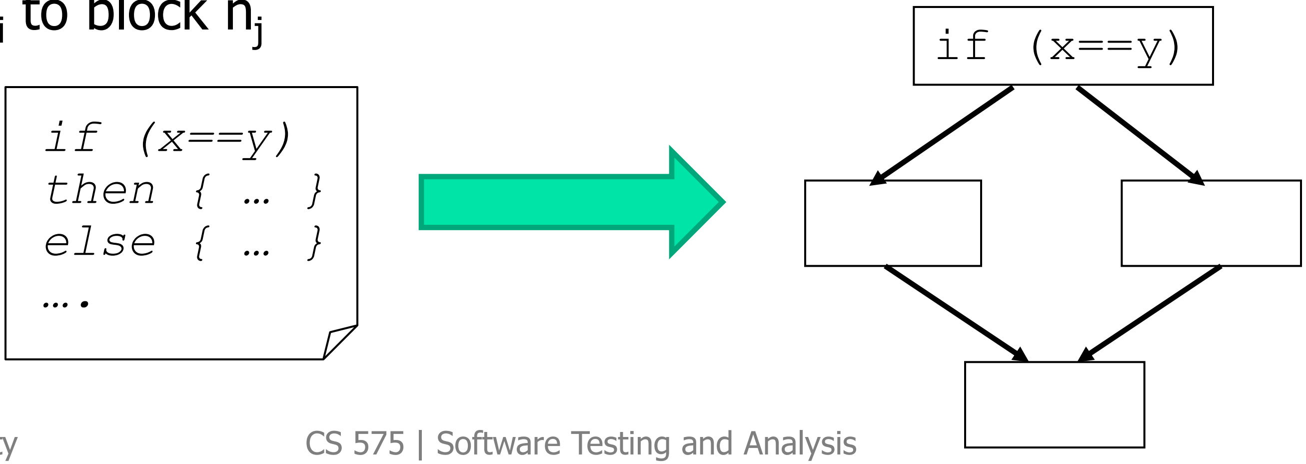
Control Flow Analysis

- **Control Flow** is a sequence of operations represented by:
  - Control flow graph
  - Control dependence graph
  - Call graph

- **Control Flow Analysis:** analyzing a program to discover its control structure

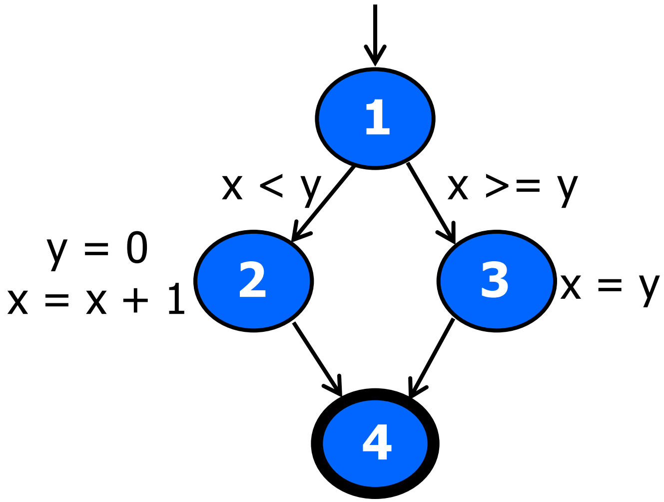
Basic Control Flow Graph

- Models flow of control in the program
- CFG = (N, E) is a directed graph
  - Node n ∈ N: basic blocks, i.e., a maximal sequence of statements with a single entry point and single exit point (no internal branches)
  - Edge e = $(n_i, n_j) \in E$: possible transfer of control from block n_i to block n_j

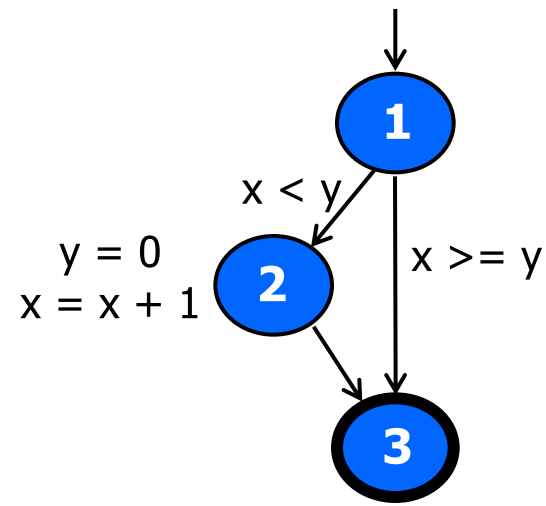


CFG: The if Statement

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

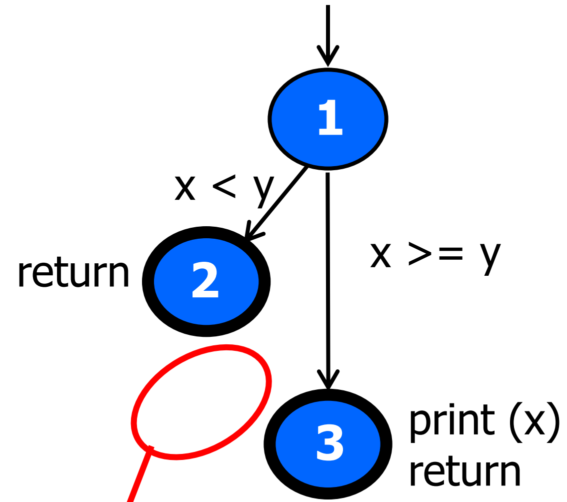



```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```



CFG: The if-Return Statement

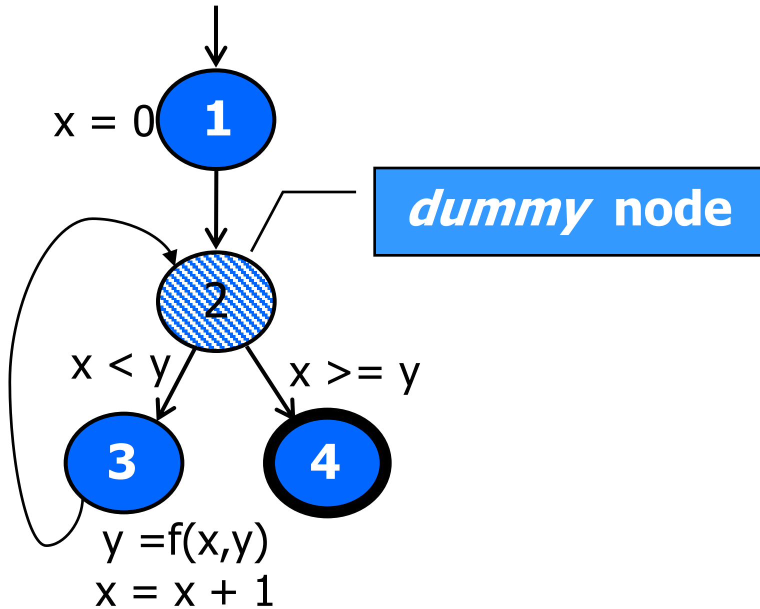
```
if (x < y)
{
   return;
}
print (x);
return;
```



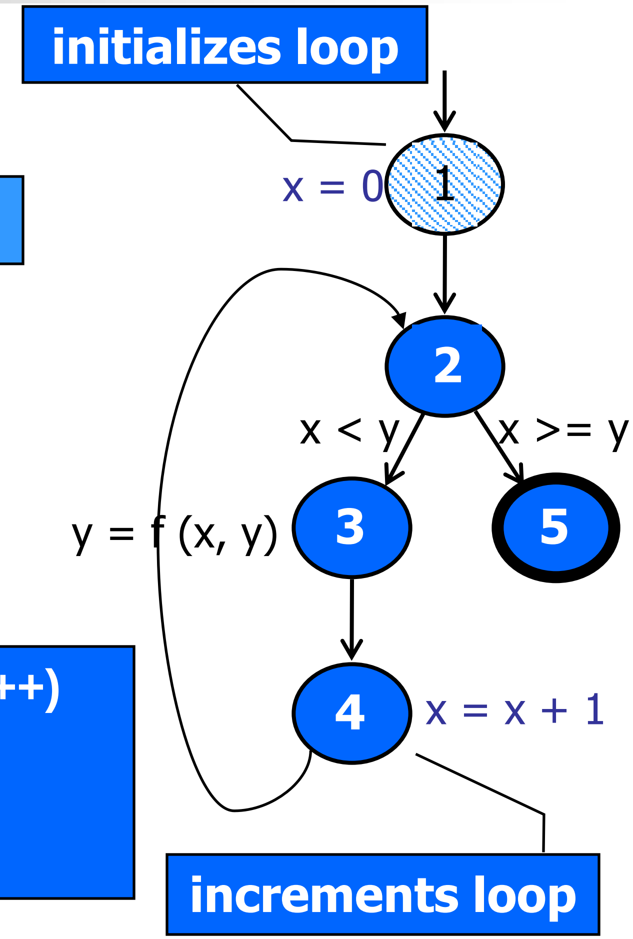
No edge from node 2 to 3.
The return nodes must be distinct.

CFG : while and for Loops

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

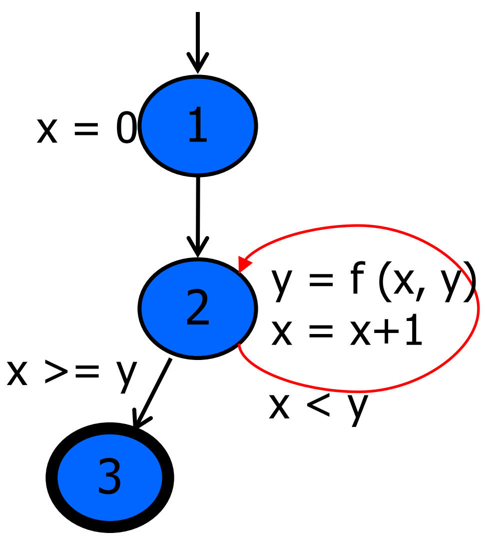



```
for (x = 0; x < y; x++)
{
   y = f (x, y);
}
```

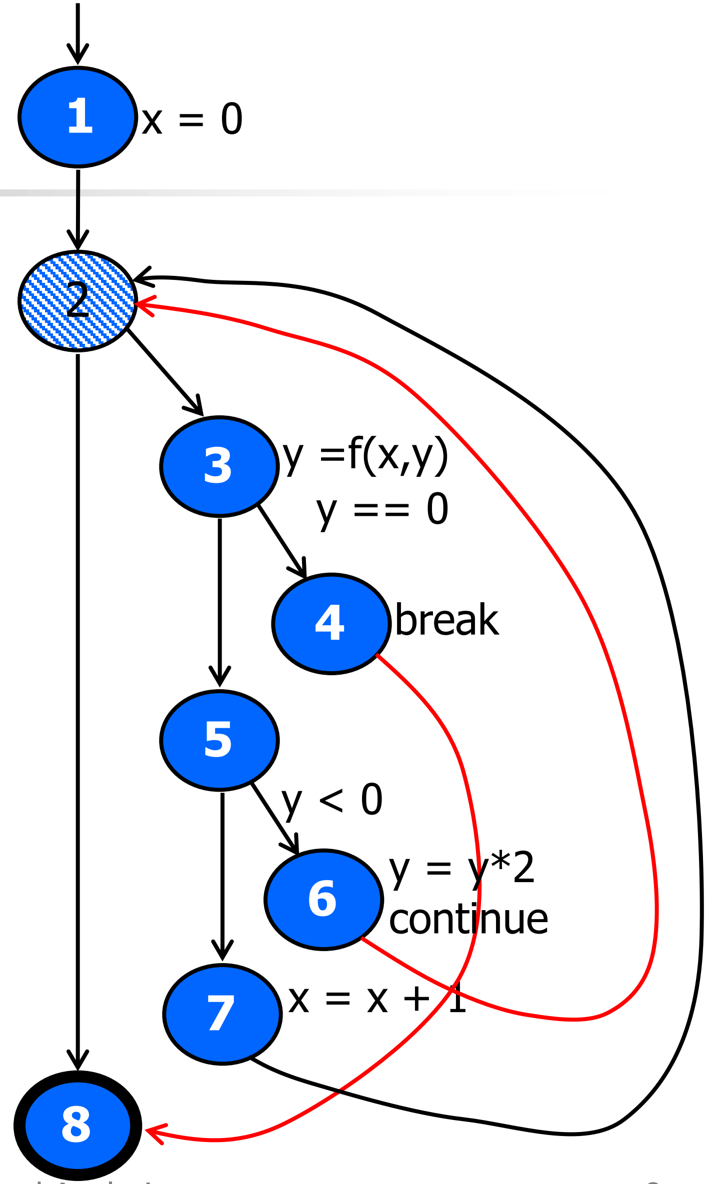


CFG: do, break and continue

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

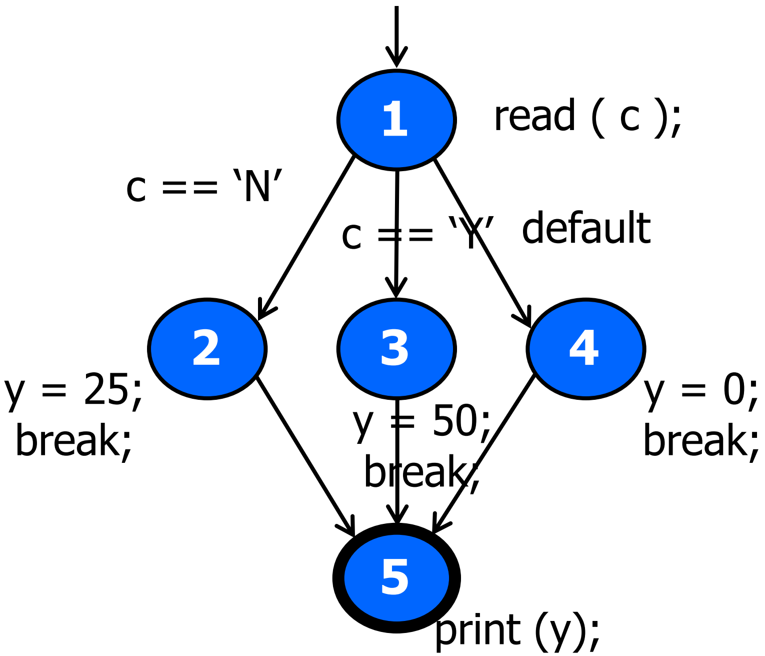



```
x = 0;
while (x < y)
{
   y = f (x, y);
   if (y == 0)
   {
      break;
   } else if (y < 0)
   {
      y = y*2;
      continue;
   }
   x = x + 1;
}
print (y);
```



CFG: case (switch)

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



Nodes in CFG

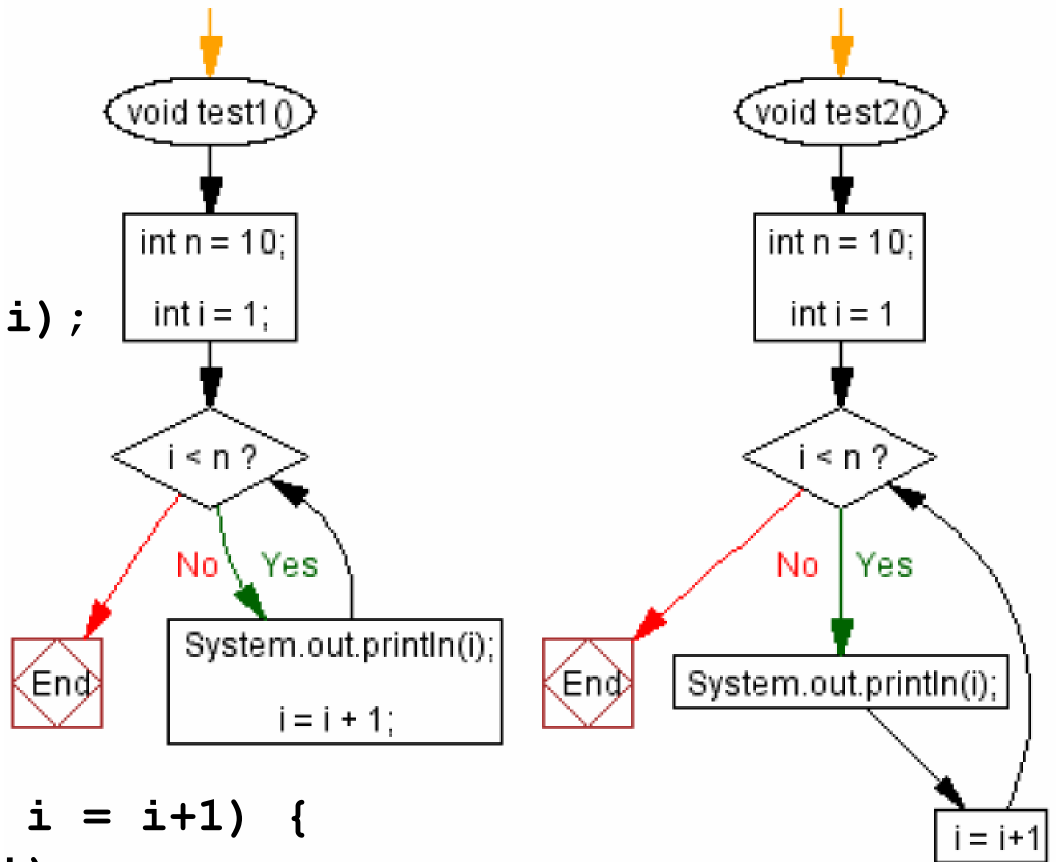
- If there is an edge from n_i to n_j
  - n_i is a **predecessor** of n_j
  - n_j is a **successor** of n_i
- For any node n
  - pred(n): the set of predecessors of n
  - succ(n): the set of successors of n
  - is a predicate/branch node if out-degree(n) > 1
  - is a terminal/end node if out-degree(n) = 0

in/out-degree: the number of ingoing/outgoing edges to/from a node

Examples: Visustin CFG generator

Demo version available: http://www.aivosto.com/visustin.html

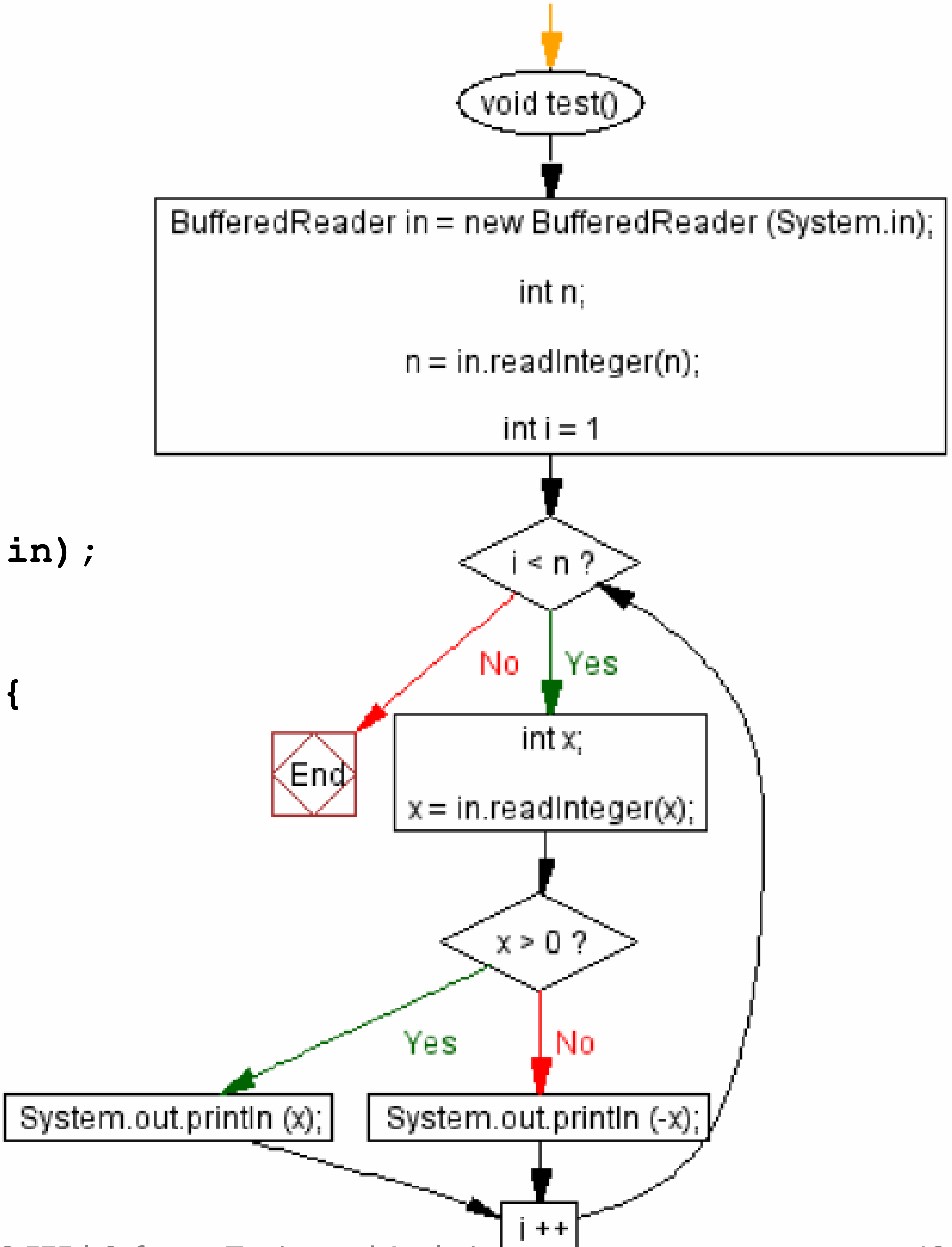
```
void test1() {
    int n = 10;
    int i = 1;
    while ( i < n) {
        System.out.println(i);
        i =  i + 1;
    }
}
```




```
void test2() {
    int n = 10;
    for (int i = 1; i < n; i = i+1) {
        System.out.println(i);
    }
}
```

Examples:

```
void test() {
   BufferedReader in =
     new BufferedReader (System.in);
   int n;
   n = in.readInteger(n);
   for (int i = 1; i < n; i ++) {
     int x;
     x = in.readInteger(x);
     if ( x > 0) {
        System.out.println (x);
     }
     else {
        System.out.println (-x);
     }
   }
}
```



Other CFG Generator Tools

- Eclipse plugin for CFG Generator
  - http://eclipsefcg.sourceforge.net/
- GNU tools
  - http://gcc.gnu.org/
- Avrora tool for assembly language
  - http://compilers.cs.ucla.edu/avrora/cfg.html

- and many more available on the Web..

Depth First Traversal

- CFG is a rooted, directed graph
  - Entry node as the root
- Depth-first traversal (depth-first searching)
  - Start at the root and explore as far/deep as possible along each branch before backtracking
  - Can build a **spanning tree** for the graph
- Spanning tree of a directed graph G contains all nodes of G such that
  - There is a path from the root **to any node reachable** in the original graph and
  - There are no cycles

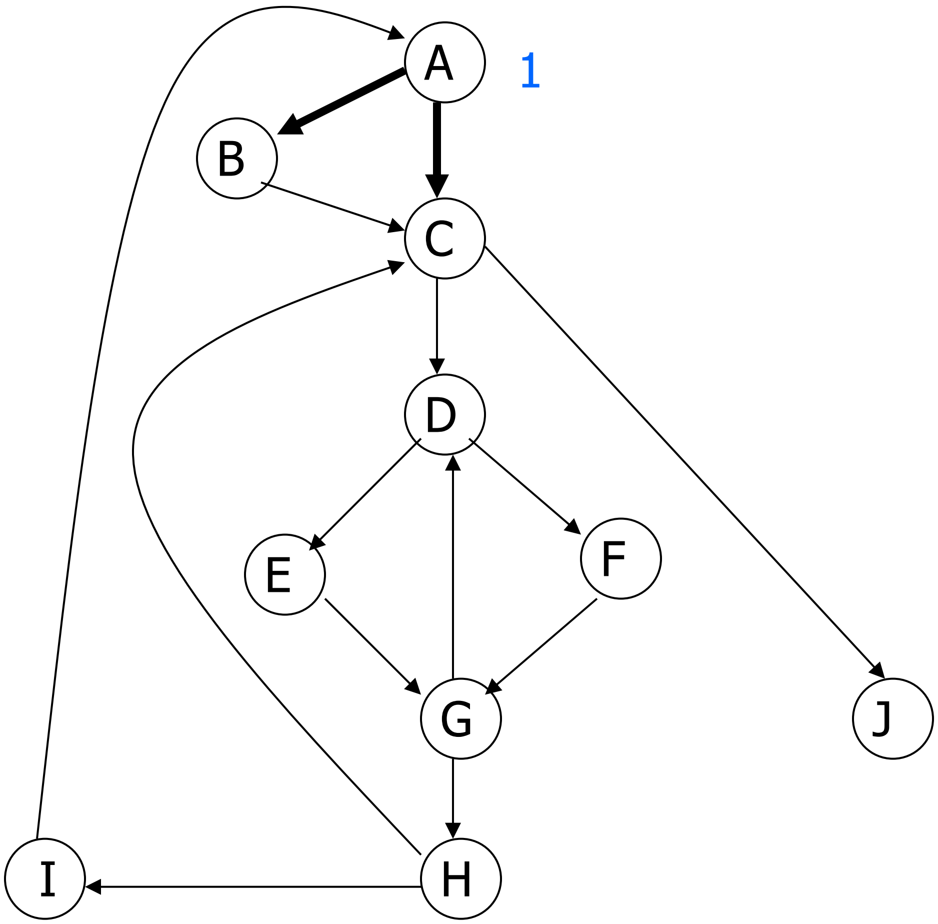
DFS Spanning Tree Algorithm

```
procedure span(v)  /* v is a node in the graph */
   inTree(v) = true
   for each w that is a successor of v do
        if (!inTree(w)) then
         add edge v → w to spanning tree
         span(w)
end span
```

- Initial: `span(n0)`

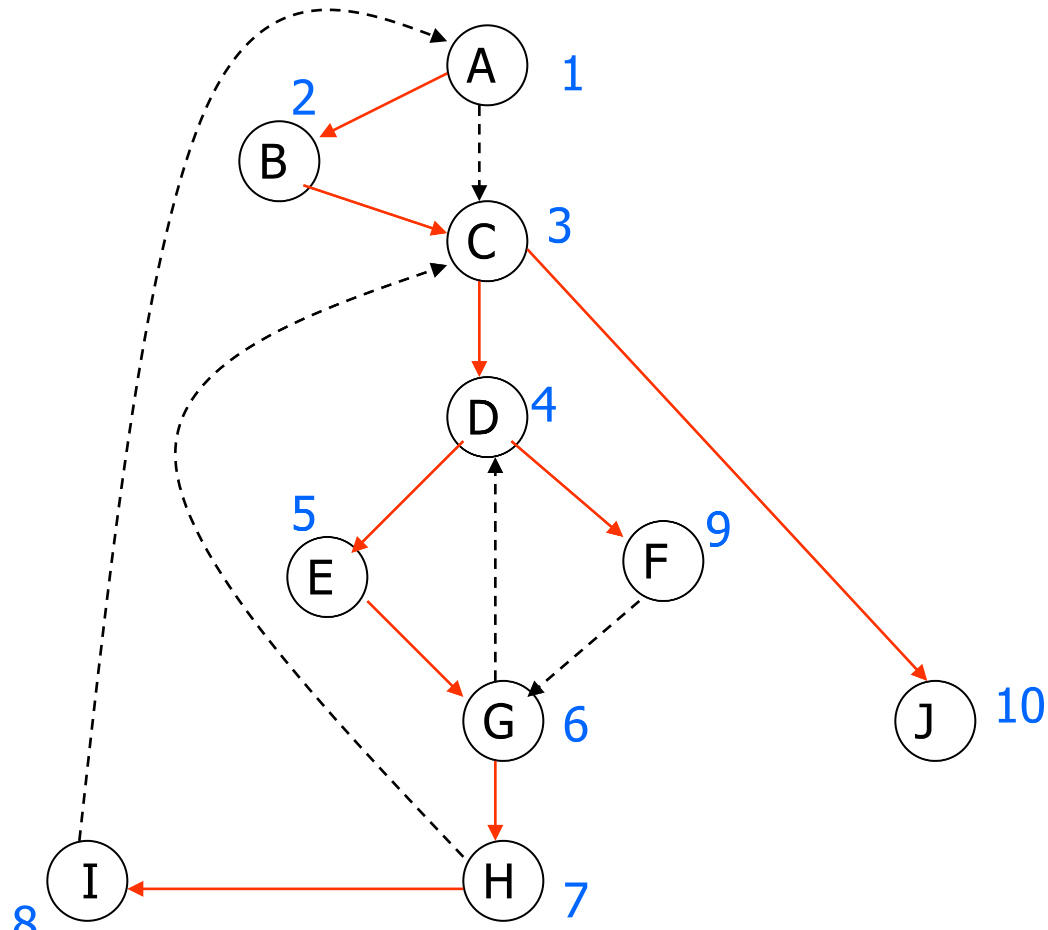
DFST Example

Nodes are numbered in the order visited during the search == ***depth first pre-order numbering.***



DFST Example

Nodes are numbered
in the order visited
during the search
== ***depth first pre-order numbering.***



Dominance

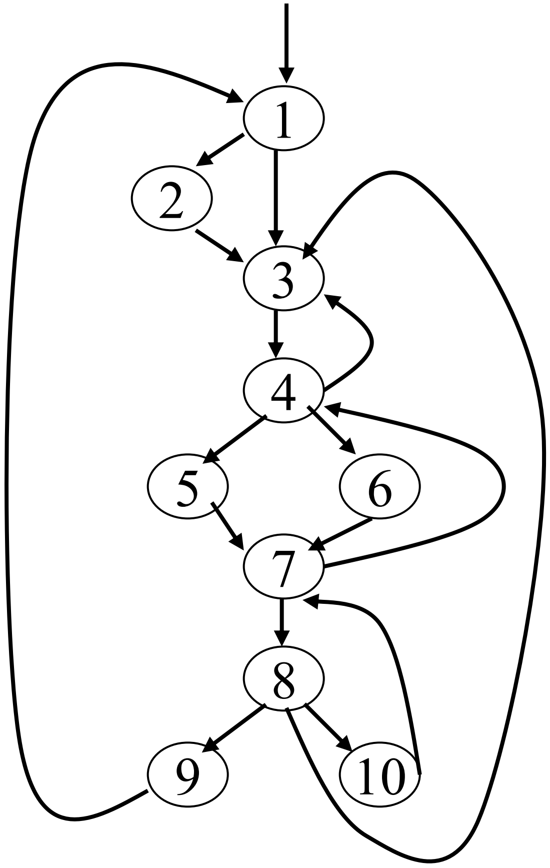
- Node d of a CFG *dominates* node n if every path from the entry node of the graph to n passes through d (d *dom* n)
  - Dom(n): the set of dominators of node n
  - Every node dominates itself: $n \in$ Dom(n)
  - Node d strictly dominates n if $d \in$ Dom(n) and $d \neq n$
  - Dominance-based loop recognition: entry of a loop dominates all nodes in the loop

Immediate Dominator

- Each node *n* has a unique *immediate dominator m* which is the last dominator of *n* on any path from the entry to *n* (*m idom n*), $m \neq n$
  - The immediate dominator *m* of *n* is the strict dominator of *n* that is closest to *n*

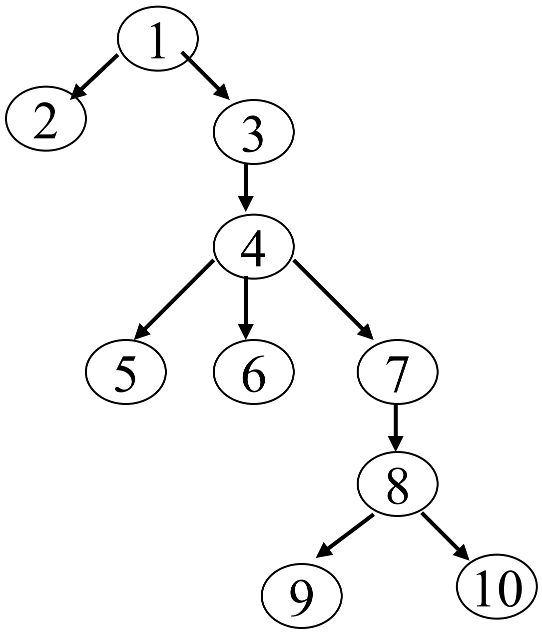
Dominator Example

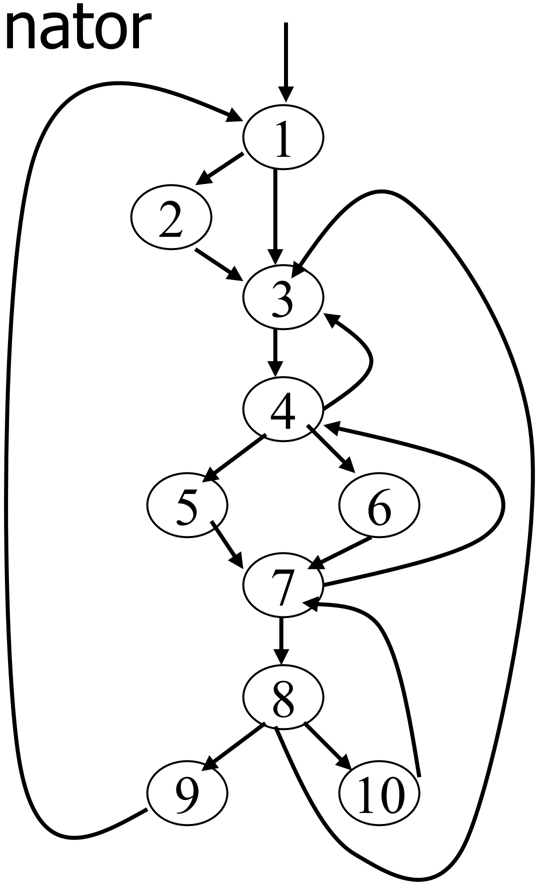
Block	Dom	IDom
1	{1}	—
2	{1,2}	1
3	{1,3}	1
4	{1,3,4}	3
5	{1,3,4,5}	4
6	{1,3,4,6}	4
7	{1,3,4,7}	4
8	{1,3,4,7,8}	7
9	{1,3,4,7,8,9}	8
10	{1,3,4,7,8,10}	8



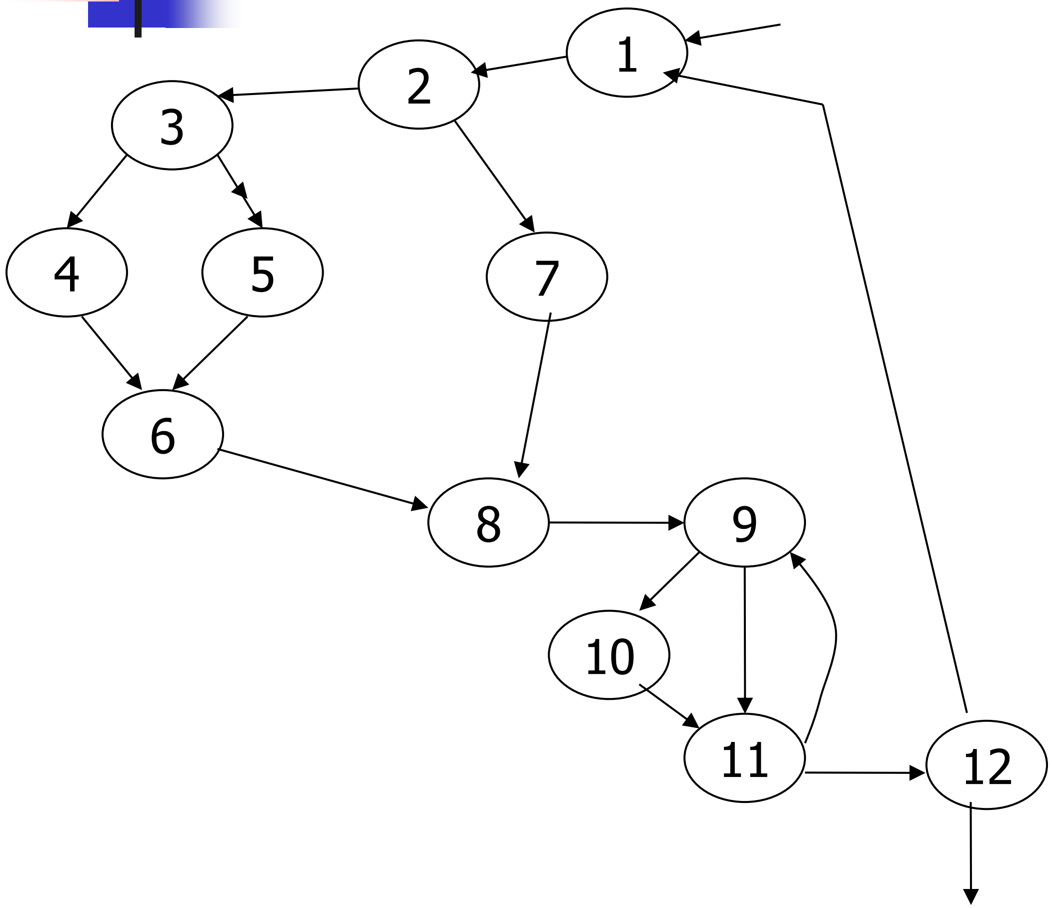
Dominator Trees

- A node's parent is its immediate dominator



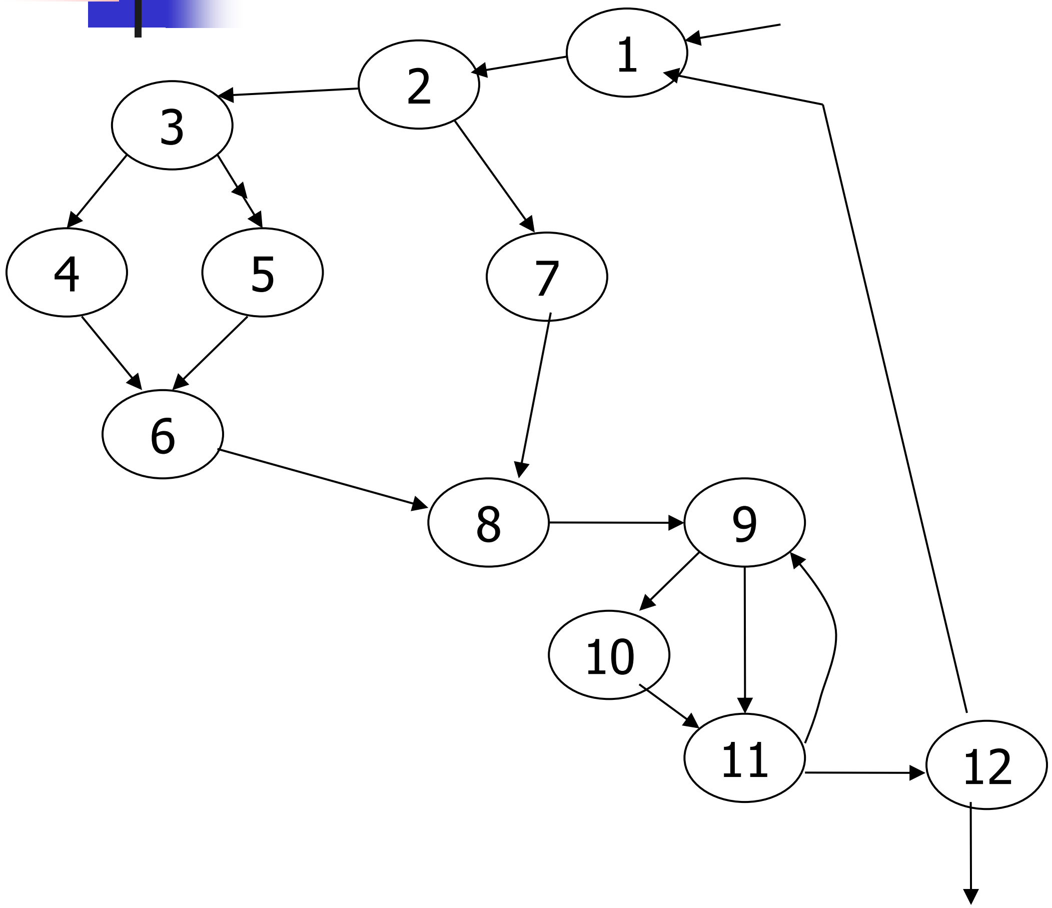


Exercise



Block	Dom	IDom
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		

Exercise

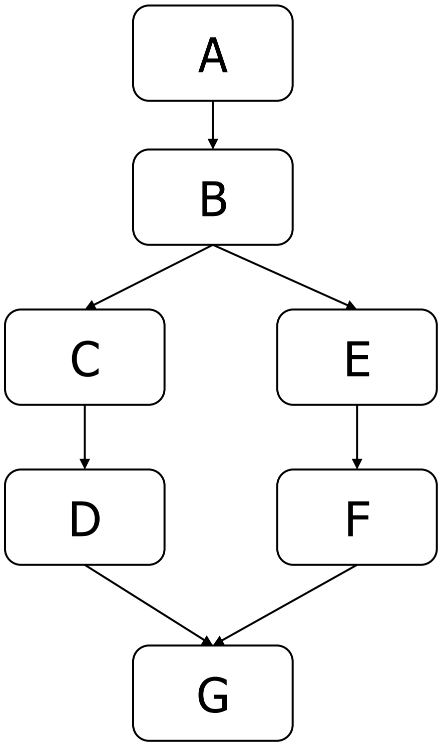


Block	Dom	IDom
1	1	-
2	1,2	1
3	1,2,3	2
4	1,2,3,4	3
5	1,2,3,5	3
6	1,2,3,6	3
7	1,2,7	2
8	1,2,8	2
9	1,2,8,9	8
10	1,2,8,9,10	9
11	1,2,8,9,11	9
12	1,2,8,9,11,12	11

Post-Dominators

- **Post-dominators**: Calculated in the reverse of the CFG, using a special "exit" node as the root.

Example:



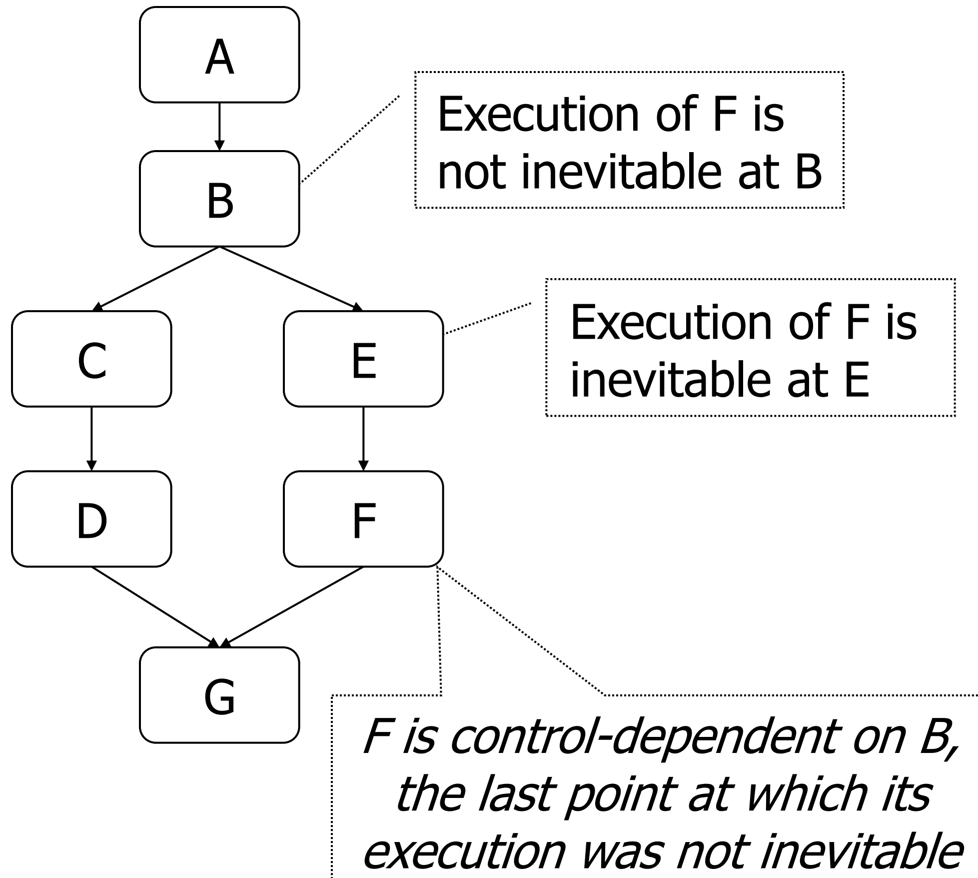
- A pre-dominates all nodes; G post-dominates all nodes
- F and G post-dominate E
- G is the immediate post-dominator of B
  - C does *not* post-dominate B
- B is the immediate pre-dominator of G
  - F does *not* pre-dominate G

Control dependence

- Control dependence defined by post-dominators:
  - Consider again a node *N* that is (not always) reachable
  - There must be some node *C* with the following property:
    - *C* is a predicate node
    - *C* is not post-dominated by *N*
    - a successor of C in the CFG is post-dominated by *N*
  - Then, *N* is control-dependent on *C*.

- Intuitively, *C* was the last decision that controlled whether *N* executed

Control Dependence

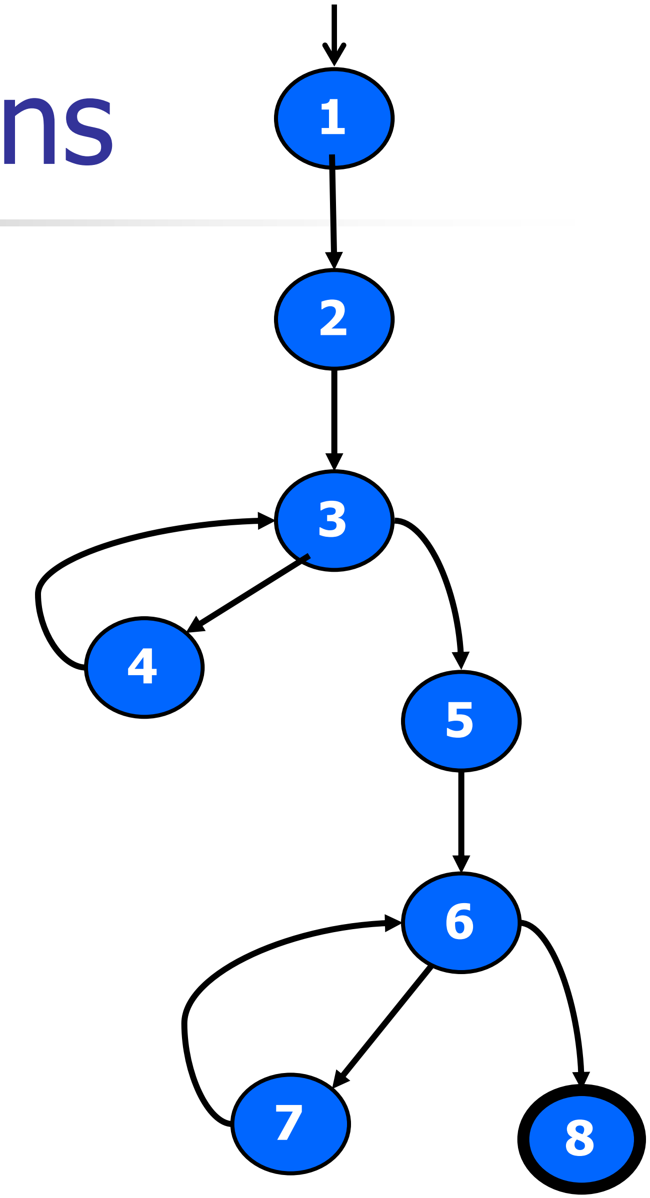


Coverage Criteria Based on Control Flow and any Graph Model in General

- Node Coverage (NC)
- Edge Coverage (EC)
  - ~ Transition Coverage
- Edge-Pair Coverage (EPC)
  - each reachable path of length up to 2
- Prime Path Coverage (PPC)
- Complete Path Coverage (CPC)
  - Not practical in general

Covering Transitions

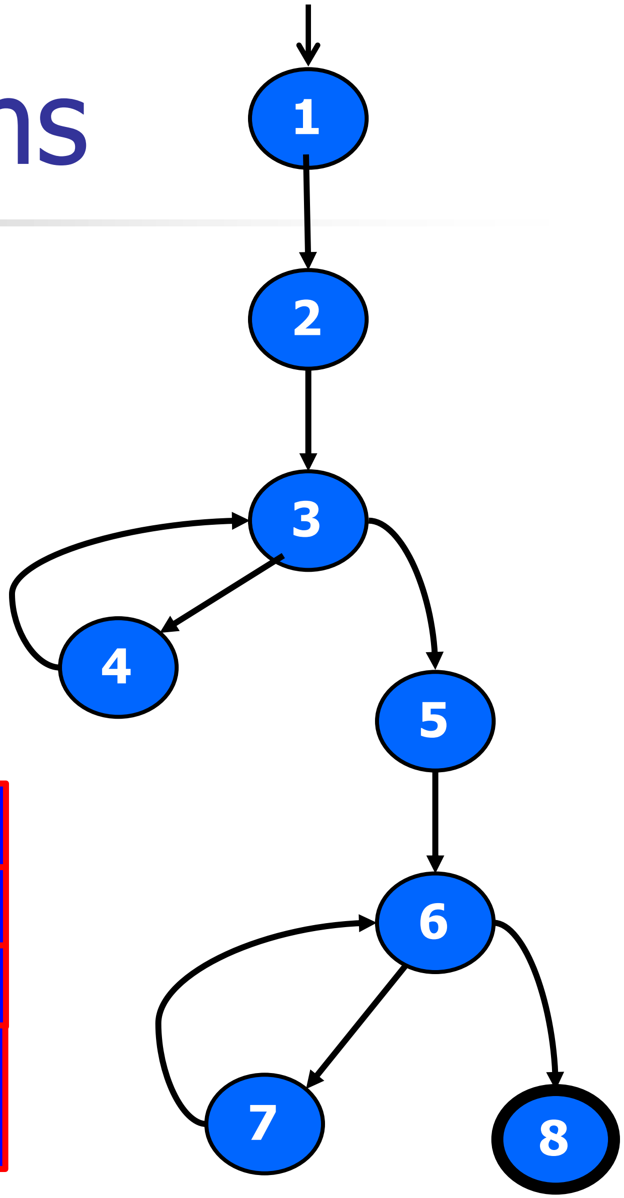
Edge Coverage	
TR	**Test Path**
A. [1, 2]	[1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [2, 3]	
C. [3, 4]	
D. [3, 5]	
E. [4, 3]	
F. [5, 6]	
G. [6, 7]	
H. [6, 8]	
I. [7, 6]	



Covering Transitions

Edge-Pair Coverage	
TR	**Test Paths**
A. [1, 2, 3] **B. [2, 3, 4]** **C. [2, 3, 5]** **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]**	**i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]** **ii. [1, 2, 3, 5, 6, 8]** **iii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]**

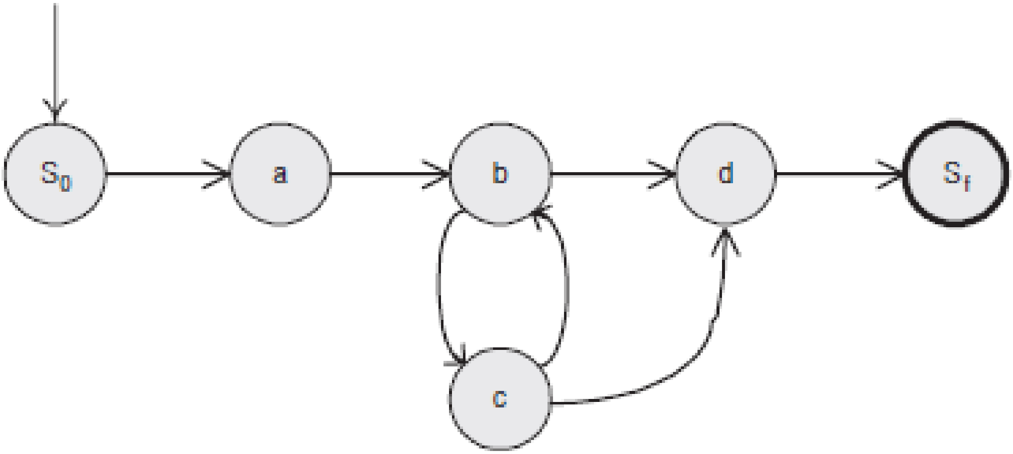
TP	TRs toured	*sidetrips*
i	A, B, D, E, F, G, I, J	C, H
ii	A, **C**, E, **H**	
iii	A, B, D, E, F, G, I, J, **K**, **L**	C, H



Some definitions..

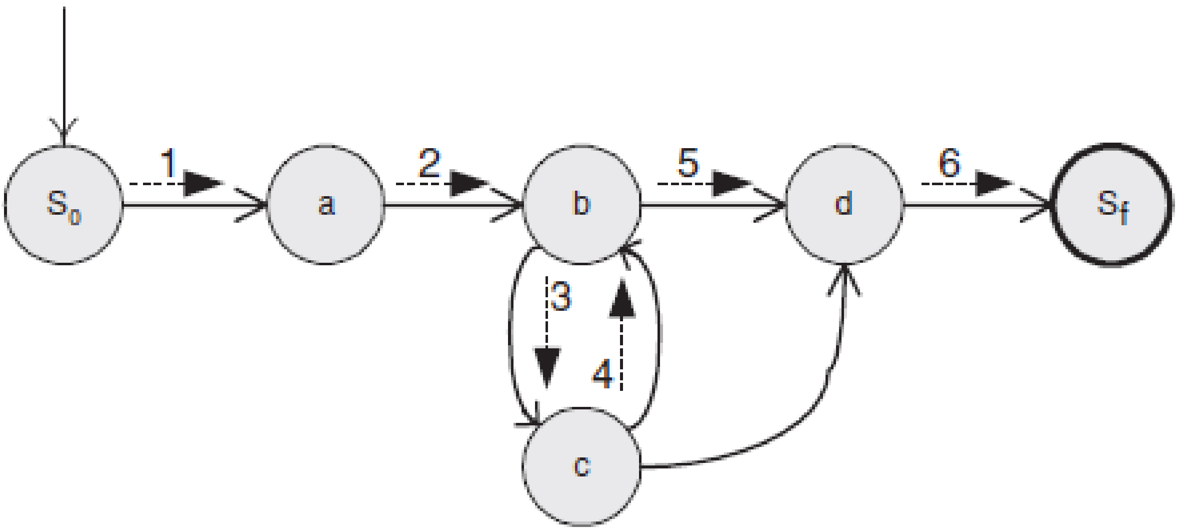
- For relaxing the test requirements

q = [a, b, d] is a strict definition that does not confirm, e.g., p = [a, b, c, b, d]



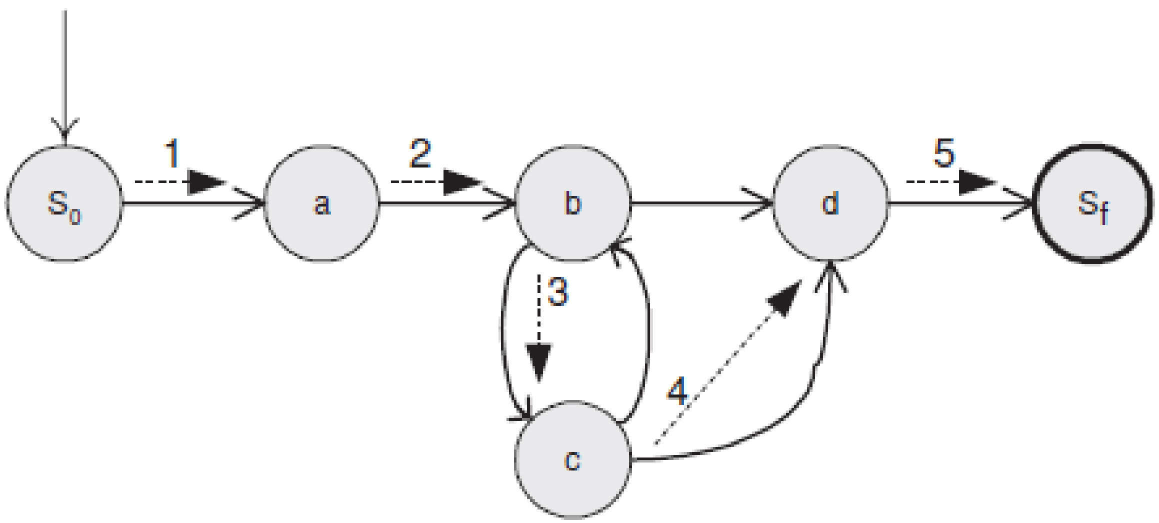
Some definitions..

- Sidetrip



Some definitions..

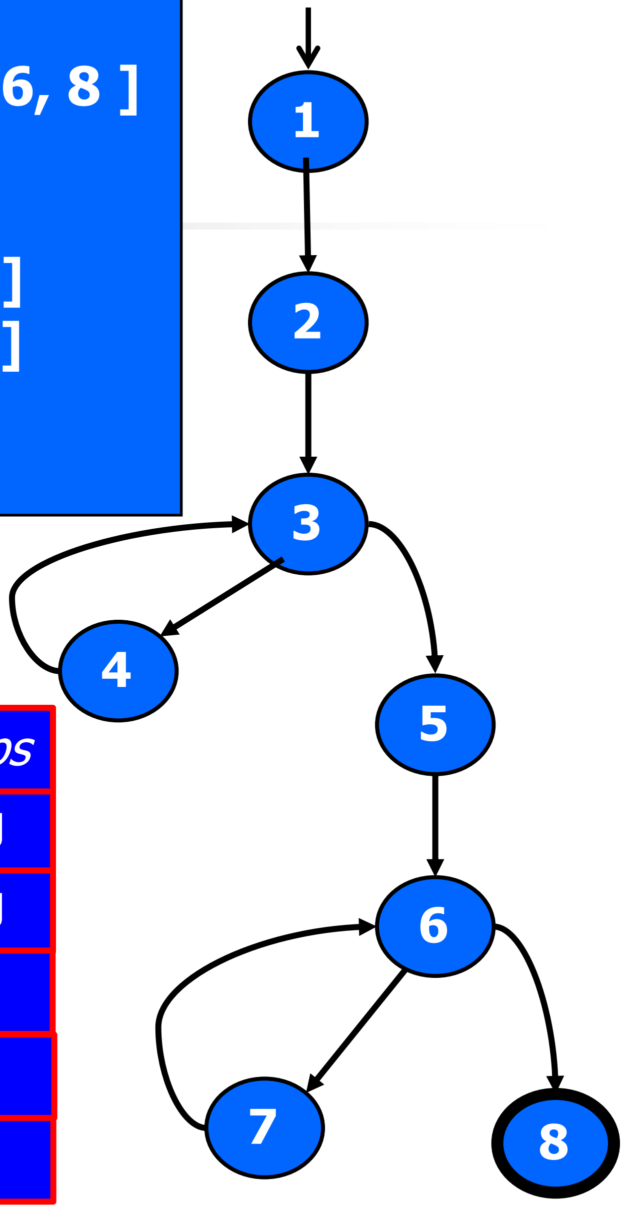
- Detour



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**	



Prime Paths

- A **prime path is** a simple path that does not appear as a proper subpath of any other simple path

- Prime paths can be systematically discovered

Discovering Prime Paths

cannot be extended

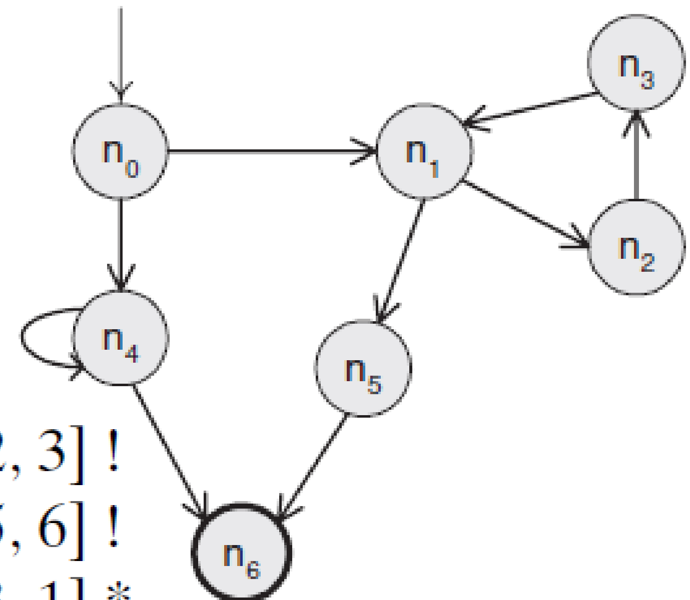
cycle

1) [0]
2) [1]
3) [2]
4) [3]
5) [4]
6) [5]
7) [6] !

8) [0, 1]
9) [0, 4]
10) [1, 2]
11) [1, 5]
12) [2, 3]
13) [3, 1]
14) [4, 4] *
15) [4, 6] !
16) [5, 6] !

17) [0, 1, 2]
18) [0, 1, 5]
19) [0, 4, 6] !
20) [1, 2, 3]
21) [1, 5, 6] !
22) [2, 3, 1]
23) [3, 1, 2]
24) [3, 1, 5]

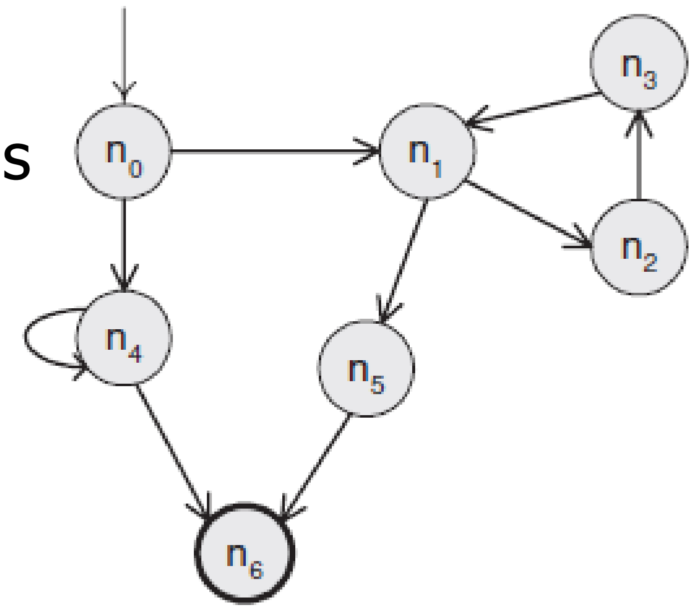
25) [0, 1, 2, 3] !
26) [0, 1, 5, 6] !
27) [1, 2, 3, 1] *
28) [2, 3, 1, 2] *
29) [2, 3, 1, 5]
30) [3, 1, 2, 3] *
31) [3, 1, 5, 6] !



32) [2, 3, 1, 5, 6]!

Discovering Prime Paths

- Eliminating the paths that are proper subpaths of other paths leads to 8 prime paths



14) [4, 4] *
19) [0, 4, 6] !
25) [0, 1, 2, 3] !
26) [0, 1, 5, 6] !
27) [1, 2, 3, 1] *
28) [2, 3, 1, 2] *
30) [3, 1, 2, 3] *
32) [2, 3, 1, 5, 6]!

Recall: Coverage Criteria Based on CFG

and graphs in general..

- Node Coverage (NC)
- Edge Coverage (EC)
- Edge-Pair Coverage (EPC)
  - each reachable path of length up to 2
- Prime Path Coverage (PPC)
- Complete Path Coverage (CPC)

The *subsumes* relation

- *Test adequacy criterion A subsumes test adequacy criterion B iff, for every program P, every test suite satisfying A with respect to P also satisfies B with respect to P*

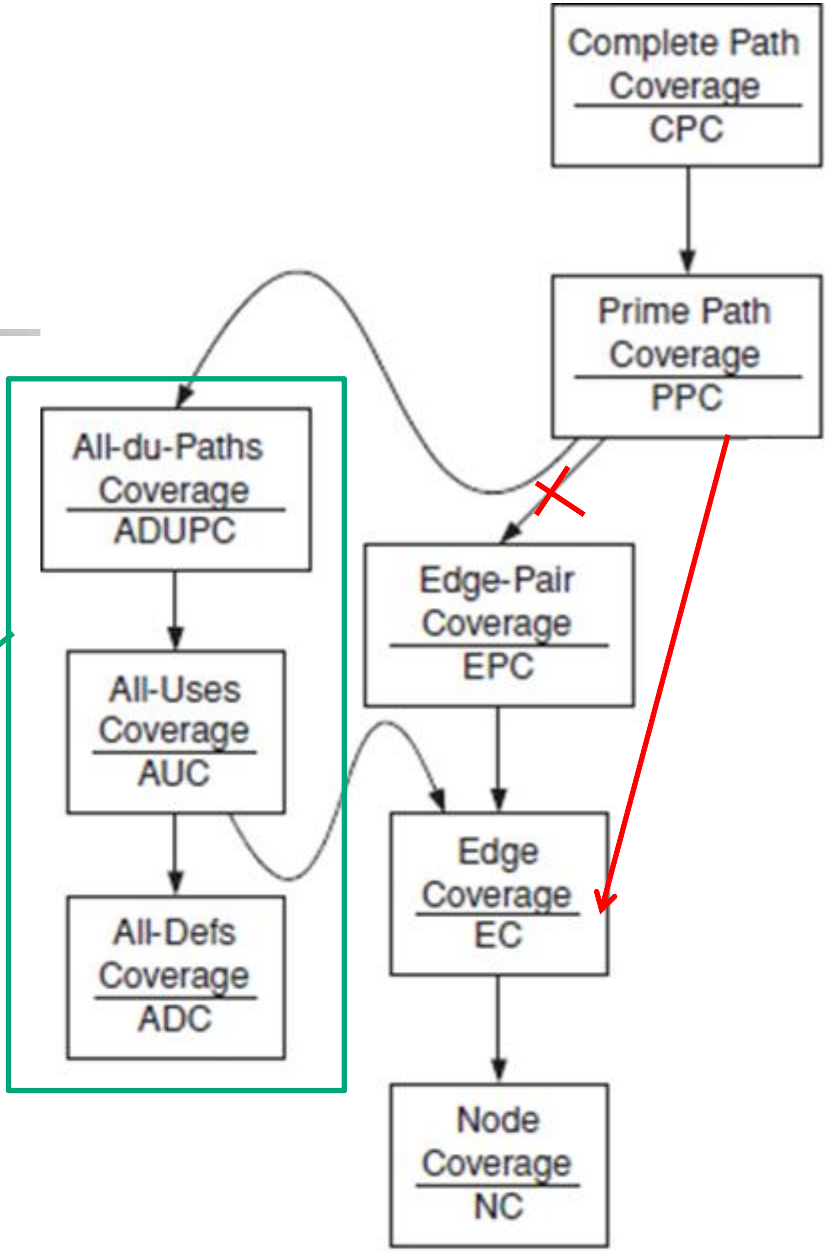
- Example:

  Exercising all program branches (branch coverage) *subsumes* exercising all program statements

- A common analytical comparison of closely related criteria
  - Useful for working from easier to harder levels of coverage, but not a direct indication of quality

Subsumes Relations

to be discussed later



Counter Example

- Not possible to satisfy the test requirement [2,3,3] for edge-pair coverage with any prime path