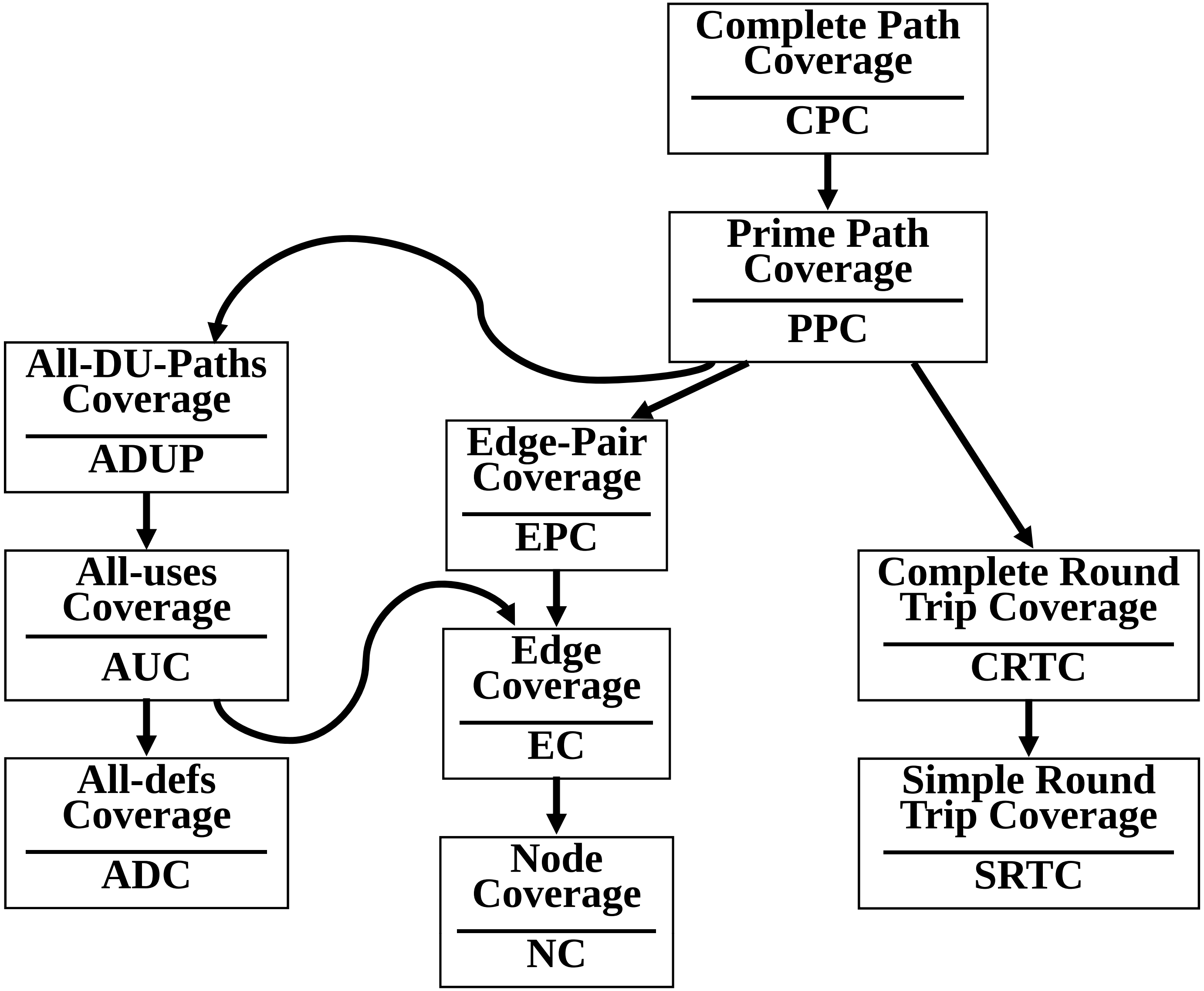
Overview Graph Coverage Criteria
(Introduction to Software Testing Chapter 2.1, 2.2)

Paul Ammann & Jeff Offutt

KAIST

Graph Coverage Criteria Subsumption



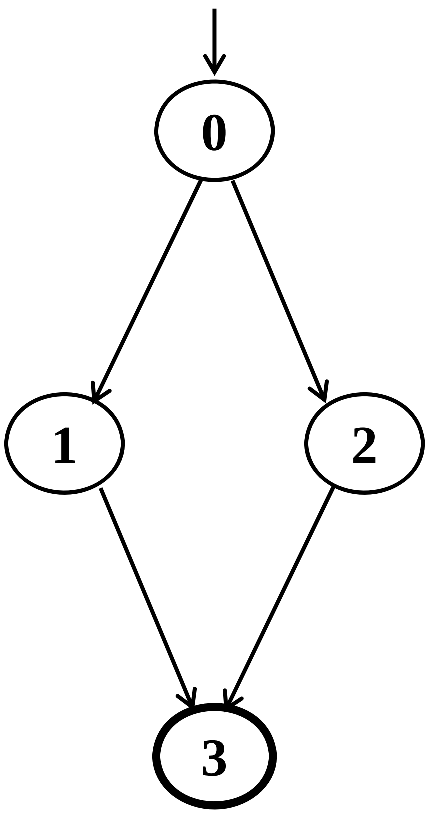
Covering Graphs (2.1)

- Graphs are the most commonly used structure for testing
- Graphs can come from many sources
  - Control flow graphs
  - Design structure
  - FSMs and statecharts
  - Use cases
- Tests usually are intended to “cover” the graph in some way

Definition of a Graph

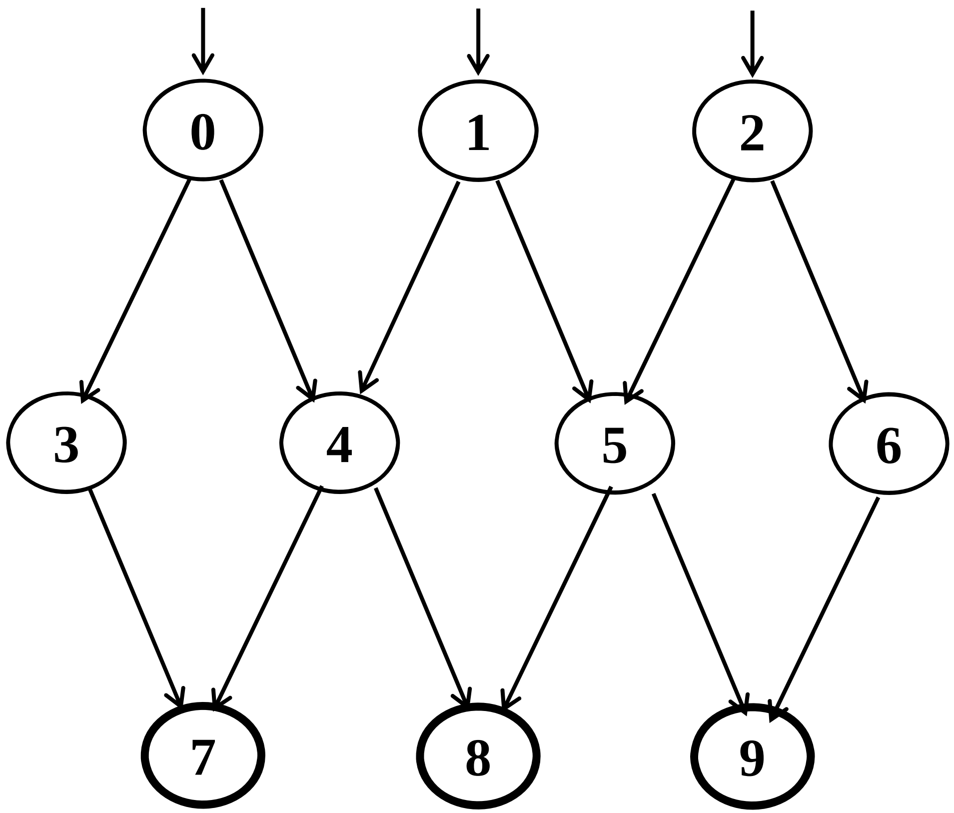
- A set N of nodes, N is not empty
- A set N_0 of initial nodes, N_0 is not empty
- A set N_f of final nodes, N_f is not empty
- A set E of edges, each edge from one node to another
  - (n_i , n_j), i is predecessor, j is successor

Three Example Graphs



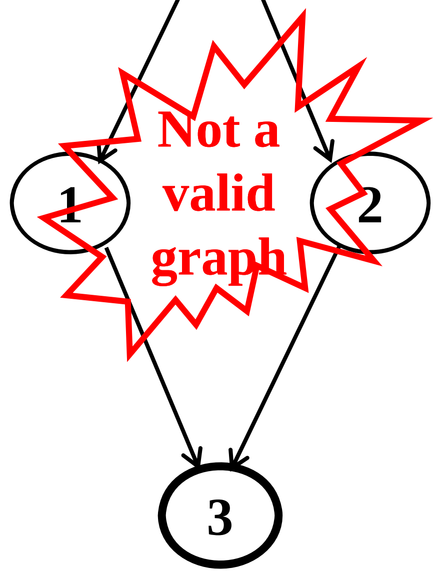
$N_0 = \{ 0 \}$

$N_f = \{ 3 \}$



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

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

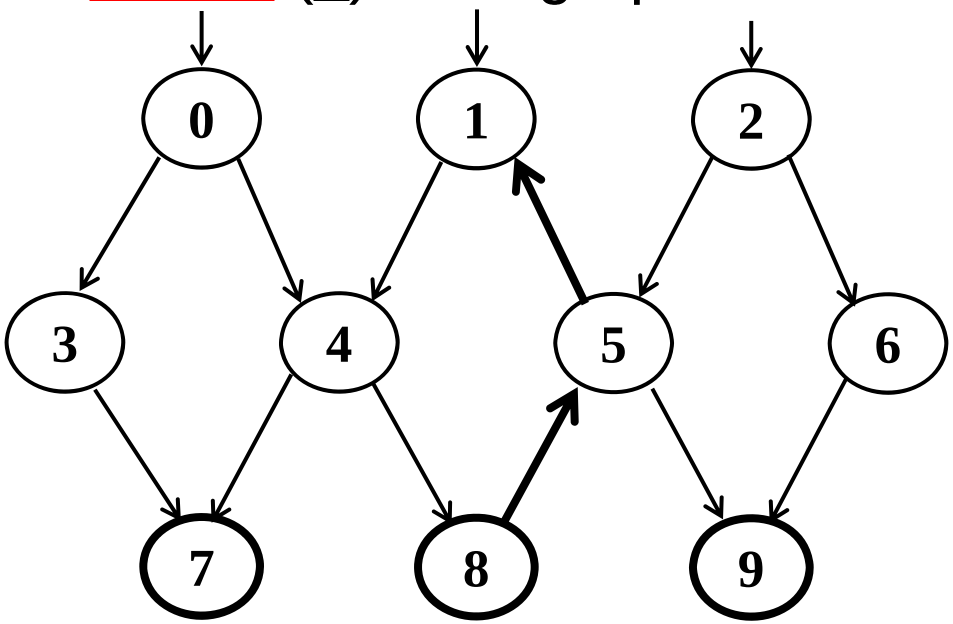


$N_0 = \{ \}$

$N_f = \{ 3 \}$

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*



Paths

[0, 3, 7]

[1, 4, 8, 5, 1]

[2, 6, 9]

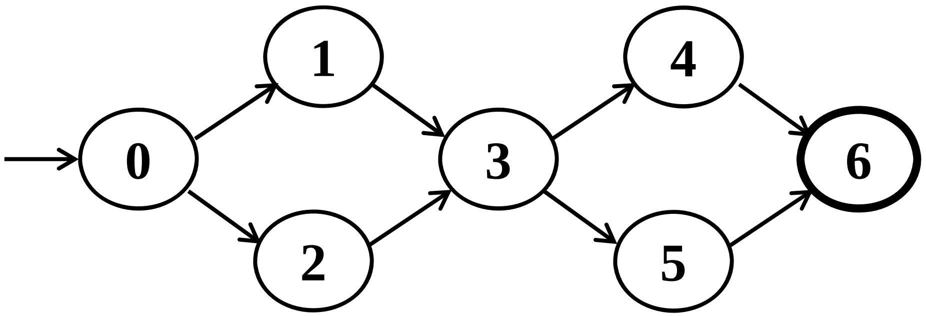
Reach (0) = { 0, 3, 4, 7, 8, 5, 1, 9 }

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 any tests
- 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



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)

KAIST

Tests and Test Paths

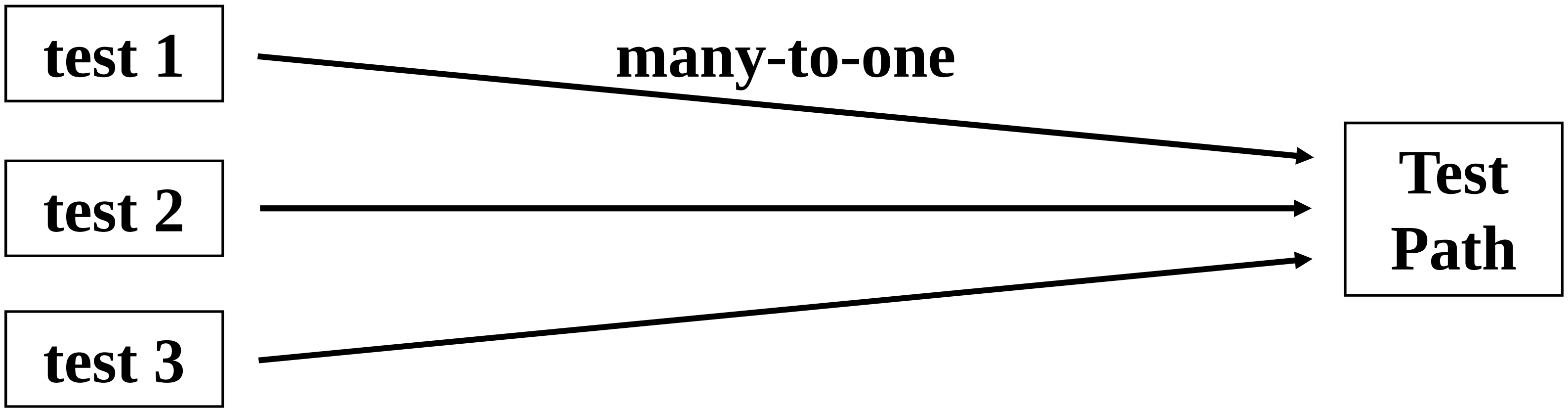
- path (*t*) : The test path executed by test *t*

- path (*T*) : The set of test paths executed by the set of tests *T*

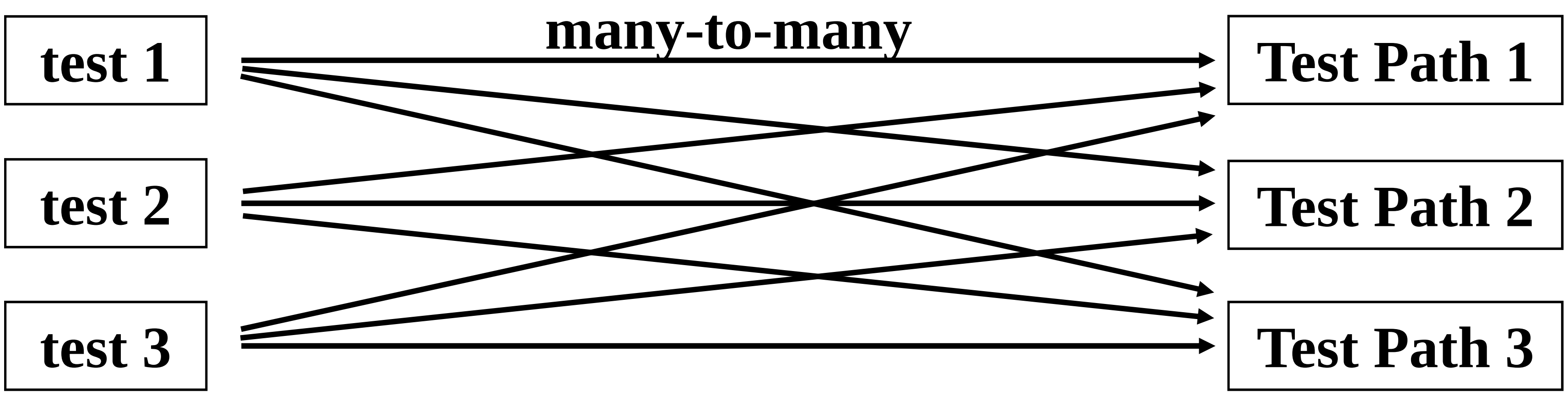
- Each test executes one and only one test path
- A location in a graph (node or edge) can be reached from another location if there is a sequence of edges from the first location to the second
  - *Syntactic reach* : A subpath exists in the graph
  - *Semantic reach* : A test exists that can execute that subpath

KAIST

Tests and Test Paths



Deterministic software – a test always executes the same test path



Non-deterministic software – a test can execute different test paths

Testing and Covering Graphs (2.2)

- 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***

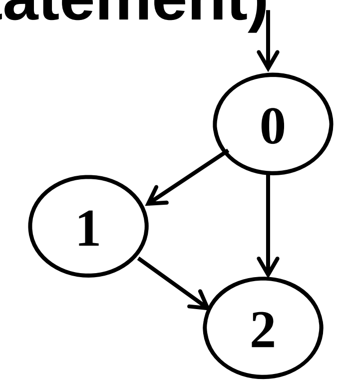
- Structural Coverage Criteria : **Defined on a graph just in terms of nodes and edges**
- Data Flow Coverage Criteria : **Requires a graph to be annotated with references to variables**

Node and Edge Coverage

- 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 "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)**

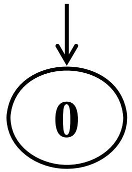


<u>Node Coverage</u> : TR = { 0, 1, 2 }
Test Path = [0, 1, 2]

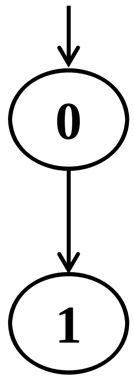
<u>Edge Coverage</u> : TR = { (0,1), (0, 2), (1, 2) }
Test Paths = [0, 1, 2]
[0, 2]

Paths of Length 1 and 0

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



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

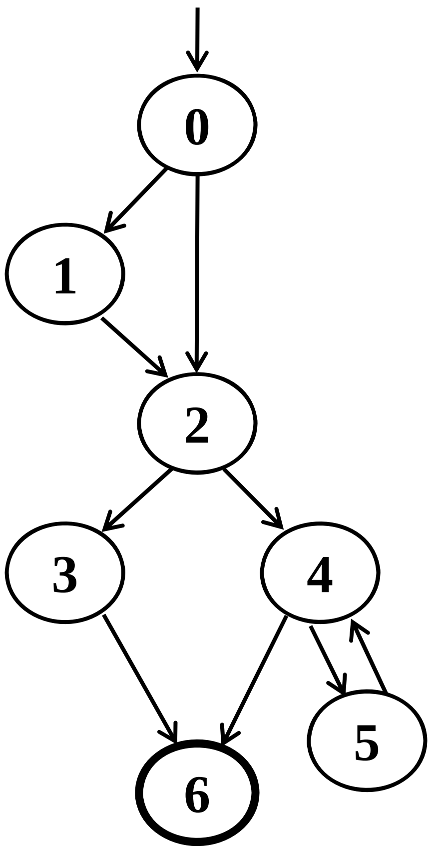
- **The logical extension is to require all paths …**

> 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_{NC} = { 0, 1, 2, 3, 4, 5, 6 }

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

Edge Coverage

TR_{EC} ={(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_{EPC} = { [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

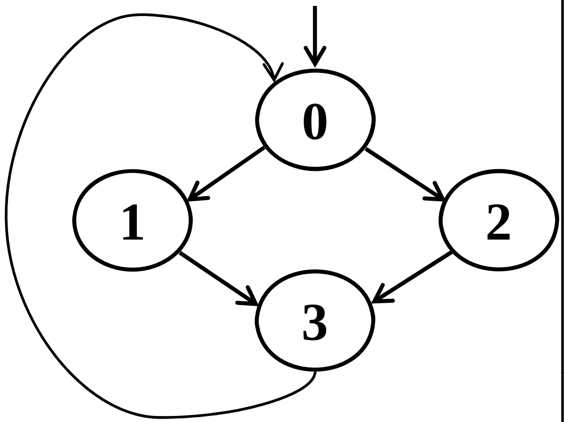
- Thus, CPC is not feasible

- 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
  - Includes all other subpaths
  - 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

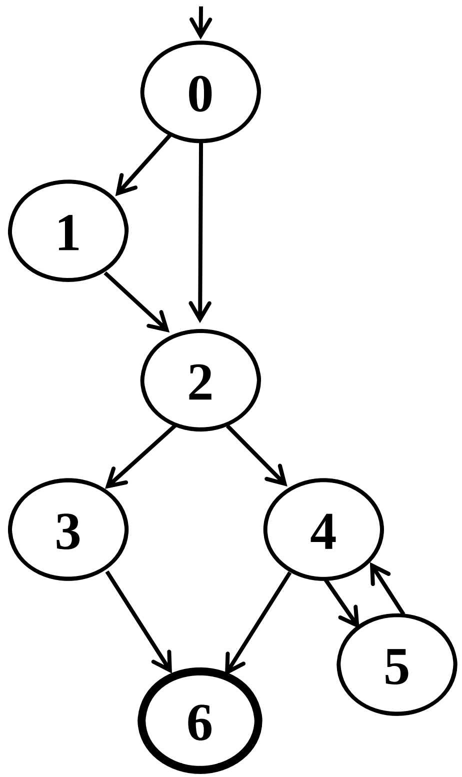
- A simple, elegant and finite criterion that requires loops to be executed as well as skipped

<u>Prime Path Coverage (PPC)</u> : TR contains each prime path in G.

- **Will tour all paths of length 0, 1, …**
- **That is, it subsumes node, edge, and edge-pair coverage**

Prime Path Example

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



Prime Paths

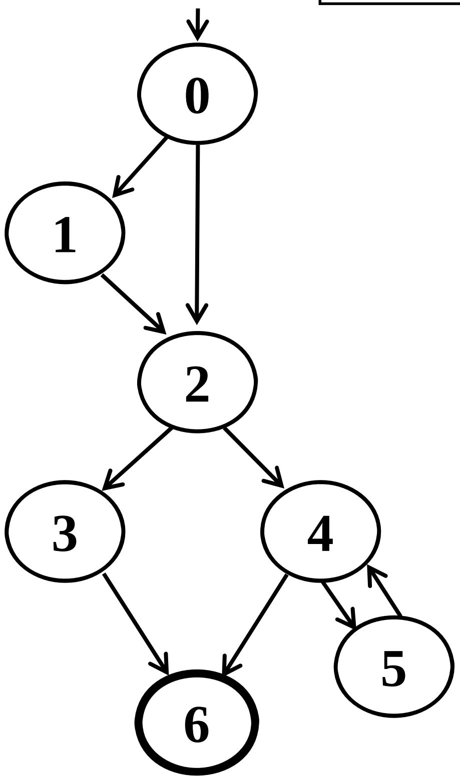
[0, 1, 2, 3, 6]
[0, 1, 2, 4, 5]
[0, 1, 2, 4, 6]
[0, 2, 3, 6]
[0, 2, 4, 5]
[0, 2, 4, 6]
[5, 4, 6]
[4, 5, 4]
[5, 4, 5]

Execute loop 0 times

Execute loop once

Execute loop more than once

Simple & Prime Path Example



Simple paths

‘!’ means path
es

s path
es

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

Note that paths w/o ! or * cannot be prime paths

Round Trips

- Round-Trip Path : *A prime path that starts and ends at the same node*

Simple Round Trip Coverage (SRTC) : TR contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path.

Complete Round Trip Coverage (CRTC) : TR contains all round-trip paths for each reachable node in G.

- **These criteria omit nodes and edges that are not in round trips**
- **That is, they do not subsume edge-pair, edge, or node coverage**

Touring, Sidetrips and Detours

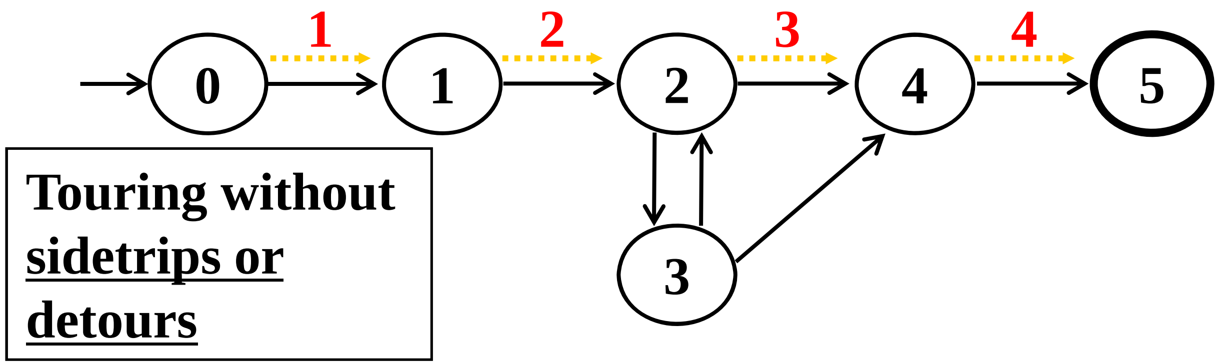
- Prime paths do not have internal loops … test paths <u>might</u>

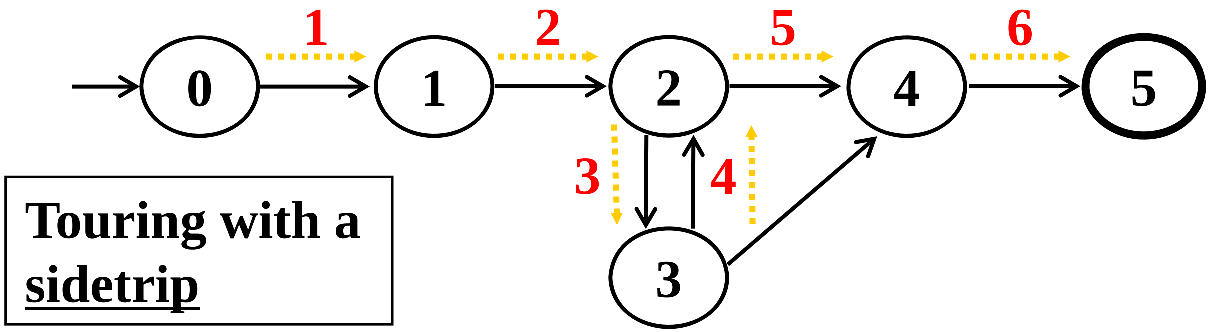
- <u>**Tour**</u> **:** ***A test path p tours subpath q if q is a subpath of p***

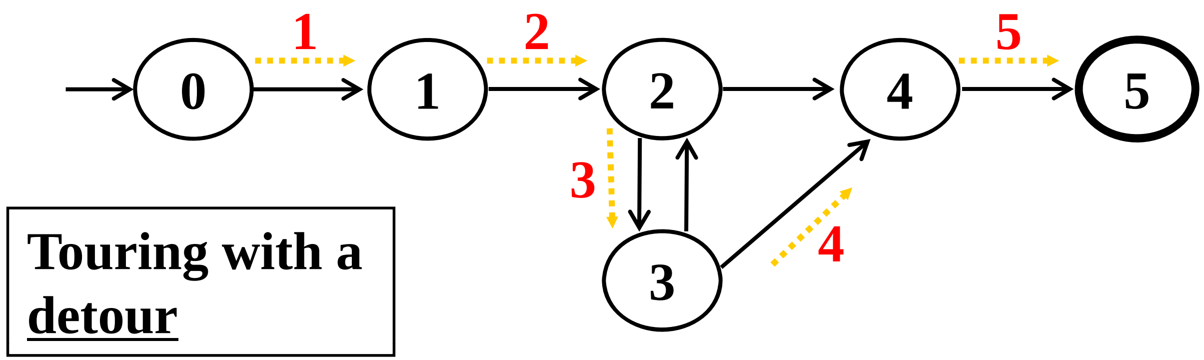
- <u>**Tour With Sidetrips**</u> **:** ***A test path p tours subpath q with <u>sidetrips</u> iff every <u>edge</u> in q is also in p in <u>the same order</u>***
  - **The tour can include a sidetrip, as long as it comes back to the same node**

- <u>**Tour With Detours**</u> **:** ***A test path p tours subpath q with <u>detours</u> iff 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







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

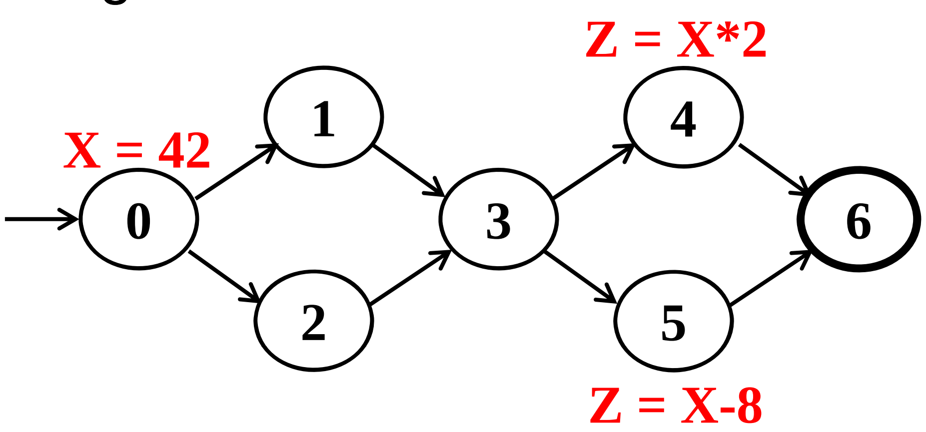
KAIST

Data Flow Coverage

Data Flow Criteria

<u>Goal</u>: Try to ensure that values are computed and used correctly

- <u>Definition</u> : A location where a value for a variable is stored into memory
- <u>Use</u> : A location where a variable's value is accessed
- <u>def (n) or def (e)</u> : The set of variables that are defined by node n or edge e
- <u>use (n) or use (e)</u> : The set of variables that are used by node n or edge e



<u>Defs</u>: def (0) = {X}

def (4) = {Z}

def (5) = {Z}

<u>Uses</u>: use (4) = {X}

use (5) = {X}

DU Pairs and DU Paths

- 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 n odes 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

KAIST

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

KAIST

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**

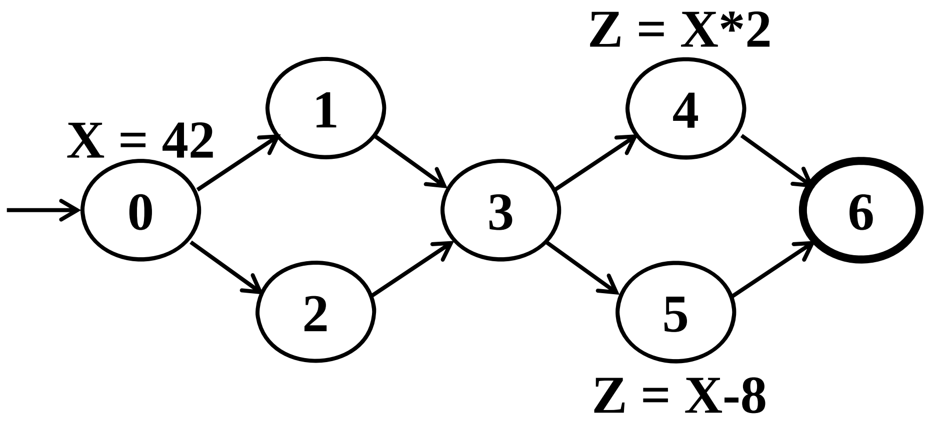
<u>All-uses coverage (AUC)</u> : For each set of du-paths to uses $S = du(n_i, n_j, v)$, TR contains at least one path d in S.

- **Finally, we cover all the paths between defs and uses**

<u>All-du-paths coverage (ADUPC)</u> : For each set $S = du(n_i, n_j, v)$, TR contains every path d in S.

KAIST

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

Assumptions for Data Flow Coverage

1. Every use is preceded by a def
2. Every def reaches at least one use
3. For every node with multiple outgoing edges, at least one variable is used on each out edge, and the same variables are used on each out edge.