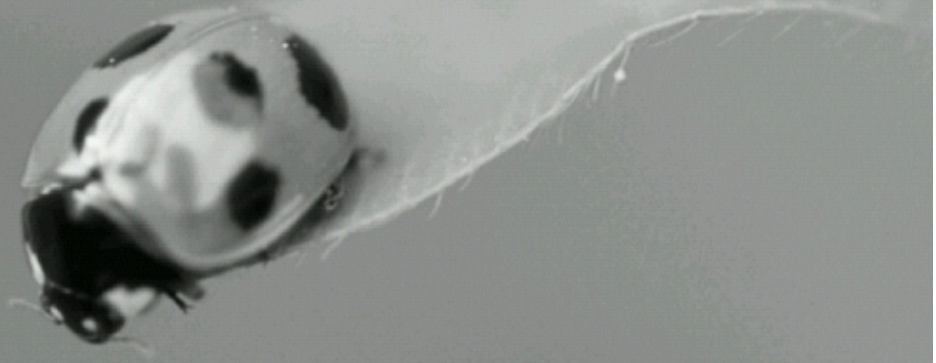
ECE453/SE465/CS447/ECE653/CS647:

Structural Coverage



Lin Tan

January 3, 2016

How would you test this program?

floor(x) is the largest integer not greater than x.

```
public class Floor{
    public static int Floor(int x, int y) {
        // effects:  return floor(max(x,y)/min(x,y))
        if (x > y)
                return x/y;
        else
                return y/x;
    }

}
```

Testing

- Static Testing [at compile time]
  - Static Analysis
  - Review
    - Walk-through [informal]
    - Code inspection [formal]
- Dynamic Testing [at run time]
  - Black-box testing
  - White-box testing
- Commonly, testing refers to dynamic testing.

Complete Testing?

- Poorly defined terms: “complete testing”, “exhaustive testing”, “full coverage”
- The number of potential inputs are infinite.
- Impossible to completely test a nontrivial system
  - Practical limitations: Complete testing is prohibitive in time and cost [e.g., 30 branches, 50 branches, ...]
  - Theoretical limitations: e.g. Halting problem
- Need testing criteria

Test Case

- Test Case: [informally]
  - What you feed to software; and
  - What the software should output in response.

- Test Set: A set of test cases

- Test Case: test case values, expected results, prefix values, and postfix values necessary to evaluate software under test

- Expected Results: The result that will be produced when executing the test if and only if the program satisfies its intended behaviour

Anatomy of a Test Case

- Consider testing a cellphone from the off state:

$$\underbrace{\langle \text{ on } \rangle}_{\text{prefix values}} \quad \underbrace{1\ 519\ 888\ 4567}_{\text{test case values}} \quad \underbrace{\underbrace{\langle \text{ talk } \rangle}_{\text{verification values}} \quad \underbrace{\langle \text{ end } \rangle}_{\text{exit codes}}}_{\text{postfix values}}$$

- Test Case Values: The input values necessary to complete some execution of the software under test.
  - Traditionally called Test Case
- Prefix Values: inputs to prepare software for test case values.
- Postfix Values: inputs for software after test case values;
  - Verification values: inputs to show results of test case values;
  - Exit commands: inputs to terminate program or to return it to initial state.

Test Requirement & Coverage Criterion

- Test Requirement: A test requirement is a specific element of a software artifact that a test case must satisfy or cover.
  - Ice cream cone flavors: vanilla, chocolate, mint
  - One test requirement: test one chocolate cone
  - TR denotes a set of test requirements

- A coverage criterion is a rule or collection of rules that impose test requirements on a test set.
  - Coverage criterion is a recipe for generating TR in a systematic way.
  - Flavor criterion [cover all flavors] **More general**
  - TR = {flavor=chocolate, flavor=vanilla, flavor=mint}

Measure test sets

- How to know how good a test set is?
  - Testing a ice cream stand:
  - Test set 1:
    - 3 chocolate cones, 1 vanilla cone
  - Test set 2:
    - 1 chocolate cone, 1 vanilla cone, 1 mint cone

Coverage

- Coverage: Given a set of test requirements TR for a coverage criterion C, a test set T satisfies C iff for every test requirement tr $\in$ TR, at least one t $\in$ T satisfies tr.

Infeasible Test Requirements

```
if (false)
  unreachableCall();
```

Real code from the Linux kernel:

```
while (0)
  {local_irq_disable();}
```

Statement criterion cannot be satisfied for many programs.

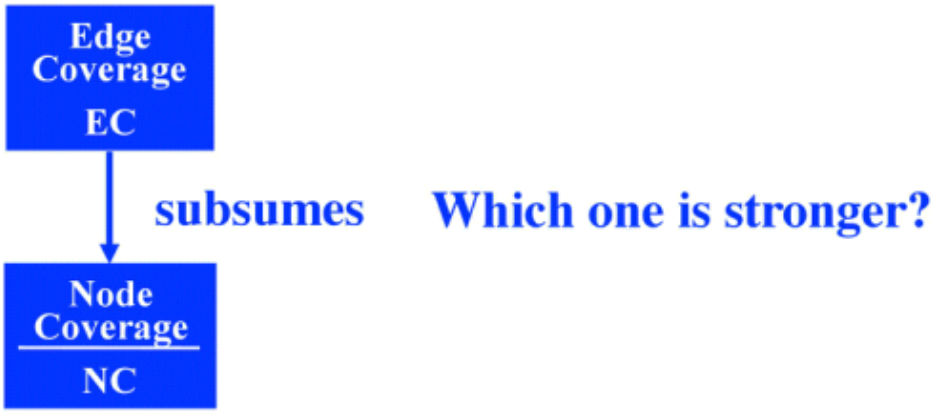
Coverage Level

- Coverage Level: Given a set of test requirements TR and a test set T, the coverage level is the ratio of the number of test requirements satisfied by T to the size of TR.
  - TR = {flavor=chocolate, flavor=vanilla, flavor=mint}
  - Test set 1 T1 = {3 chocolate cones, 1 vanilla cone}
  - Coverage Level = 2/3 = 66.7%
- Coverage levels help us evaluate the goodness of a test set, especially in the presence of infeasible test requirements.

Subsumption

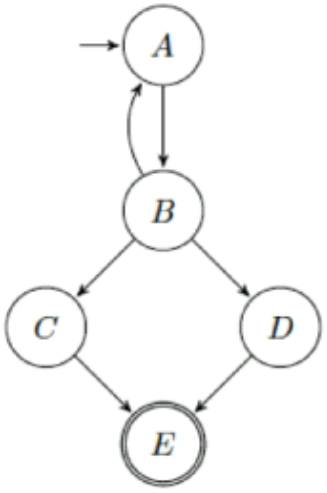
- Criteria Subsumption: A test criterion C1 subsumes C2 if and only if **every** set of test cases that satisfies criterion C1 also satisfies C2

- Must be true for every set of test cases



- Subsumption is a rough guide for comparing criteria, although it's hard to use in practice.

Graph Coverage



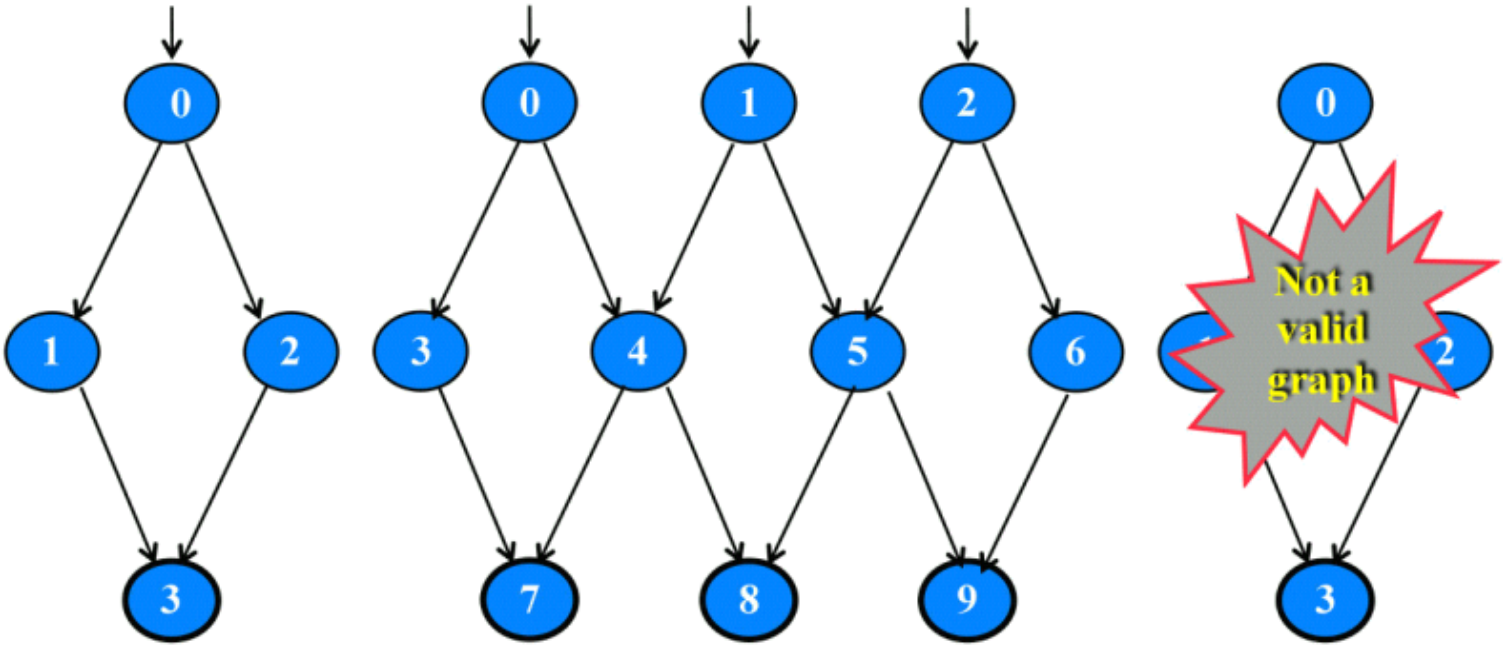
N : Set of nodes $\{A, B, C, D, E\}$

N_0 : Set of initial nodes $\{A\}$

N_f : Set of final nodes $\{E\}$

$E \subseteq N \times N$: Edges, e.g. (A, B) and (C, E);
C is the predecessor and E is the successor in (C, E).

Three Example Graphs



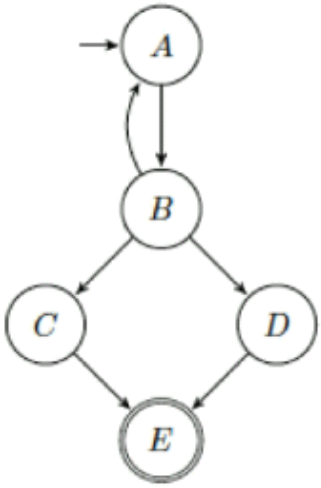
$N_0 = \{ 0 \}$ $N_f = \{ 3 \}$

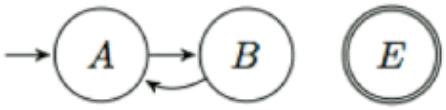
$N_0 = \{ 0, 1, 2 \}$ $N_f = \{ 7, 8, 9 \}$

$N_0 = \{ \}$ $N_f = \{ 3 \}$

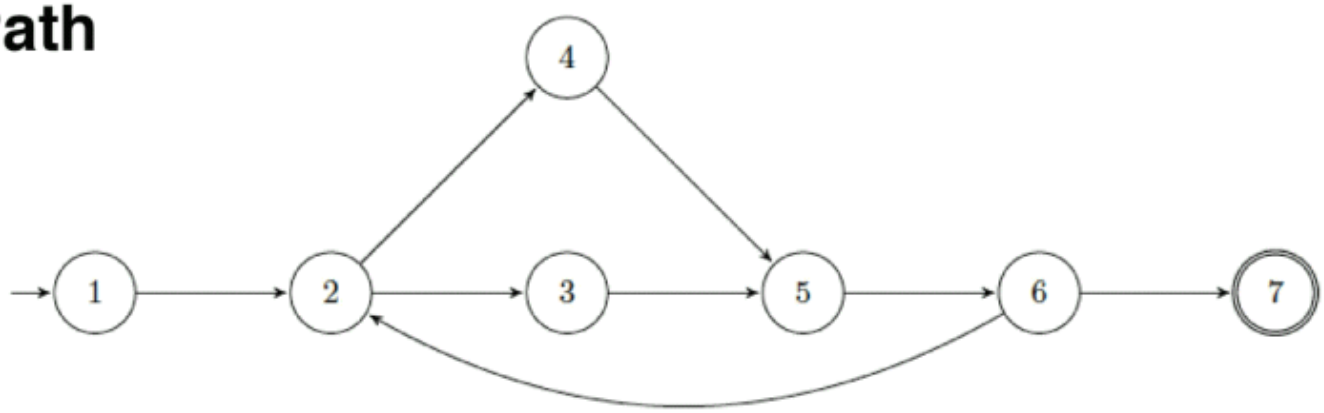
Subgraph

Subgraph: Let G' be a subgraph of G; then the nodes of G' must be a subset N_{sub} of N. Then the initial nodes of G' are $N_0 \cap N_{\text{sub}}$ and its final nodes are $N_f \cap N_{\text{sub}}$. The edges of G' are $E \cap (N_{\text{sub}} \times N_{\text{sub}})$.





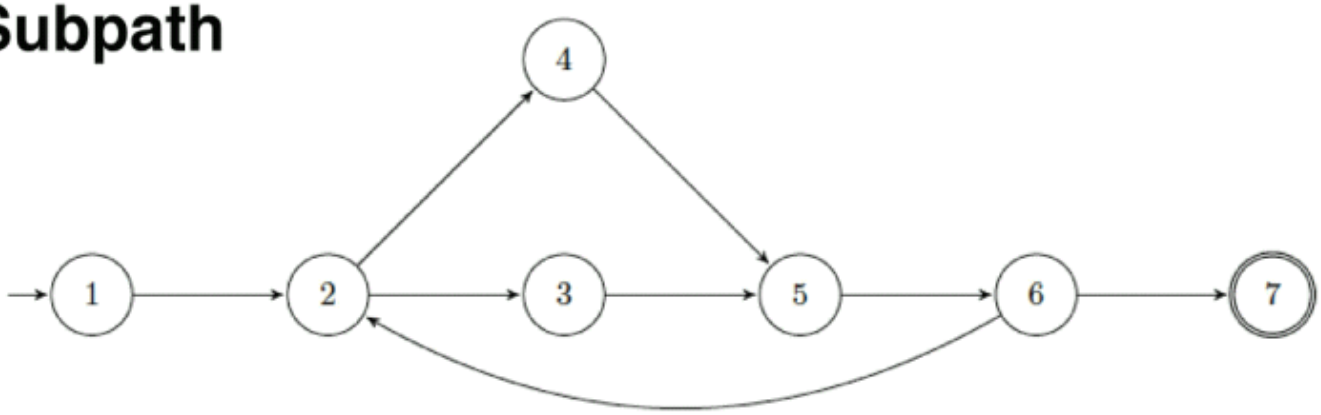
Path



- path 1: $[2, 3, 5]$, with length 2.
- path 2: $[1, 2, 3, 5, 6, 2]$, with length 5.
- not a path: $[1, 2, 5]$.

- A path is a sequence of nodes from a graph G whose adjacent pairs all belong to the set of edges E of G.

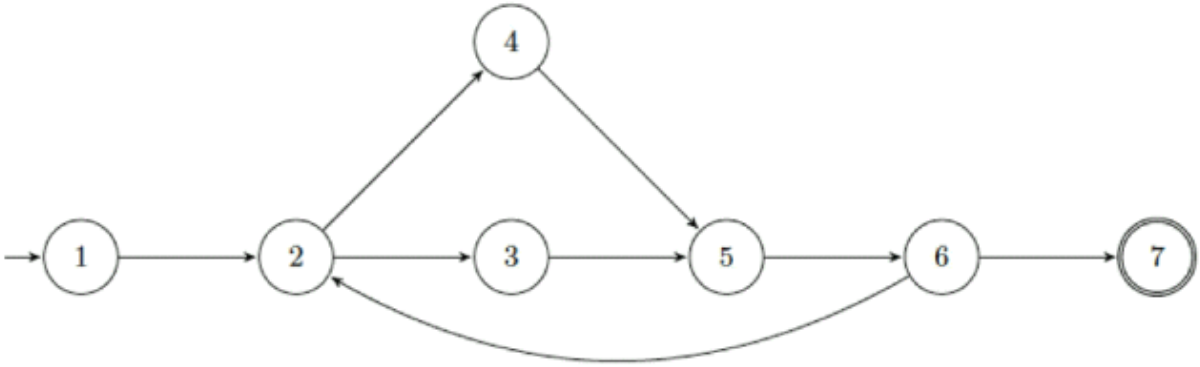
Subpath



- path 1: $[2, 3, 5]$, with length 2.
- path 2: $[1, 2, 3, 5, 6, 2]$, with length 5.
- not a path: $[1, 2, 5]$.

- A subpath is a subsequence of a path.
  - This textbook definition is ambiguous.
- Is [1,2,5] a subpath of [1,2,3,5,6,2]?

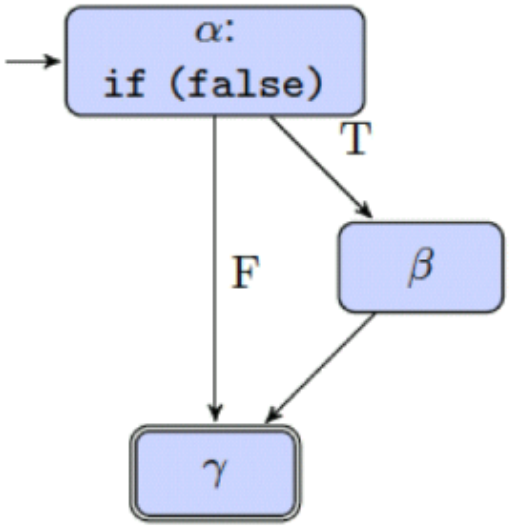
Test Path



- Test path examples:
  - [1, 2, 3, 5, 6, 7]
  - [1, 2, 3, 5, 6, 2, 3, 5, 6, 7]

- A test path is a path p [possibly of length 0] that starts at some node in N_0 and ends at some node in N_f.

Paths & Semantics



β is never executed!

- Some paths in a control flow graph may not correspond to program *semantics*.
- In this course, we generally only talk about the *syntax* of a graph -- its nodes and edges -- and not its *semantics*.

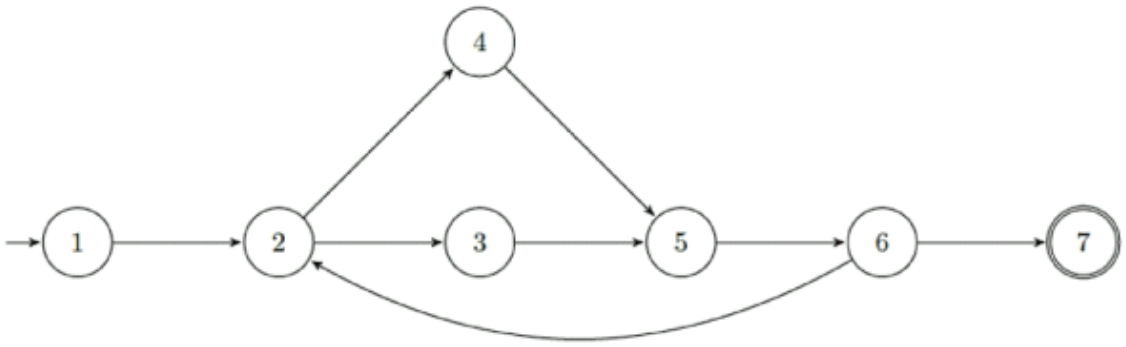
Syntactical and Semantic Reachability

- A node n is *syntactically* reachable from ni if there exists a path from ni to n.
- A node n is *semantically* reachable if one of the paths from ni to n can be reached on some input.

- Standard graph algorithms, like breadth-first search and depth-first search, can compute syntactic reachability.
- Semantic reachability is undecidable.

Reachability

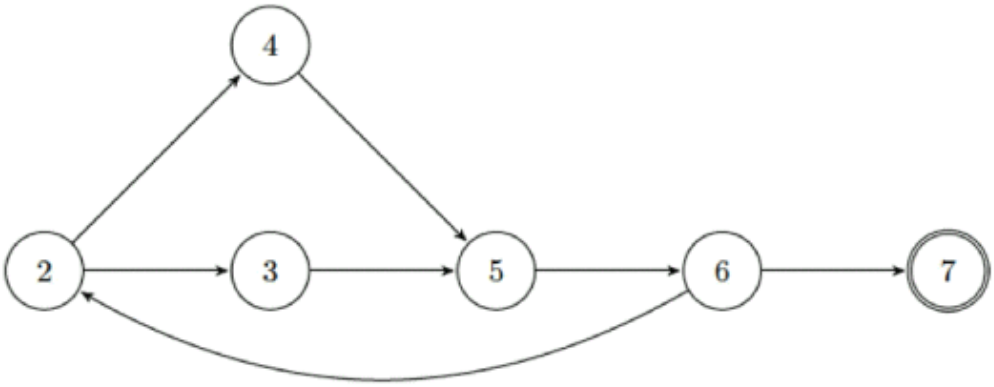
- We define $\text{reach}_G(\chi)$ as the subgraph that is syntactically reachable from χ.
  - χ is a node, an edge, or a set of nodes or edges.



$\text{reach}_G[1]$ is G in this example.

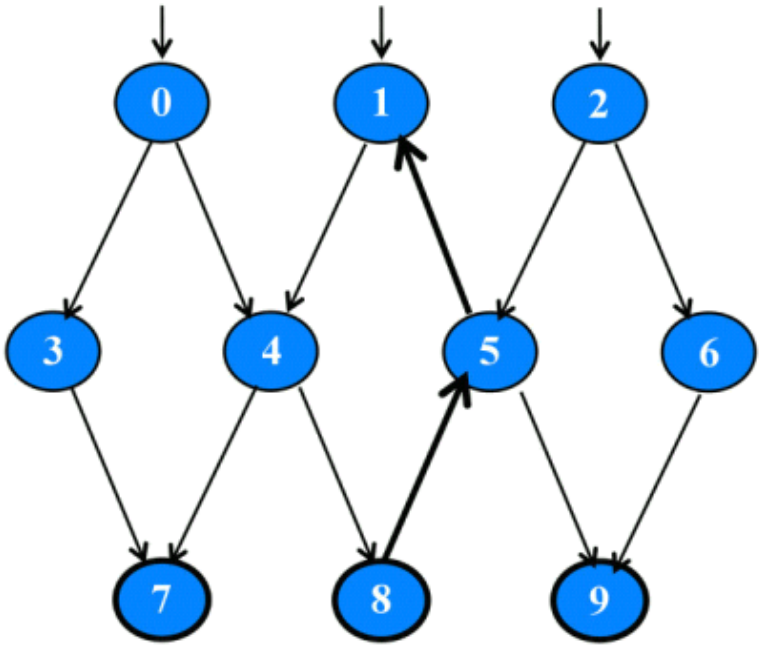
Syntactical Reachability

- $\text{reach}_{G\#}(2)$ is the subgraph that is syntactically reachable from node 2.



- $\text{reach}_{G\#}(7)$ is:

Reachability Example



Reach [0] = { 0, 3, 4, 7, 8, 5, 1, 9 }

Reach [{0, 2}] = G

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

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

© Ammann & Offutt

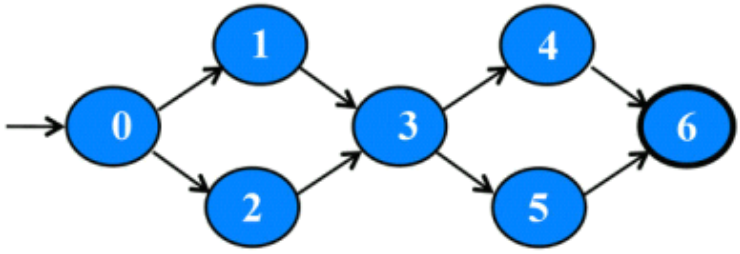
$\mathbf{Reach}_G[N_0]$

- When we talk about the nodes or edges in a graph G in a coverage criterion, we'll generally mean $\mathbf{Reach}_G[N_0]$.

- The unreachable nodes tend to
  - be uninteresting; and
  - frustrate coverage criteria.

SESEs

- **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.**



- p *visits* node 3 and edge [0, 1]
- $3 \in p$
- $[0, 1] \in p$

Double-diamond graph
Four test paths
p=[0, 1, 3, 4, 6]
[0, 1, 3, 5, 6]
[0, 2, 3, 4, 6]
[0, 2, 3, 5, 6]

- p0 = [1, 3, 4], p0 is a subpath of p, and conversely, p *tours* p0.
- Any path tours itself.

Connect Test Cases and Test Paths

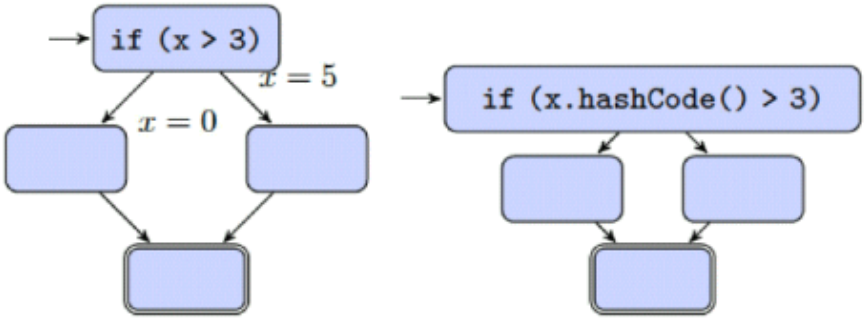
- Connect test cases and test paths with a mapping *path*$_G$ from test cases to test paths
  - e.g., *path*$_G$*[t]* is the set of test paths corresponding to test case t.
  - Usually just write *path,* as G is obvious from the context
  - Lift the definition of path to test set T by defining path(T)

$$\text{path}(T) = \{\text{path}(t) | t \in T\}.$$

  - Each test case gives at least one test path. If the software is deterministic, then each test case gives exactly one test path; otherwise, multiple test cases may arise from one test path.

Deterministic and Nondeterministic CFG

Here's an example of deterministic and nondeterministic control-flow graphs:



Causes of nondeterminism include dependence on inputs; on the thread scheduler; and on memory addresses, for instance as seen in calls to the default Java `hashCode()` implementation.

Nondeterminism makes it hard to check test case output, since more than one output might be a valid result of a single test input.