EN.601.422 / EN.601.622

Software Testing & Debugging

Intro to Graphs

- Directed graphs are the basis of many coverage criteria
- Graph models can be extracted from many different software artifacts:
  - User manual, API doc, data structures, source code, Graphical user interface, UML statecharts, etc.
- Different coverage criteria cover the graph in different ways

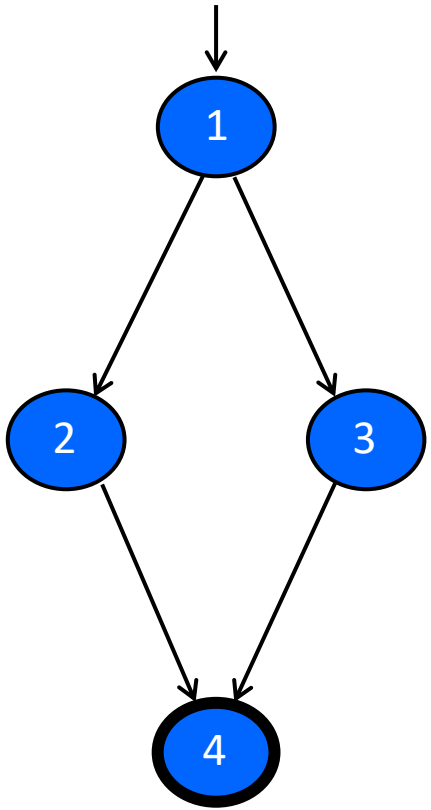
Intro to Graphs

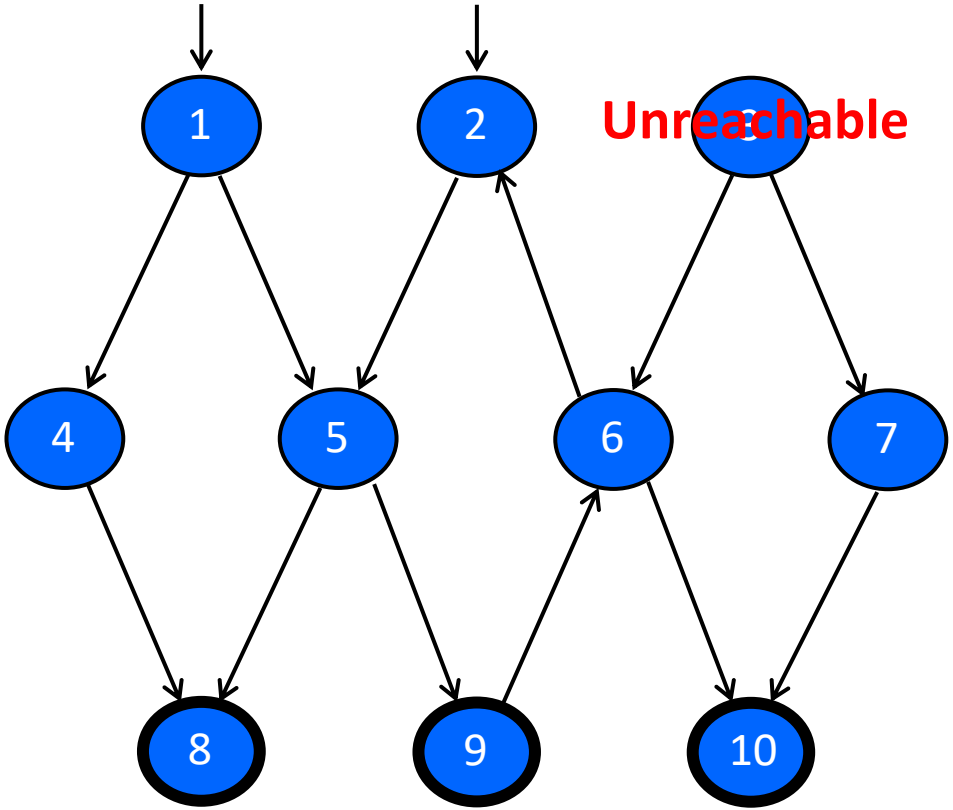
- A Graph **G** is defined as:
  - A set N of nodes, N is not empty
  - A set N_0 of initial nodes, N_0 is not empty
  - A set N_f of final nodes, N_f is not empty
  - A set E of edges, each edge from one node to another
  - (n_i, n_j), i is predecessor, j is successor
- N, N_o, and N_f must have at least one node
- Edges are typically written as (n_i, n_j) or simply $n_i\, n_j$

Intro to Graphs

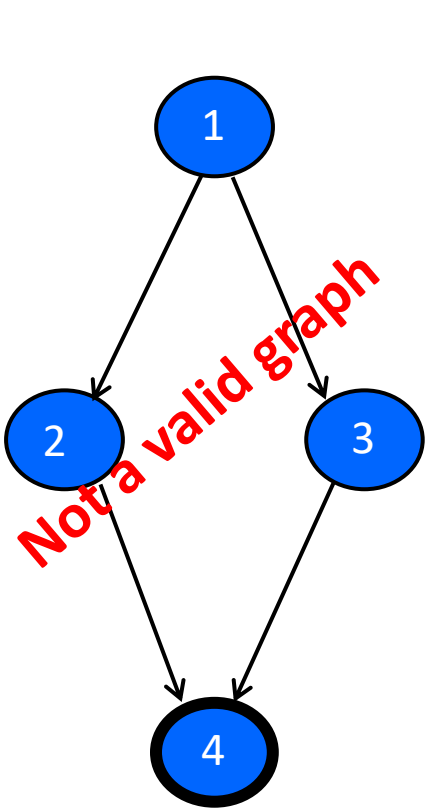
- A *Path* is a sequence of nodes where each pair of adjacent nodes (n_i , n_j) in the path must be present in the set *E*
- Length of a path is the number of edges it contains
- A *subpath* of a given path P is a subsequence of P
- A *cycle* is a path that begins and ends at the same node
- We say node n_j is reachable from node n_i if there exists a path from n_i to n_j
- A node is unreachable if it is not possible to reach that node from an initial node

Intro to Graphs





Unreachable



Not a valid graph

Intro to Graphs

- **Test Path**: a path P taken by a test case on a given graph G where P starts at a node $n_i \in N_o$ and ends at $n_j \in N_f$
- Any $n_i \in n_f$ must be reachable from the other nodes in N (excluding final nodes) and no nodes in N should be reachable from any $n_i \in n_f$
- **Visit**: Test path P visits node n if n is in P. Test path P visits edge (n_i, n_j) if n_i, n_j is in P.
- **Tour**: Test path p tours subpath q if q is in p.

Exercise

- Give all test paths for graph G
- Give a path that is not a test path

Infinite number of test paths:

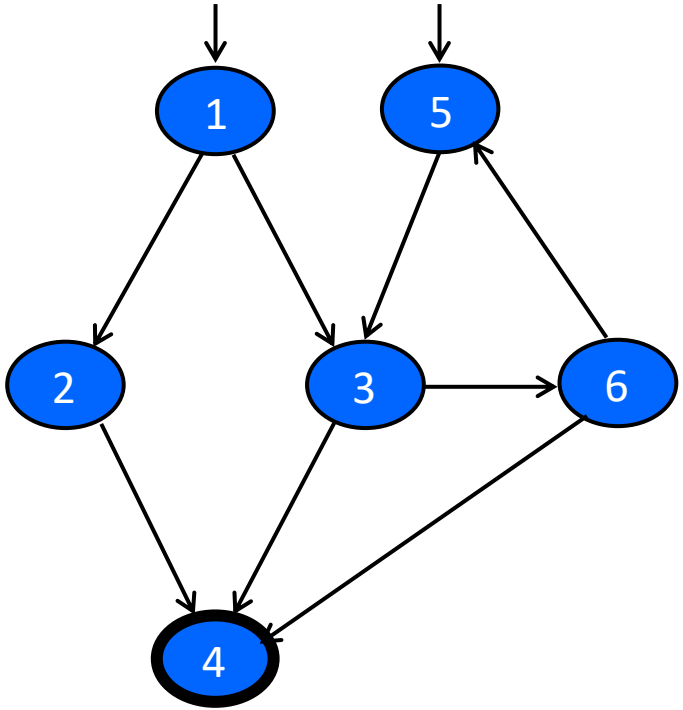
1 2 4

1 (3 6 5)* 3 4

1 (3 6 5)* 3 6 4

5 (3 6 5)* 3 4

5 (3 6 5)* 3 6 4



Structural Graph Coverage Criteria

- **path(t)**: The test path executed by test *t*
- **path (*T*)** : The set of test paths executed by the set of tests *T*
- **Graph Coverage**: 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 named tr in TR, there is a test path in path(T) that meets the test requirement tr

Graph Coverage Criteria

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*.

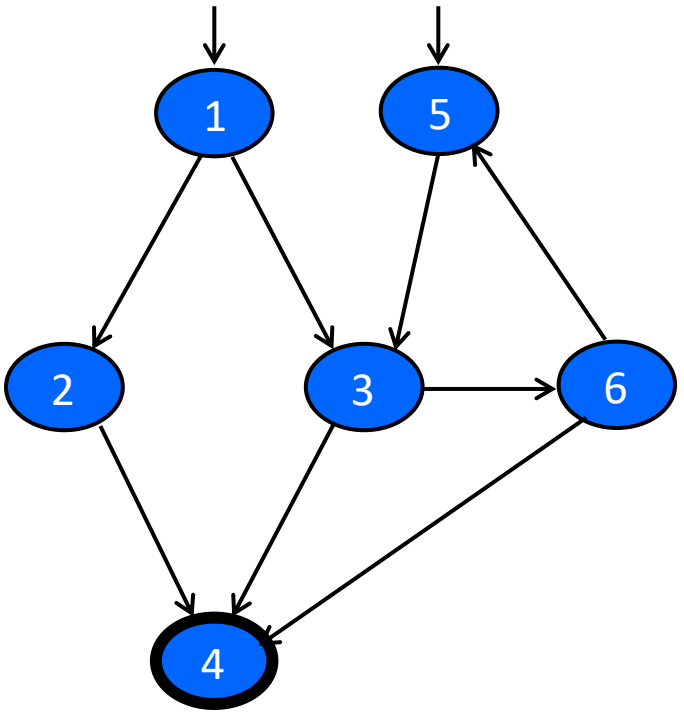
Edge Coverage (EC) : TR contains each reachable path of length up to 1, inclusive, in G.

Example

TR_Node = {1, 2, 3, 4, 5, 6}
TR_Edge = {(1, 2), (2, 4), (1, 3), (3, 4)
(5, 3), (3, 6), (6, 5), (6,4)}

T_Node = **{** t1 = {1, 2, 4}, t2 = {1, 3, 4},
t3 = {5, 3, 6, 4} **}**
achieves node coverage.

T_Edge = **{** t1 = {1, 2, 4}, t2 = {1, 3, 4},
t3= {5, 3, 6, 5, 3, 6, 4} **}**
achieves both node
and edge coverages.



Graph Coverage Criteria

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

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

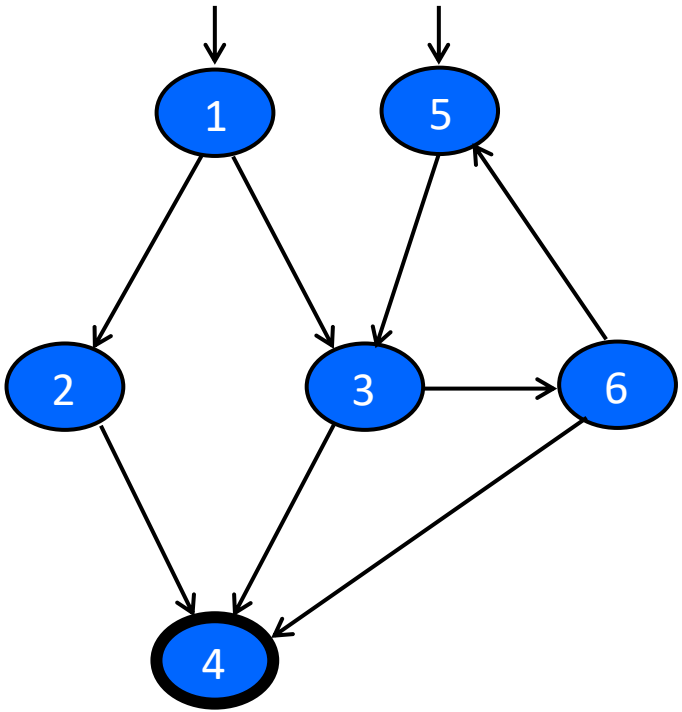
Example

Write EPC TRs and test paths to achieve EPC

TR_EdgePair = {**(1, 2, 4)**, (1, 3, 4), (1, 3, 6),
(5, 3, 4), (5, 3, 6), (3, 6, 5),
(3, 6, 4), (6, 5, 3)}

T_EdgePair = { t1 = **{1, 2, 4}**, t2 = {1, 3, 4},
t3 = {1, 3, 6, 4}, t4 = {5, 3, 6, 5, 3, 4} }
achieves Edge-Pair coverage.

How about Path Coverage?

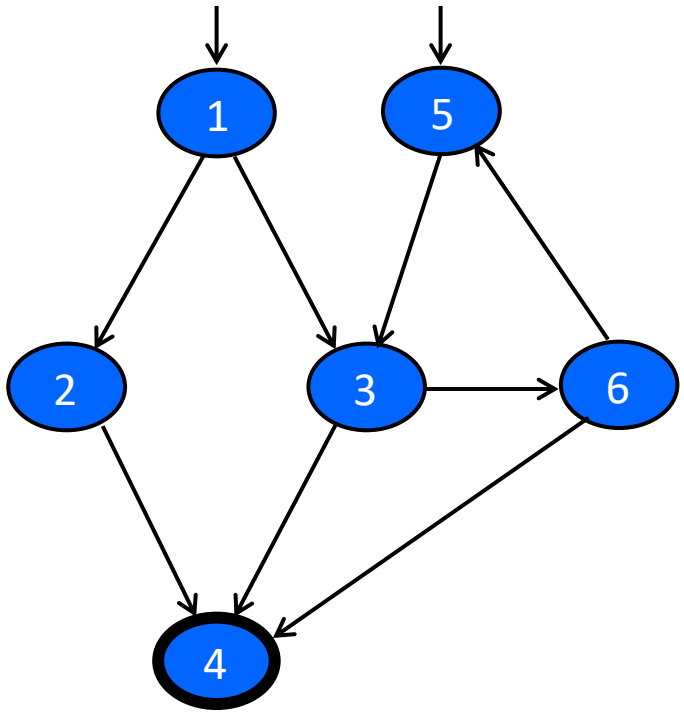


Prime Path

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

Prime Path

Prime Paths = { {1, 2, 4}, {1, 3, 4},
{1, 3, 6, 4}, {5, 3, 4},
{5, 3, 6, 4}, {5, 3, 6, 5},
{6, 5, 3, 6}, {3, 6, 5, 3} }



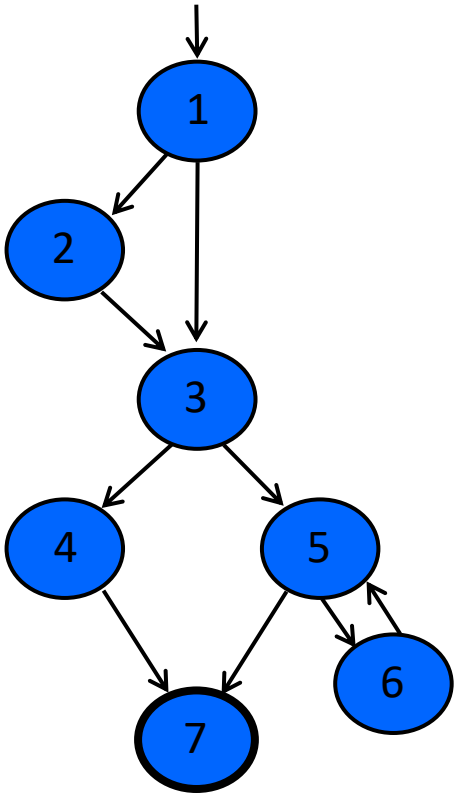
Graph Coverage Criteria

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

- Attempts to “deal with” loops:
  - **1970s** : Execute cycles once ([5, 3, 6] in previous example, informal)
  - **1980s** : Execute each loop, exactly once
  - **1990s** : Execute loops 0 times, once, and more than once (loop adequacy)
  - **2000s** : Prime paths

Exercise

List all the prime paths



Example

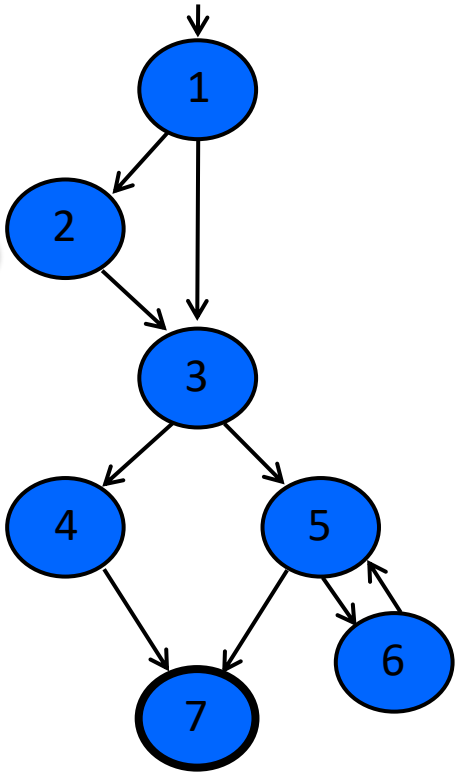
List all the prime paths

Prime Paths = { {1, 2, 3, 4, 7}, {1, 2, 3, 5, 7} ,
{1, 2, 3, 5, 6}, {1, 3, 4, 7}, {1, 3, 5, 7},
{1, 3, 5, 6} {6, 5, 7}, {6, 5, 6}
{5, 6, 5} }

Execute the loop once

Execute the loop more than once

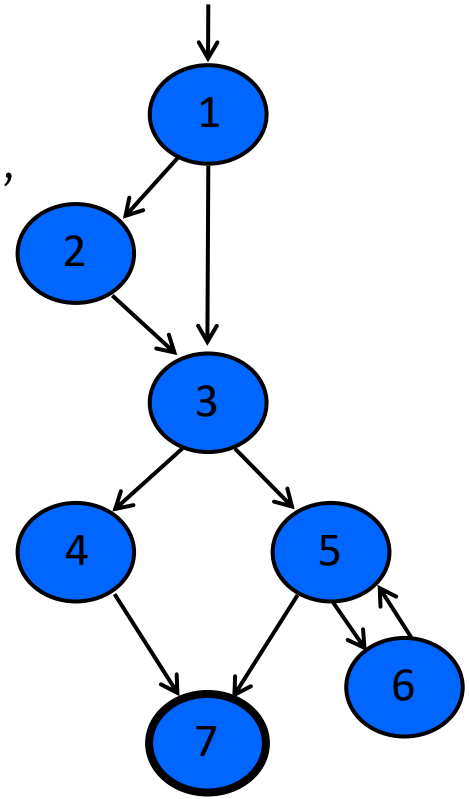
Execute the loop 0 times



Prime Path

Prime Paths = { {1, 2, 3, 4, 7}, {1, 2, 3, 5, 7} ,
{1, 2, 3, 5, 6}, {1, 3, 4, 7}, {1, 3, 5, 7},
{1, 3, 5, 6}, {6, 5, 7}, {6, 5, 6},
{5, 6, 5} }

Test Paths = { {1, 2, 3, 4, 7}, {1, 2, 3, 5, 7},
{1, 2, 3, 5, 6, 5, 6, 5, 7}, {1, 3, 4, 7},
{1, 3, 5, 7}, {1, 3, 5, 6, 5, 7} }



Finding Prime Paths

List all the simple paths

Length 0

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

Length 1

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

Length 2

[1, 2, 3]
[1, 3, 4]
[1, 3, 5]
[2, 3, 4]
[2, 3, 5]
[3, 4, 7] !
[3, 5, 7] !
[3, 5, 6] !
[5, 6, 5] *
[6, 5, 7] !
[6, 5, 6] *

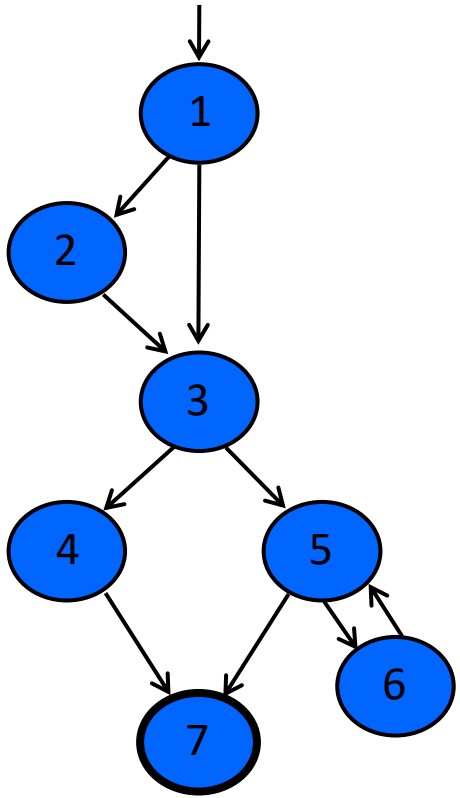
Length 3

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

Length 4

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

1. *** means path cycles**
2. **! means path terminates**



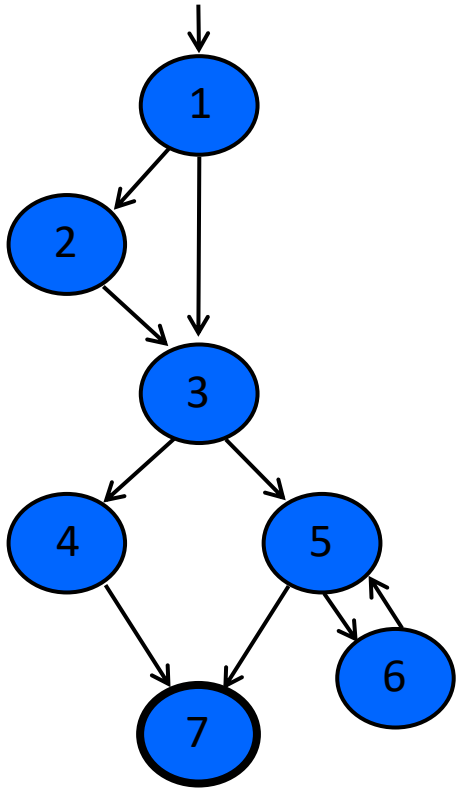
Cross out the ones that are proper subpath

List all prime paths

Length 0	Length 1	Length 2	Length 3
~~[1]~~	~~[1, 2]~~	~~[1, 2, 3]~~	~~[1, 2, 3, 4]~~
~~[2]~~	~~[1, 3]~~	~~[1, 3, 4]~~	~~[1, 2, 3, 5]~~
~~[3]~~	~~[2, 3]~~	~~[1, 3, 5]~~	**[1, 3, 4, 7] !**
~~[4]~~	~~[3, 4]~~	~~[2, 3, 4]~~	**[1, 3, 5, 7] !**
~~[5]~~	~~[3, 5]~~	~~[2, 3, 5]~~	**[1, 3, 5, 6] !**
~~[6]~~	~~[4, 7] !~~	~~[3, 4, 7] !~~	~~[2, 3, 4, 7] !~~
~~[7] !~~	~~[5, 7] !~~	~~[3, 5, 7] !~~	~~[2, 3, 5, 6] !~~
	~~[5, 6]~~	~~[3, 5, 6] !~~	~~[2, 3, 5, 7] !~~
	~~[6, 5]~~	**[5, 6, 5] ***	
		[6, 5, 7] !	
		[6, 5, 6] *	

Length 4

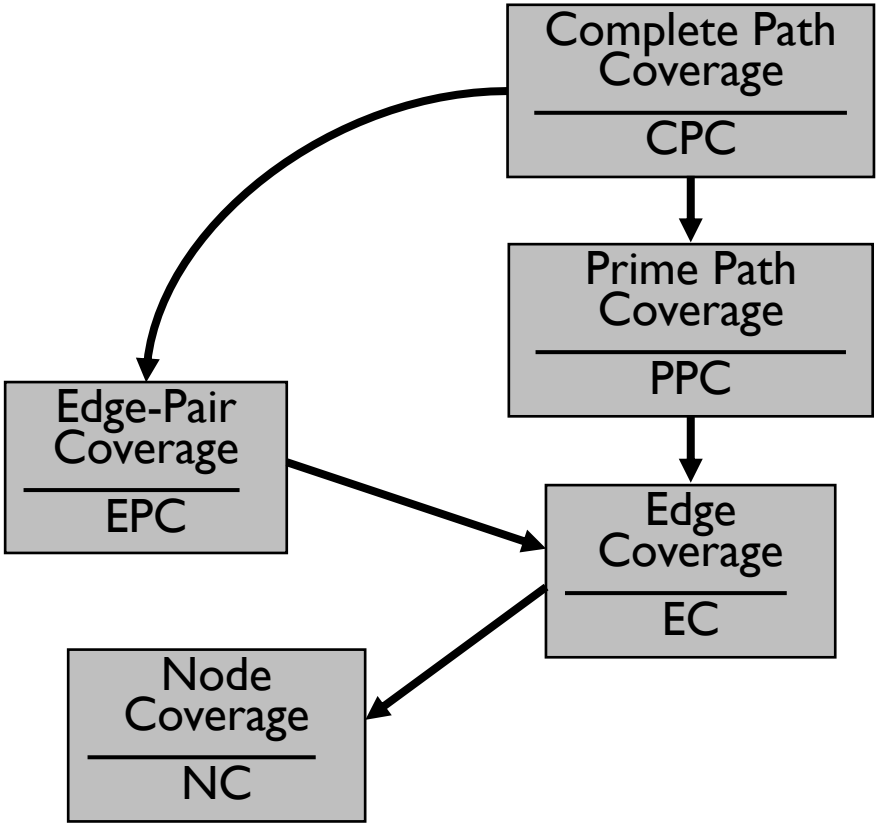
[1, 2, 3, 4, 7] !
[1, 2, 3, 5, 7] !
[1, 2, 3, 5, 6] !



Questions

- Does edge coverage subsume node coverage?
- Does node coverage subsume edge coverage?
- Does edge-pair coverage subsume edge coverage?
- Does prime path coverage subsume edge-pair coverage?

Graph Coverage Criteria Subsumption Relationship



Extracting Graphs From Source Code

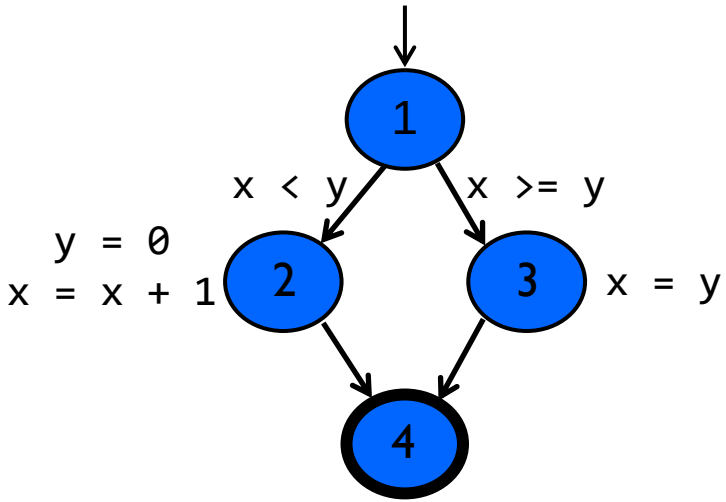
- A common application of graph criteria is the source code
  - Control flow graph (CFG)
- **Node coverage**: execute every statement
- **Edge coverage**: execute every branch/decision
- **Path coverage**: execute loops (i.e., while, for, do-while etc.)

Control Flow Graph

- Control flow graph: graph representation of source code showing the execution flow of the program
  - **Basic Block**: a straight-line code sequence with no branches (i.e., with exactly one entry point and one output point). A basic block corresponds to a node in CFG with one in-edge and one out-edge.
  - **Decision Node**: A node with at least two out-edges
  - **Junction Node**: A node with more than one in-edges
  - **Dummy Node**: A node that does not correspond to a program statement.

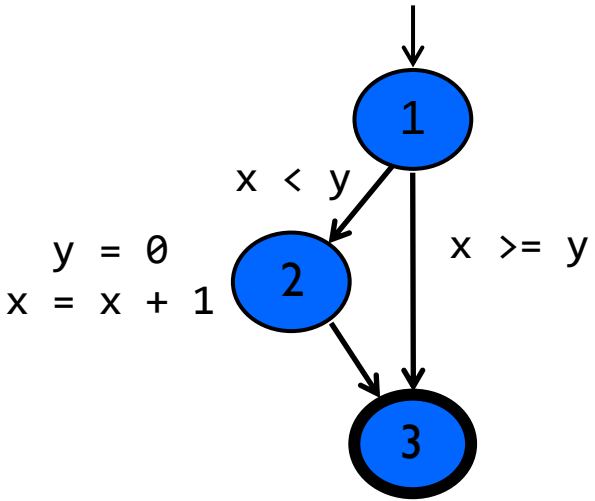
If statement

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



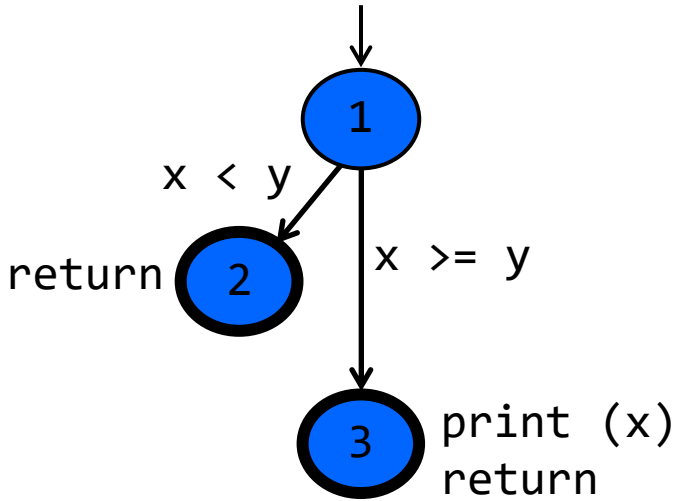
If statement

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



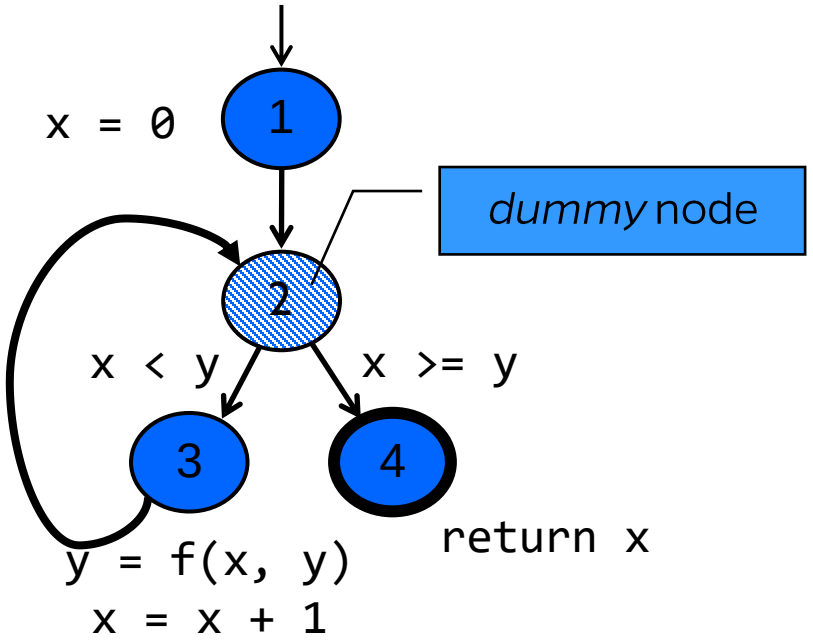
If statement

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



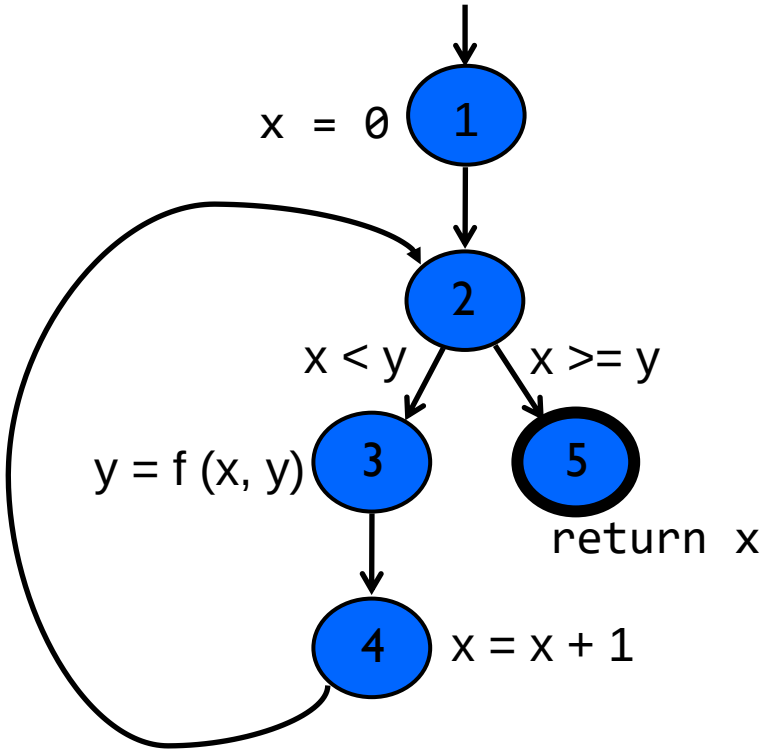
While loop

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



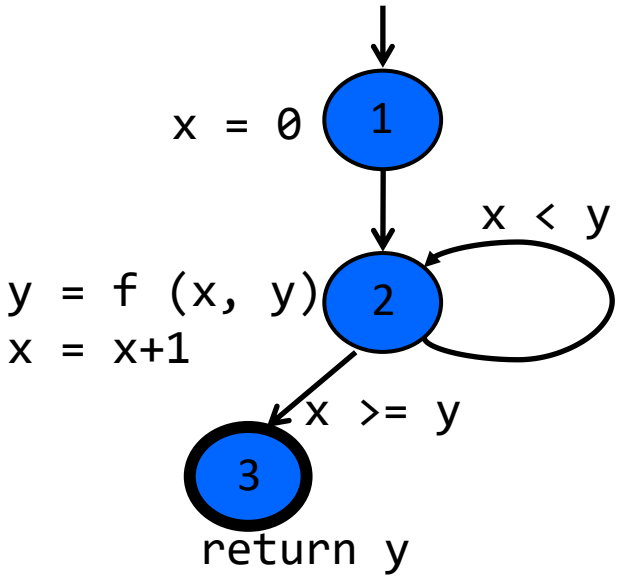
For loop

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

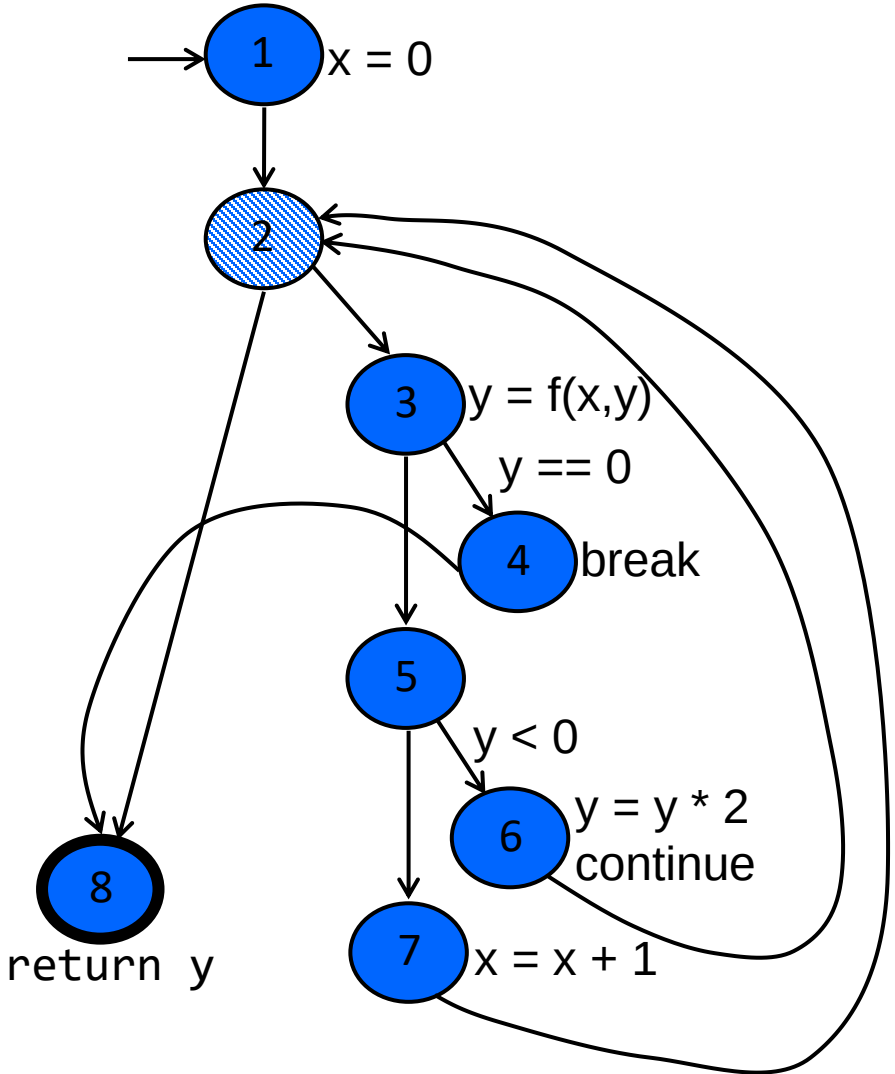


do-while

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

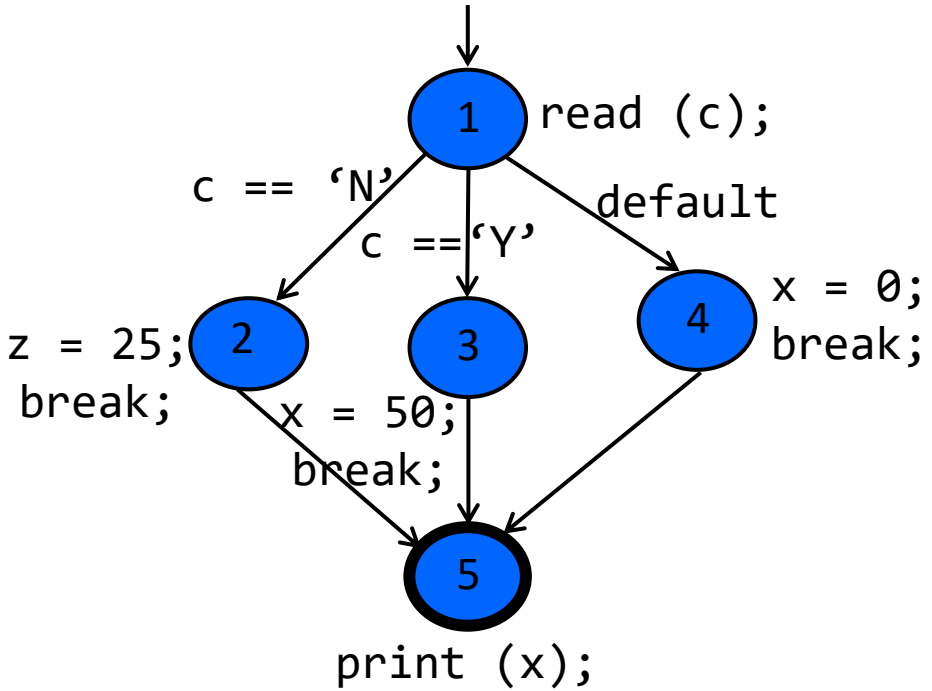


```
x = 0;
while (x < y)
{
    y = f (x, y);
    if (y == 0)
    {
        break;
    } else if (y < 0)
    {
        y = y * 2;
        continue;
    }
    x = x + 1;
}
return (y);
```



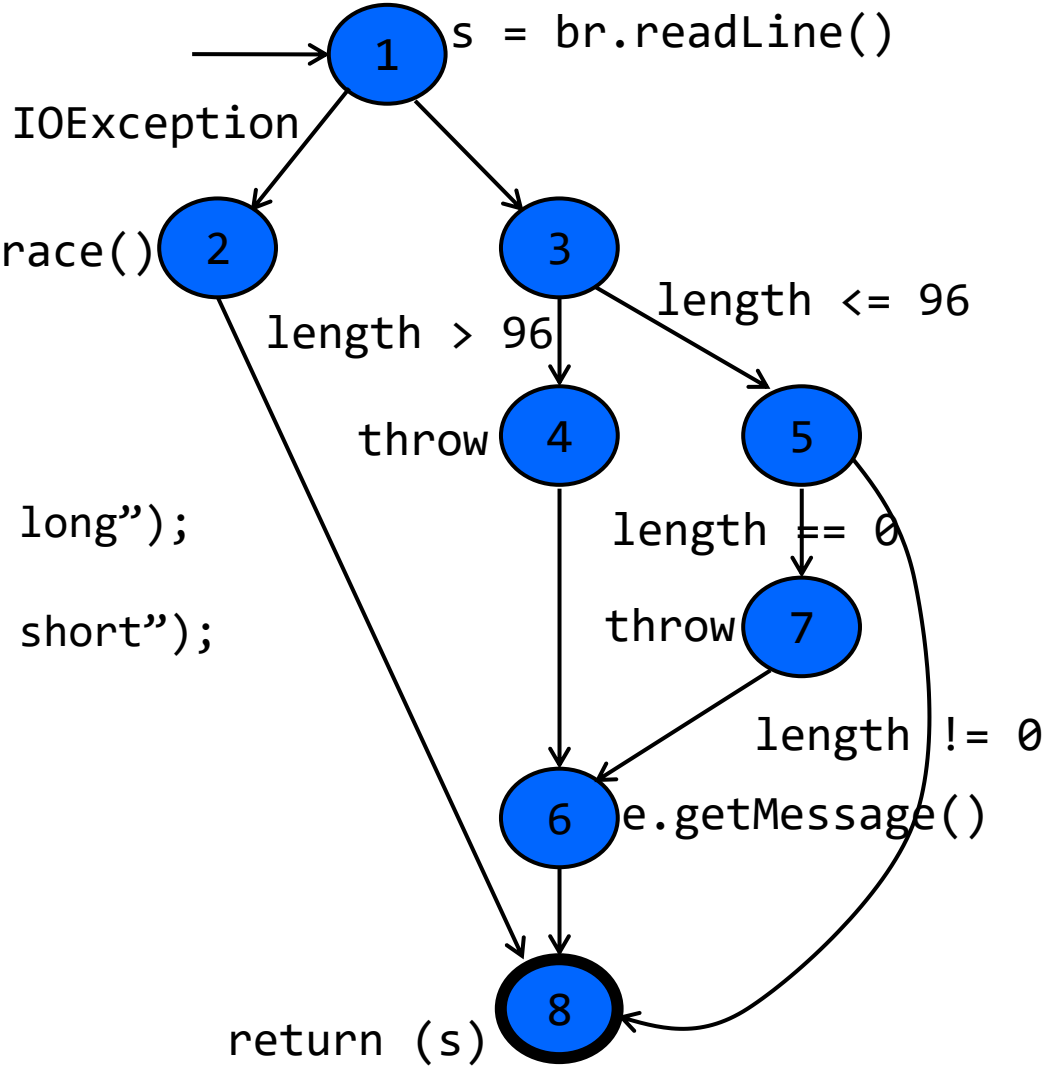
Switch-case

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



Exceptions

```
try
{
   s = br.readLine();
   if (s.length() > 96)
      throw new Exception (“too long”);
   if (s.length() == 0)
      throw new Exception (“too short”);
} (catch IOException e) {
   e.printStackTrace();
} (catch Exception e) {
   e.getMessage();
}
return (s);
```



Relevant Reads and Resources

- Recommended text:
  - Introduction to Software Testing, 2nd Edition: ch7.1 and ch7.2