Graph-Based Testing

- Introduction
- Basic Concepts
- Control Flow Testing
- Data Flow Testing
- Summary

Motivation

- Graph-based testing first builds a graph model for the program under test, and then tries to cover certain elements in the graph model.
- Graph is one of the most widely used structures for abstraction.
  - Transportation network, social network, molecular structure, geographic modeling, etc.
- Graph is a well-defined, well-studied structure
  - Many algorithms have been reported that allow for easy manipulation of graphs.

Major Steps

- Step 1: Build a graph model
  - What information to be captured, and how to represent those information?
- Step 2: Identify test requirements
  - A test requirement is a structural entity in the graph model that must be covered during testing
- Step 3: Select test paths to cover those requirements
- Step 4: Derive test data so that those test paths can be executed

Graph Models

- Control flow graph: Captures information about how the control is transferred in a program.
- Data flow graph: Augments a CFG with data flow information
- Dependency graph: Captures the data/control dependencies among program statements
- Cause-effect graph: Modeling relationships among program input conditions, known as causes, and output conditions, known as effects

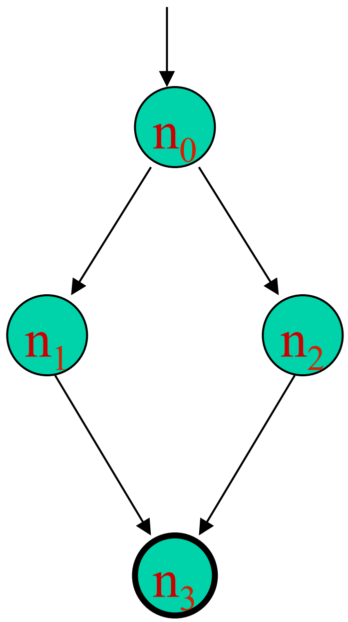
Graph-Based Testing

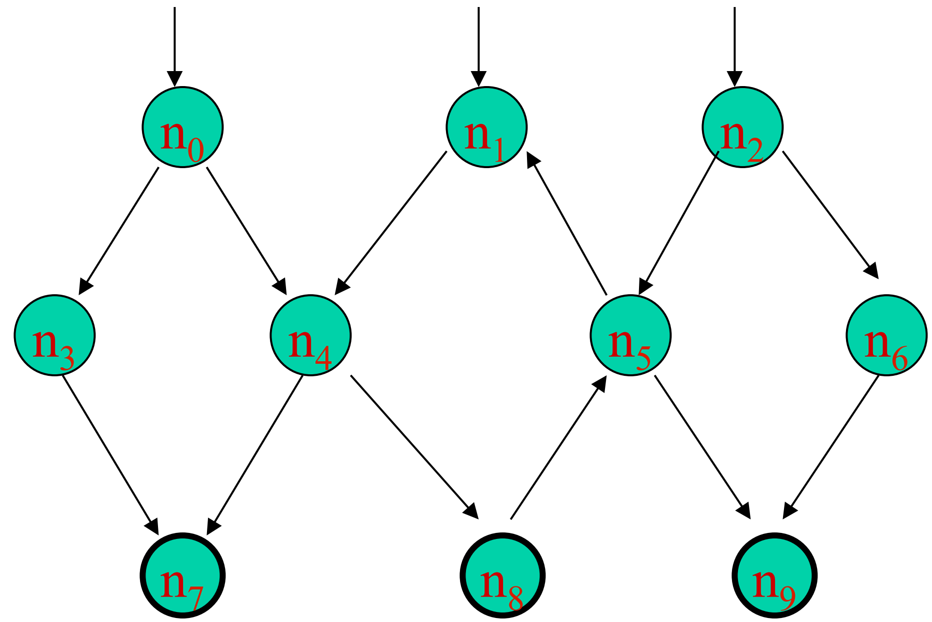
- Introduction
- Basic Concepts
- Control Flow Testing
- Data Flow Testing
- Summary

Graph

- A graph consists of a set of nodes and edges that connect pairs of nodes.
- Formally, a graph $G = <N, N_0, N_f, E)$:
  - N: a set of nodes
  - $N_0 \subseteq N$: a set of initial nodes
  - $N_f \subseteq N$: a set of final nodes
  - $E \subseteq N \times N$: a set of edges
- In our context, N, N_0, and N_f contain at least one node.

Example





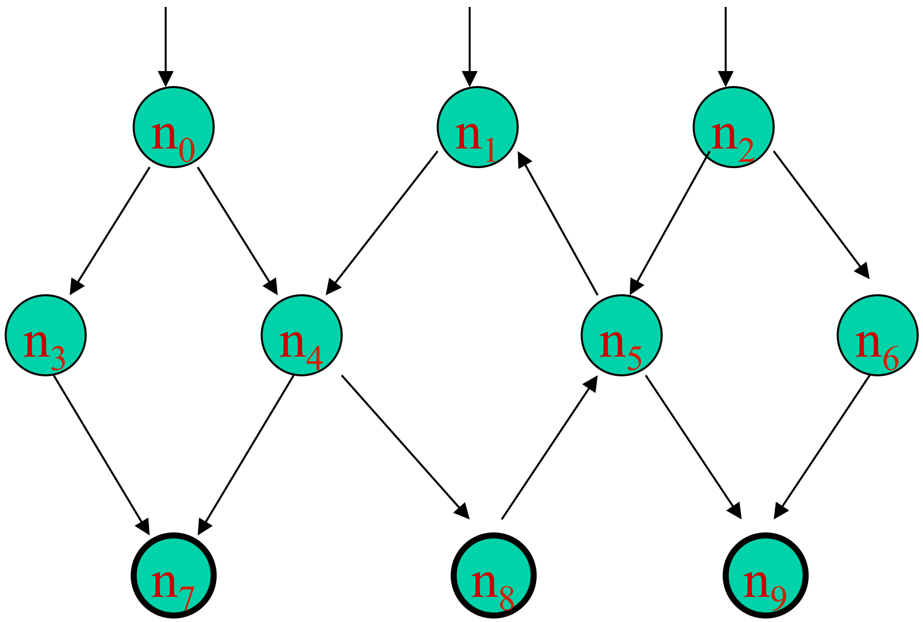
$N = \{n_0, n_1, n_2, n_3\}$
$N_0 = \{n_0\}$
$N_f = \{n_3\}$
$E = \{(n_0, n_1), (n_0, n_2), (n_1, n_3), (n_2, n_3)\}$

$N = \{n_0, n_1, n_2, n_3 , n_4, n_5, n_6 , n_7, n_8, n_9\}$
$N_0 = \{n_0 , n_1, n_2\}$
$N_f = \{n_7, n_8, n_9\}$
$E = \{(n_0, n_3), (n_0, n_4), (n_1, n_4), (n_1, n_5), \ldots\}$

Path, Subpath, Test Path

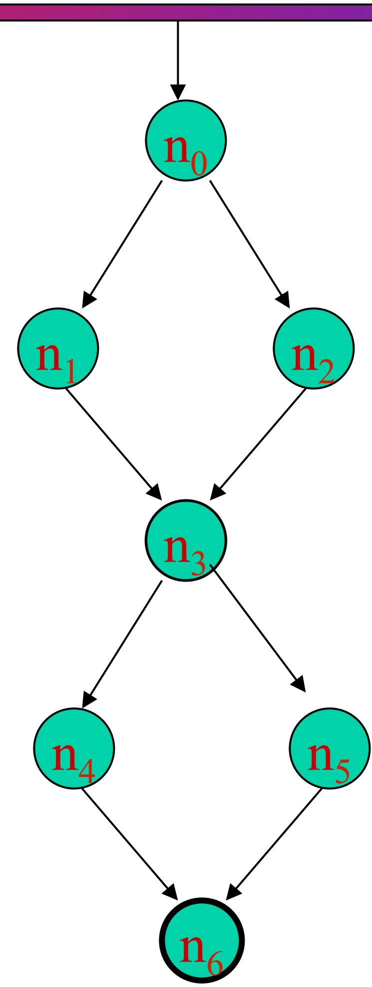
- A path is a sequence $[n_1, n_2, \ldots, n_M]$ of nodes, where each pair of adjacent nodes (n_i, n_{i+1}) is an edge.
  - The length of a path refers to the number of edges in the path
- A subpath of a path p is a subsequence of p, possibly p itself.
- A test path is a path, possibly of length zero, that starts at an initial node, and ends at a final node
  - Represents a path that is executed during a test run

Example



$p1 = [n_0, n_3, n_7]$
$p2 = [n_1, n_4, n_8, n_5, n_1]$
$p3 = [n_4, n_8, n_5]$

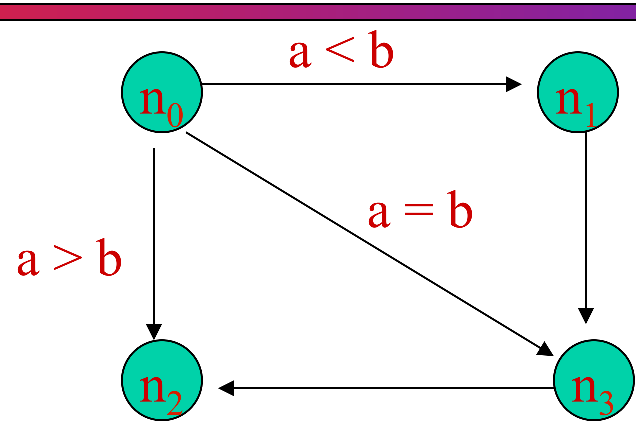
SESE Graph



Visit & Tour

- A test path p is said to visit a node n (or an edge e) if node n (or edge e) is in path p.
- A test path p is said to tour a path q if q is a subpath of p.

Test Case vs Test Path



t_1: (a = 0, b = 1) => $p_1 = [n_0, n_1, n_3, n_2]$
t_2: (a = 1, b = 1) => $p_2 = [n_0, n_3, n_2]$
t_3: (a = 2, b = 1) => $p_3 = [n_0, n_2]$

Graph-Based Testing

- Introduction
- Basic Concepts
- Control Flow Testing
- Data Flow Testing
- Summary

Basic Block

- A basic block, or simply a block, is a sequence of consecutive statements with a single entry and a single exit point.
  - If one statement is executed, all the statements must be executed.
- Control always enters a basic block at its entry point, and exits from its exit point.
  - No entry, halt, or exit inside a basic block
- If a basic block contains a single statement, then the entry and exit points coincide.

Example

```
1.  begin
2.      int x, y, power;
3.      float z;
4.      input (x, y);
5.      if (y < 0)
6.          power = -y;
7.      else
8.          power = y;
9.      z = 1;
10.     while (power != 0) {
11.         z = z * x;
12.         power = power – 1;
13.     }
14.     if (y < 0)
15.         z = 1/z;
16.     output (z);
17. end
```

Block	Lines	Entry	Exit
1	2, 3, 4, 5	2	5
2	6	6	6
3	8	8	8
4	9	9	9
5	10	10	10
6	11, 12	11	12
7	14	14	14
8	15	15	15
9	16	16	16

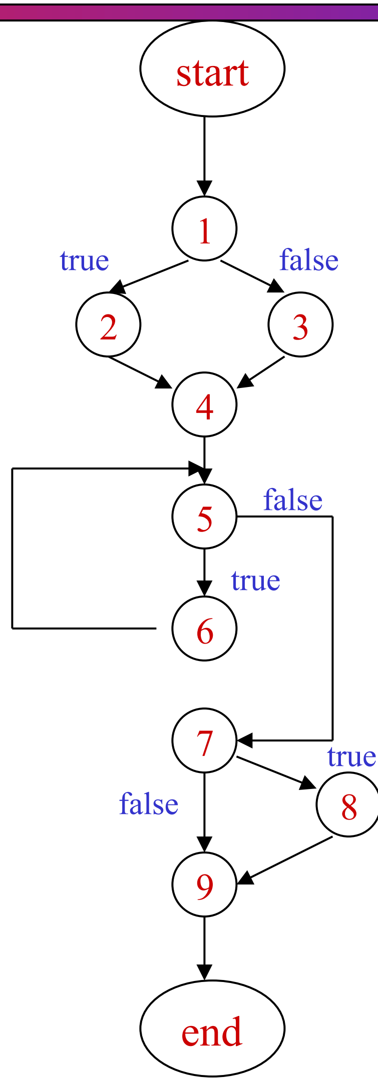
Function Calls

- Should a function call be treated like a regular statement or as a separate block of its own?
- If a function call is user-defined and needs to be tested, then the function call should be treated as a separate block. Otherwise, it could be treated like a regular statement.

Control Flow Graph

- A control flow graph is a graph with two distinguished nodes, start and end.
  - Node start has no incoming edges, and node end has no outgoing edges.
  - Every node can be reached from start, and can reach end.
- In a CFG, a node is typically a basic block, and an edge indicates the flow of control from one block to another.

Example



Reachability

- A node n is syntactically reachable from node n' if there exists a path from n' to n.
- A node n is semantically reachable from node n' if it is possible to execute a path from n' to n with some input.
- reach(n): the set of nodes and edges that can be syntactically reached from node n.

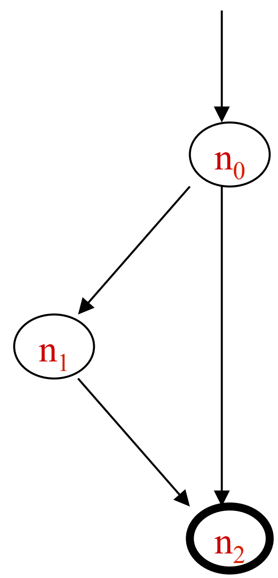
Node Coverage

- A test set T satisfies Node Coverage on graph G if and only if for every syntactically reachable node n in N, there is some path p in path(T) such that p visits n.
  - path(T): the set of paths that are exercised by the execution of T
- In other words, the set TR of test requirements for Node Coverage contains each reachable node in G.

Edge Coverage

- The TR for Edge Coverage contains each reachable path of length up to 1, inclusive, in a graph G.
- Note that Edge Coverage subsumes Node Coverage.

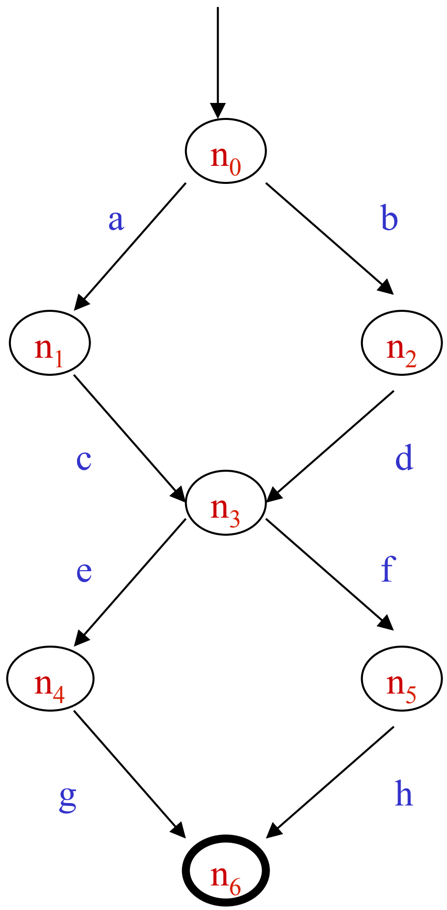
Node vs Edge Coverage



Edge-Pair Coverage

- The TR for Edge-Pair Coverage contains each reachable path of length up to 2, inclusive, in a graph G.
- This definition can be easily extended to paths of any length, although possibly with diminishing returns.

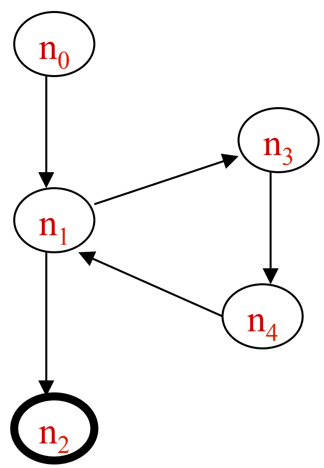
Edge-Pair vs Edge Coverage



p1 = [n0, n1, n3, n4, n5]
p2 = [n0, n2, n3, n5, n6]

Complete Path Coverage

- The TR for Complete Path Coverage contain all paths in a graph.



How many paths do we need to cover in the above graph?

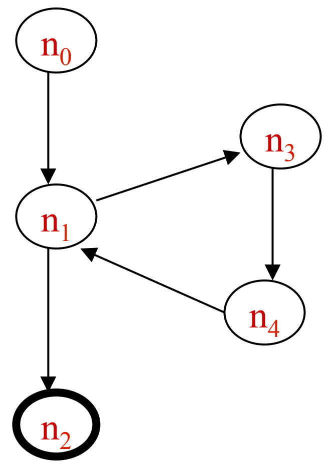
Simple & Prime Path

- A path is simple if no node appears more than once in the path, with the exception that the first and last nodes may be identical.
- A path is a prime path if it is a simple path, and it does not appear as a proper subpath of any other simple path.

Prime Path Coverage

- The TR for Prime Path Coverage contains every prime path in a graph.

Example

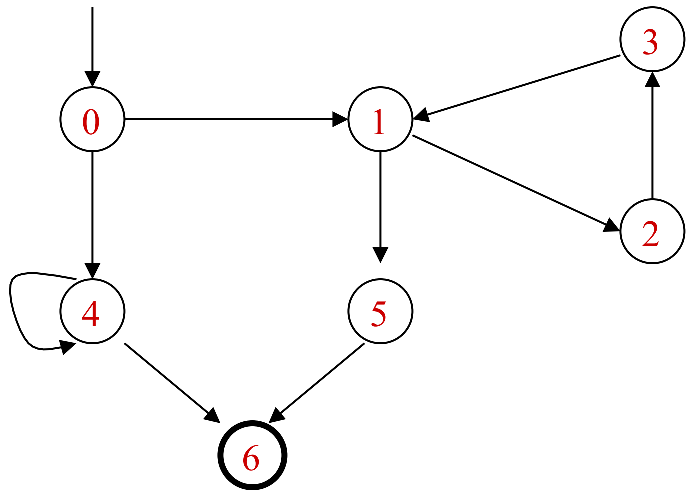


Prime paths = {[n0, n1, n2], [n0, n1, n3, n4], [n1, n3, n4, n1], [n3, n4, n1, n3], [n4, n1, n3, n4], [n3, n4, n1, n2]}
Path (t1) = [n0, n1, n2]
Path (t2) = [n0, n1, n3, n4, n1, n3, n4, n1, n2]
T = {t1, t2}

Computing Prime Paths

- Step 1: Find all the simple paths
  - Find all simple paths of length 0, extend them to length 1, and then to length 2, and so on
- Step 2: Select those that are maximal

Example



Example – Simple Paths (2)

len = 0

1. [0]
2. [1]
3. [2]
4. [3]
5. [4]
6. [5]
7. ~~[6]~~!

len = 1

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]~~!

len = 2

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]

len = 3

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]

len = 4

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

Example – 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]~~!

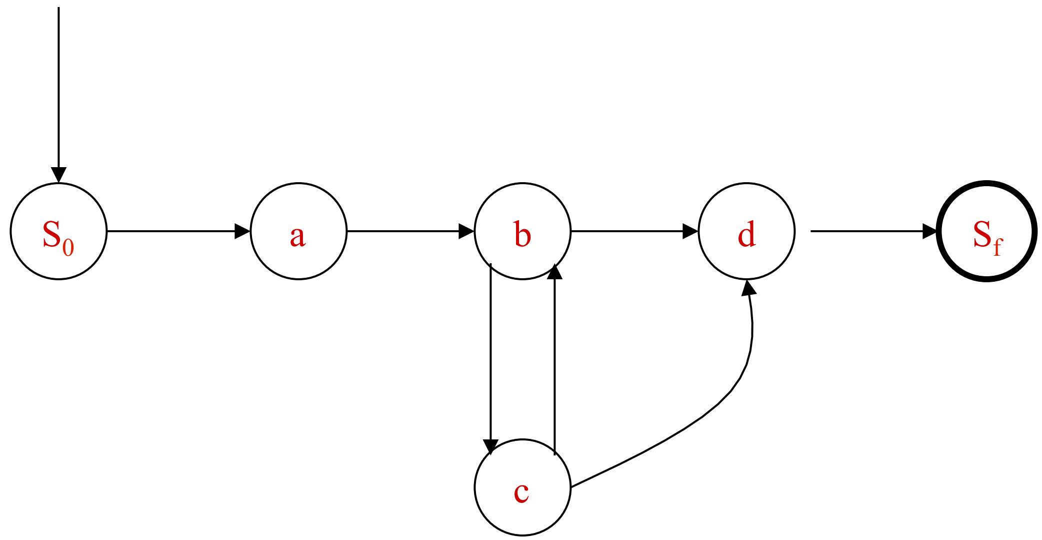
Example – Test Paths

- Start with the longest prime paths and extend them to the start and end nodes of the graph

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

Infeasible Test Requirements

- The notion of "tour" is rather strict.



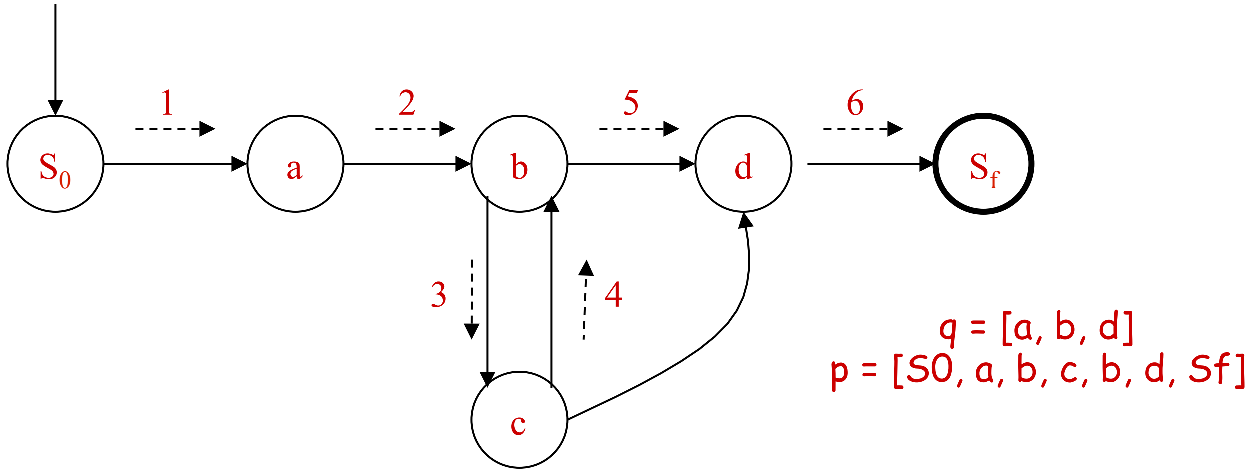
Let $q = [a, b, d]$, and $p = [S_0, a, b, c, d, S_f]$.

Does path p tour path q?

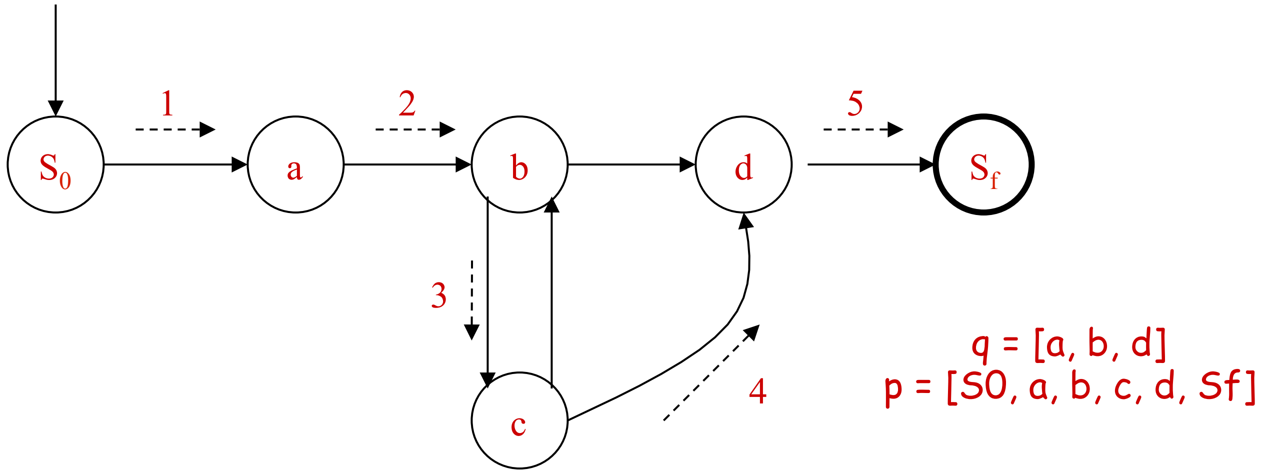
Sidetrips/Detours

- Tour: Test path p is said to tour path q if and only if q is a subpath of p.
- Tour with sidetrips: Test path p is said to tour path q with sidetrips if and only if every edge in q is also in p in the same order.
- Tour with detours: Test path p is said to tour path q with detours if and only if every node in q is also in p in the same order

Example



1
2
5
6
S0
a
b
d
Sf
3
4
c
q = [a, b, d]
p = [S0, a, b, c, b, d, Sf]



1
2
5
S0
a
b
d
Sf
3
4
c
q = [a, b, d]
p = [S0, a, b, c, d, Sf]

Best Effort Touring

- If a test requirement can be met without a sidetrip (or detour), then it should be done so.
- In other words, sidetrips or detours should be allowed only if necessary.

Graph-Based Testing

- Introduction
- Basic Concepts
- Control Flow Testing
- Data Flow Testing
- Summary

Definition/Use

- A definition is a location where a value for a variable is stored into memory.
  - Assignment, input, parameter passing, etc.
- A use is a location where a variable's value is accessed.
  - p-use: a use that occurs in a predicate expression, i.e., an expression used as a condition in a branch statement
  - c-use: a use that occurs in an expression that is used to perform certain computation

Data Flow Graph

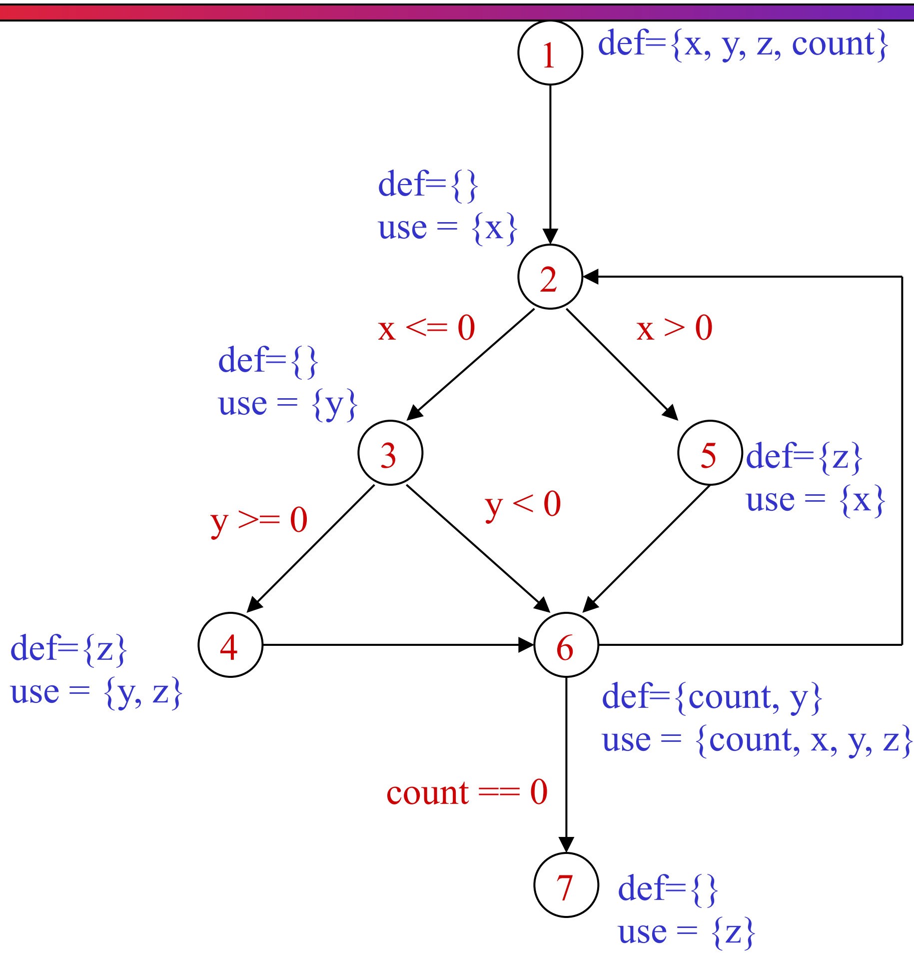
- A data flow graph (DFG) captures the flow of data in a program
- To build a DFG, we first build a CFG and then annotate each node n in the CFG with the following two sets:
  - def(n): the set of variables defined in node n
  - use(n): the set of variables used in node n

Example (1)

```
1.  begin
2.    float x, y, z = 0.0;
3.    int count;
4.    input (x, y, count);
5.    do {
6.       if (x <= 0) {
7.          if (y >= 0) {
8.             z = y * z + 1;
9.          }
10.      }
11.      else {
12.         z = 1/x;
13.      }
14.      y = x * y + z;
15.      count = count – 1;
16.   while (count > 0)
17.   output (z);
18. end
```

Node	Lines
1	2, 3, 4
2	6
3	7
4	8
5	12
6	14, 15, 16
7	17

Example (2)



1
def={x, y, z, count}
def={}
use = {x}
2
x <= 0
x > 0
def={}
use = {y}
3
5
def={z}
use = {x}
y >= 0
y < 0
def={z}
use = {y, z}
4
6
def={count, y}
use = {count, x, y, z}
count == 0
7
def={}
use = {z}

DU-pair & DU-path

- A du-pair is a pair of locations (i, j) such that a variable v is defined in i and used in j.
- Suppose that variable v is defined at node i, and there is a use of v at node j. A path $p = (i, n_1, n_2, \ldots, n_k, j)$ is def-clear w.r.t. v if v is not defined along the subpath $n_1, n_2, \ldots, n_k$.
- A definition of a variable v reaches a use of v if there is a def-clear path from the definition to the use w.r.t. v.
- A du-path for a variable v is a simple path from a definition of v to a use of v that is def-clear w.r.t. v.

Example

- Consider the previous example:
  - Path p = (1, 2, 5, 6) is def-clear w.r.t variables x, y and count, but is not def-clear w.r.t. variable z.
  - Path q = (6, 2, 5, 6) is def-clear w.r.t variables count and y.
  - Path r = (1, 2, 3, 4) is def-clear w.r.t variables y and z.

Notations

- Def-path set du(n, v): the set of du-paths w.r.t variable v that start at node n.
- Def-pair set du(n, n', v): the set of du-paths w.r.t variable v that start at node n and end at node n'.
- Note that $du(n, v) = \cup_{n'} du(n, n', v)$.

All-Defs Coverage

- For each def-path set S = du(n, v), the TR for All-Defs Coverage contains at least one path in S.
- Informally, for each def, we need to tour at least one path to at least one use.

All-Uses Coverage

- For each def-pair set $S = du(n, n', v)$, the TR for All-Uses Coverage contains at least one path in S.
- Informally, it requires us to tour at least one path for every def-use pair.

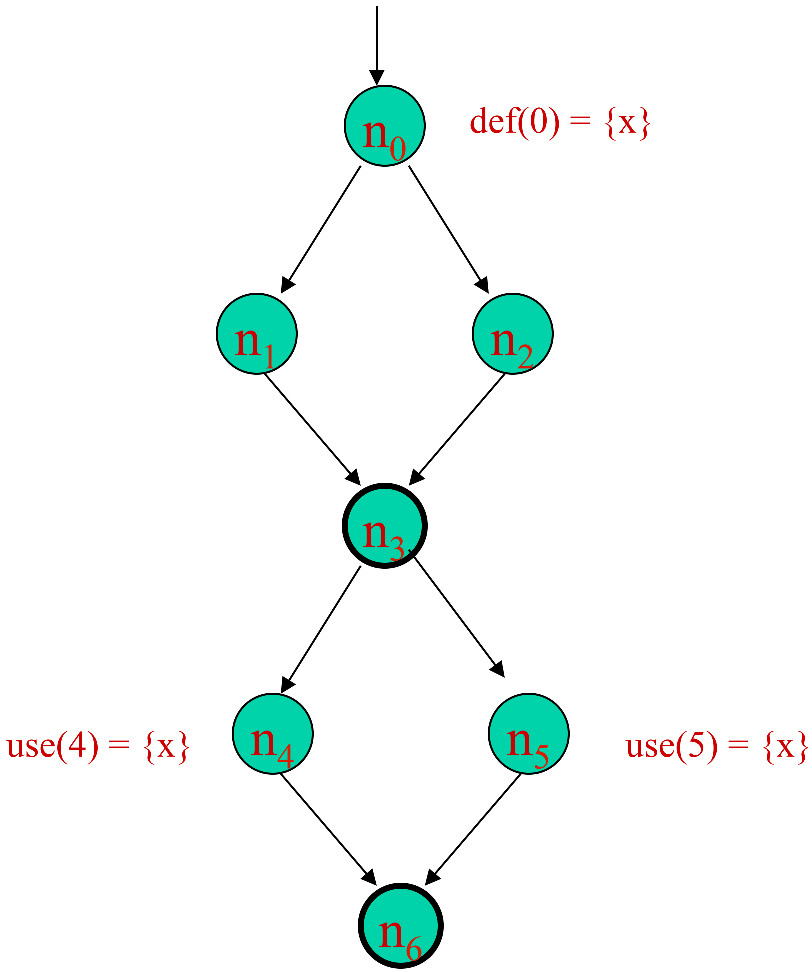
All-DU-Paths Coverage

- For each def-pair set S = du(n, n', v), the TR for All-DU-Paths Coverage contains every path in S.
- Informally, this requires to tour every du-path.

Best Effort Touring

- When we allow touring with sidetrip/detour, the sidetrip/detour must be def-clear.

Example



all-defs
0-1-3-4

all-uses
0-1-3-4
0-1-3-5

all-du-paths
0-1-3-4
0-1-3-5
0-2-3-4
0-2-3-5

Why data flow?

- Consider the previous example. Assume that there is a fault in line 14, which is supposed to be y = x + y + z.
- Does the following test set satisfy edge coverage? Can the test set detect the above fault?

	x	y	count	EO	AO
t1	-2	2	1	1	1
t2	-2	-2	1	0	0
t3	2	2	1	1/2	1/2
t4	2	2	2	1/2	1/2

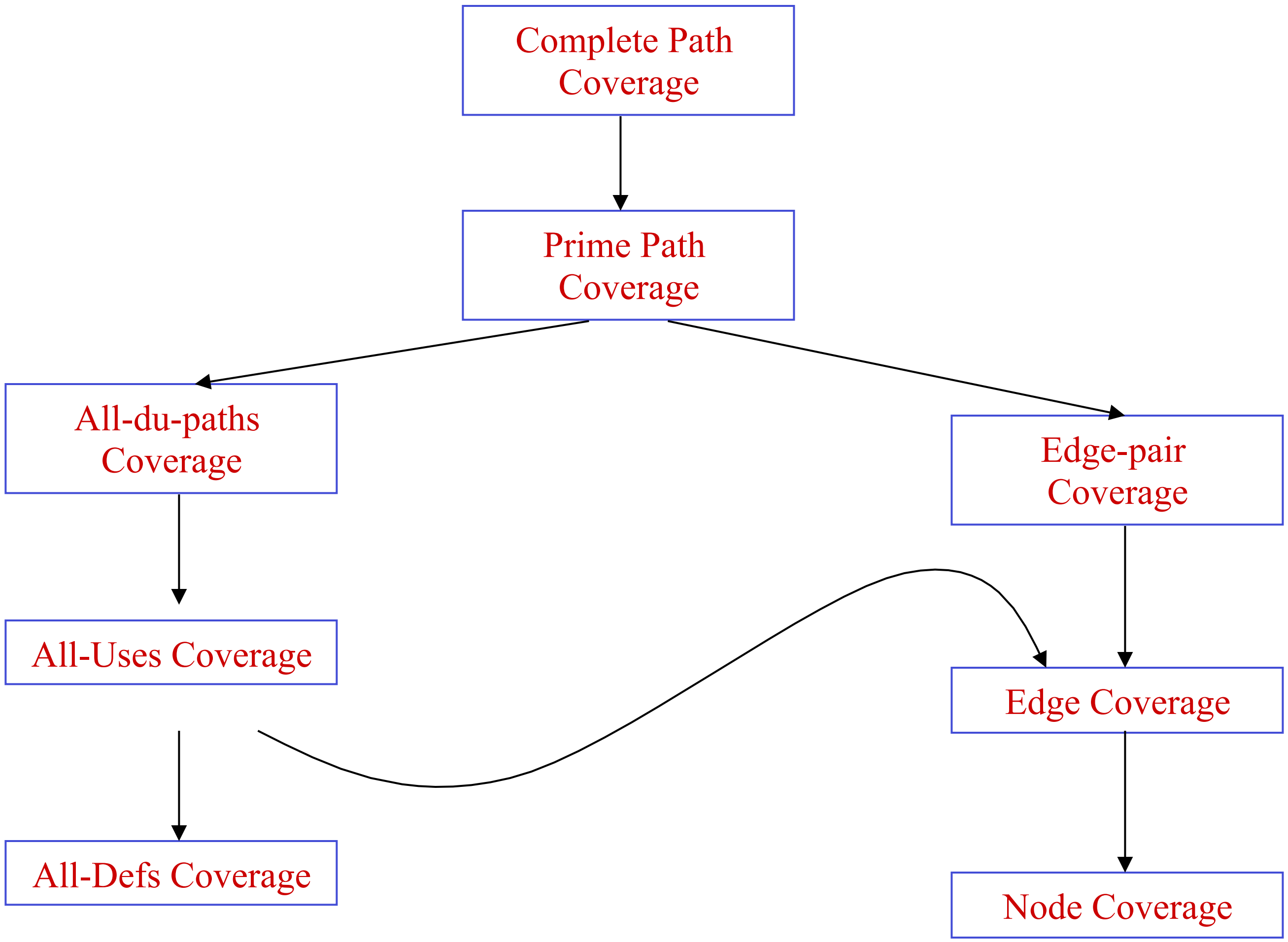
EO: Expected output, i.e. line 14: y = x + y + z

AO: Actual output, i.e. line 14: y = x * y + z

Graph-Based Testing

- Introduction
- Basic Concepts
- Control Flow Testing
- Data Flow Testing
- Summary

Subsumption Hierarchy



Recap

- Graph provides a good basis for systematic test selection.
- Control flow testing focuses on the transfer of control, while data flow testing focuses on the definitions of data and their subsequent use.
- Control flow coverage is defined in terms of nodes, edges, and paths; data flow coverage is defined in terms of def, use, and du-path.