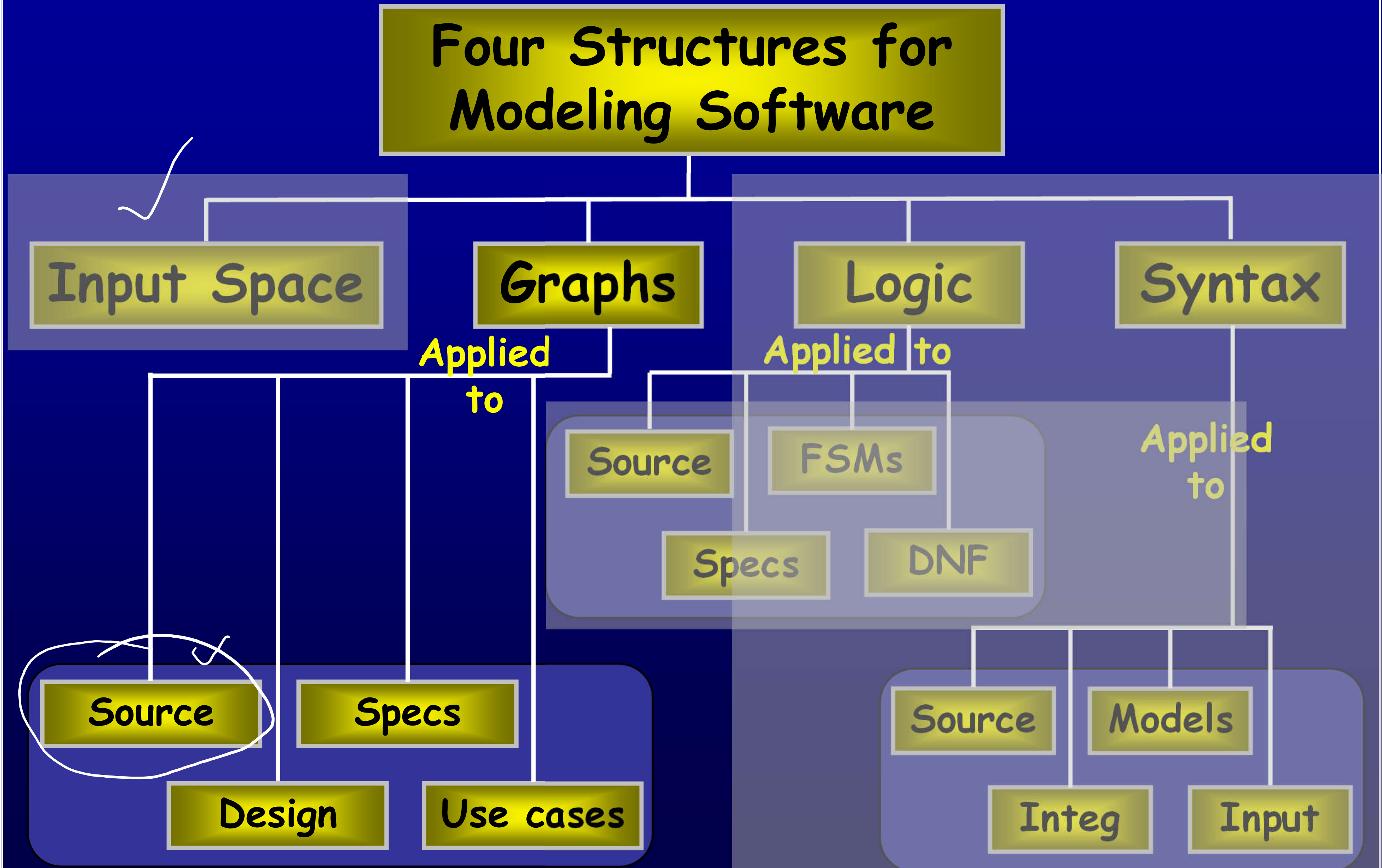
CS 5154

Graph Coverage Criteria

Owolabi Legunsen

The following are modified versions of the publicly-available slides for Chapter 7 in the Ammann and Offutt Book, "Introduction to Software Testing"
(http://www.cs.gmu.edu/~offutt/softwaretest)

Graph Coverage



Covering Graphs

- 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 somehow

Why Graph Coverage?

- Some of the most widely-used coverage criteria
- The "R" in the RIPR model
  - Graph coverage criteria help create tests that reach different parts of software

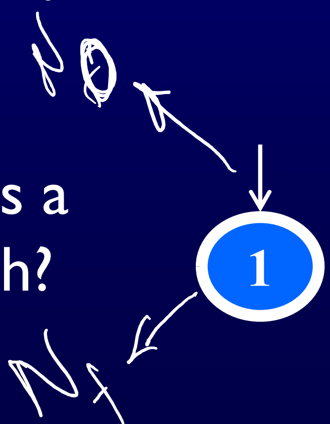
The next two classes

- Today:
  - Review of graph concepts
  - Coverage criteria defined over generic graphs
- Next class (depending on progress today):
  - Apply concepts learned in today's class to source code
- Not in CS 5154
  - Applying graph coverage criteria to design, specs, and use cases

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

Is this a graph?



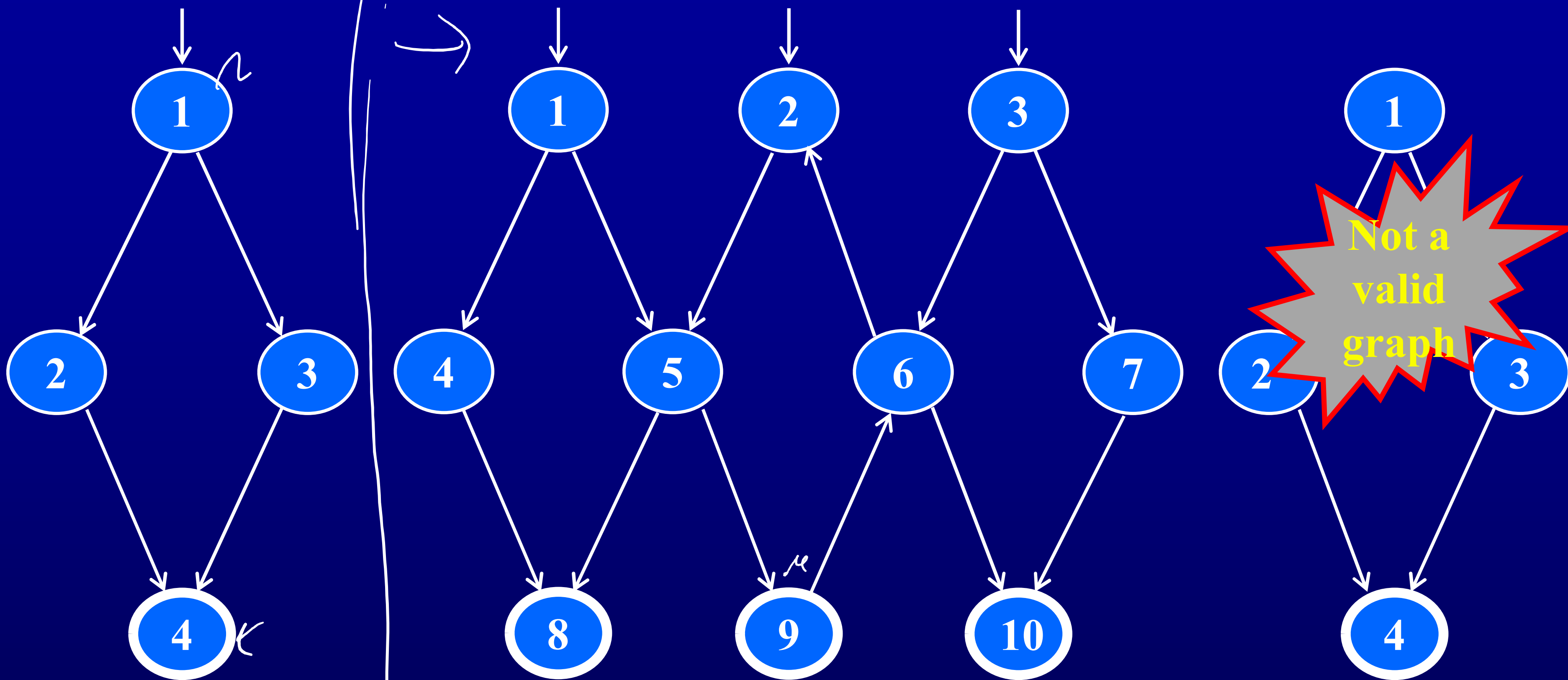
$N_0 = \{ 1 \}$

$N_f = \{ 1 \}$

$E = \{ \}$

Yes

Example Graphs



N_0 = { 1 }

N_f = { 4 }

E = { (1, 2), (1,3), (2,4), (3,4) }

N_0 = { 1, 2, 3 }

N_f = { 8, 9, 10 }

E = { (1,4), (1,5), (2,5), (3,6), (3, 7), (4, 8), (5,8), (5,9), (6,2), (6,10), (7,10) (9,6) }

N_0 = { }

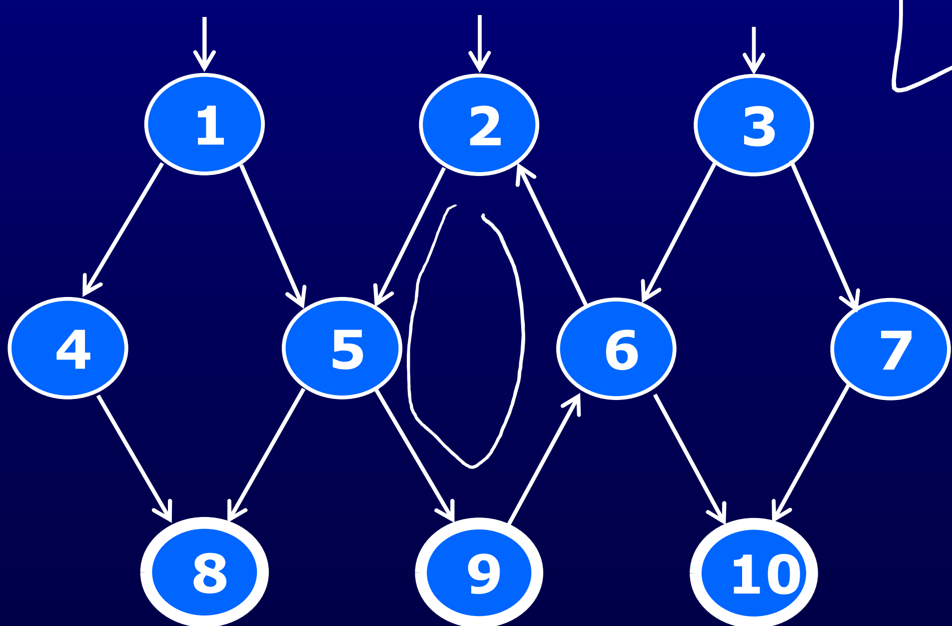
N_f = { 4 }

E = { (1,2), (1,3), (2,4), (3,4) }

Paths in Graphs

- Path : A sequence, *p*, of nodes $[n_1, n_2, \ldots, n_M]$ s.t. there is an edge between each pair of nodes in *p*
- Length of a path : The number of edges in *p*
  - A single node is a path of length 0
- Subpath : A subsequence of nodes in *p* is a subpath of *p*

[2, 5, 9, 6, 2]



A Few Paths

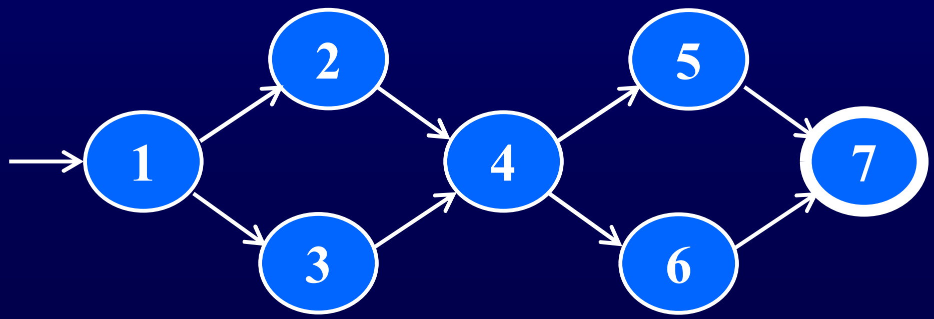
[1, 4, 8]

[2, 5, 9, 6, 2]

[3, 7, 10]

Test Paths and SESE graphs

- 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
  - N_0 and N_f have exactly one node



Double-diamond graph

Four test paths

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

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*

Test path [1, 2, 4, 5, 7]

Visits nodes ? 1, 2, 4, 5, 7

Visits edges ? (1,2), (2,4), (4, 5), (5, 7)

Tours subpaths ? [1,2,4], [2,4,5], [4,5,7], [1,2,4,5], [2,4,5,7], [1,2,4,5,7]

(Also, each edge is technically a subpath)

Tests and Test Paths

- path (*t*) : The test path executed by test *t*
- path (*T*) : The set of test paths executed by set of tests *T*
- Each test executes one and only one test path
  - Complete execution from a start node to a final node

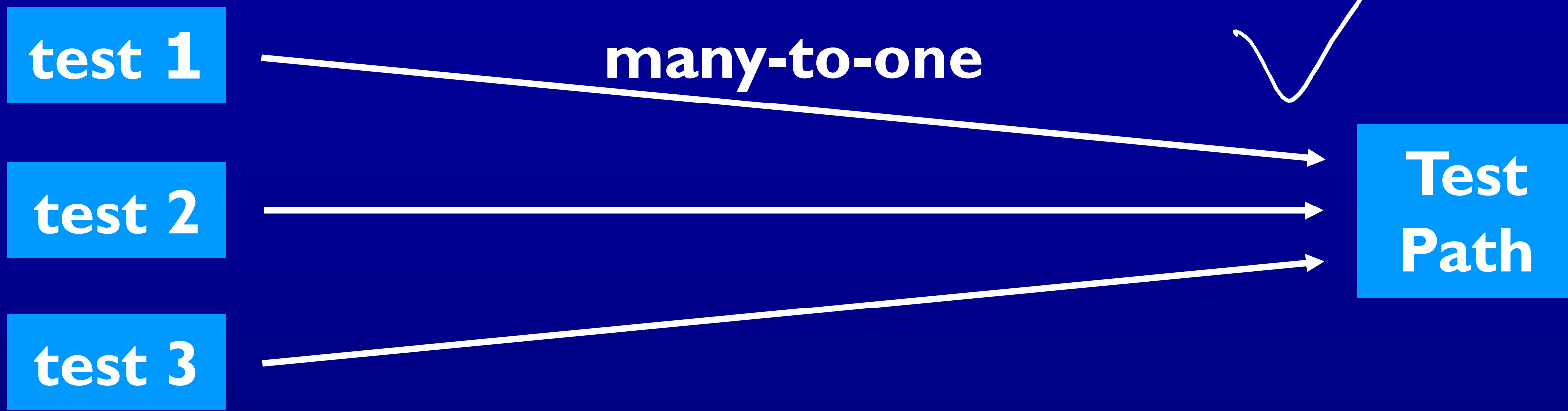
Is the last bullet true?

- recursion

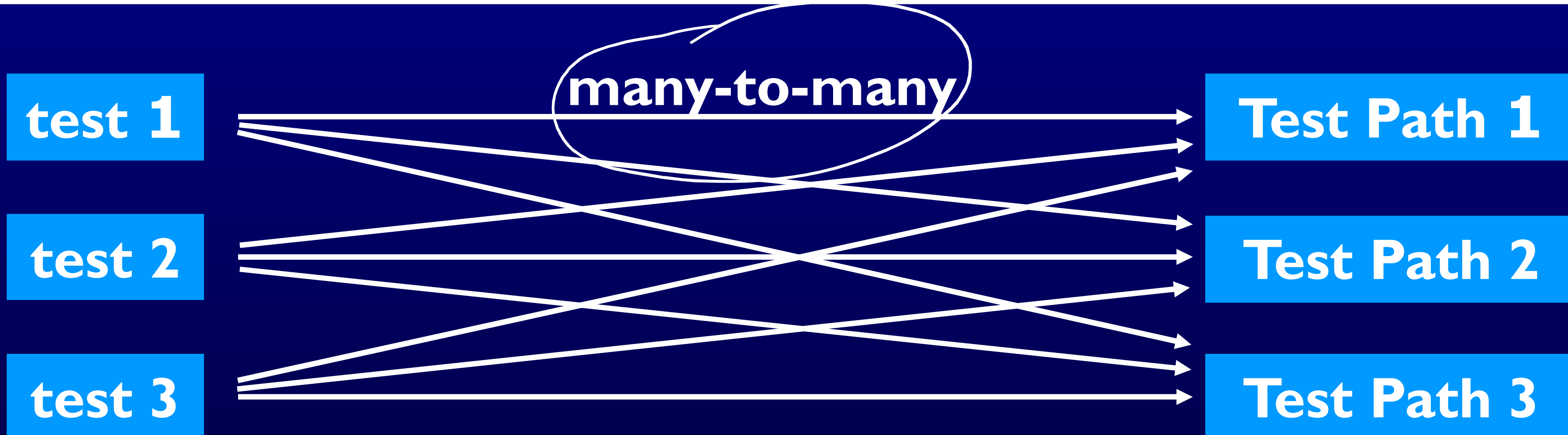
Tests and Test Paths (2)

- 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
  - This distinction (semantic vs syntactic) is important when applied to source code

Tests and Test Paths (3)



Deterministic software: a test always execute same test path



Non-deterministic software: a test can execute >1 test paths

Testing and Covering Graphs

- We use graphs in testing as follows :
  - Develop a model of the software as a graph
  - Require tests to visit/tour sets of nodes, edges or sub-paths

Testing and Covering Graphs (2)

- 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 tr in TR, there is a test path in path(T) that meets the test requirement tr*

Two kinds of graph coverage criteria

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

Node Coverage

- The first (and simplest) two criteria require that each node and edge in a graph be executed

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*.

- This statement is a bit cumbersome, so we abbreviate it in terms of the set of test requirements

Node Coverage (NC) : TR contains each reachable node in G.

Edge Coverage

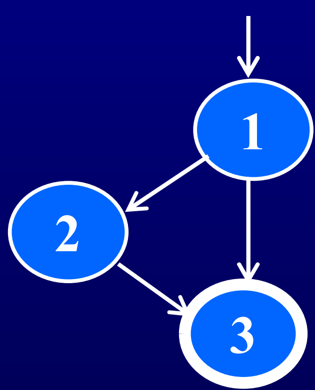
- Edge coverage is slightly stronger than node coverage

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

- The phrase "*length up to 1*" allows for graphs with one node and no edges

Node and Edge Coverage

- 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)

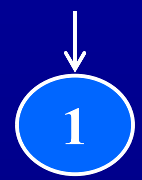


Node Coverage : ? **TR = { 1, 2, 3 }**
Test Path = [1, 2, 3]

Edge Coverage : ? **TR = { (1, 2), (1, 3), (2, 3) }**
Test Paths = [1, 2, 3] ✓
[1, 3]

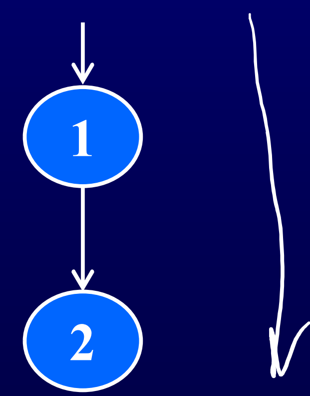
Paths of Length 1 and 0

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



- It may seem trivial, but formally, Edge Coverage needs to require Node Coverage on this graph
- Else, 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 …

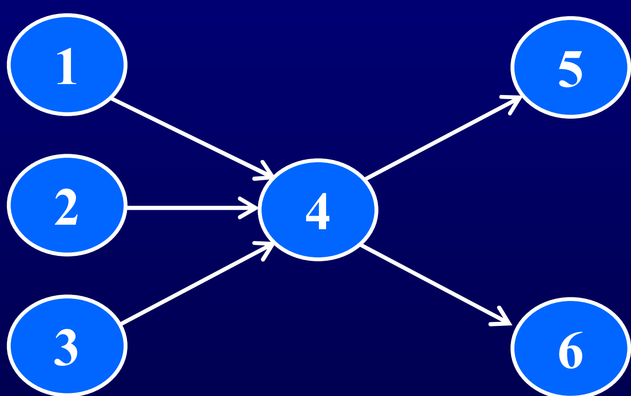


Covering Multiple Edges

- 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 phrase "length up to 2" is used to include graphs that have less than 2 edges



Edge-Pair Coverage : ?

TR = { [1,4,5], [1,4,6], [2,4,5], [2,4,6], [3,4,5], [3,4,6] }

- A logical extension is to require covering all paths …

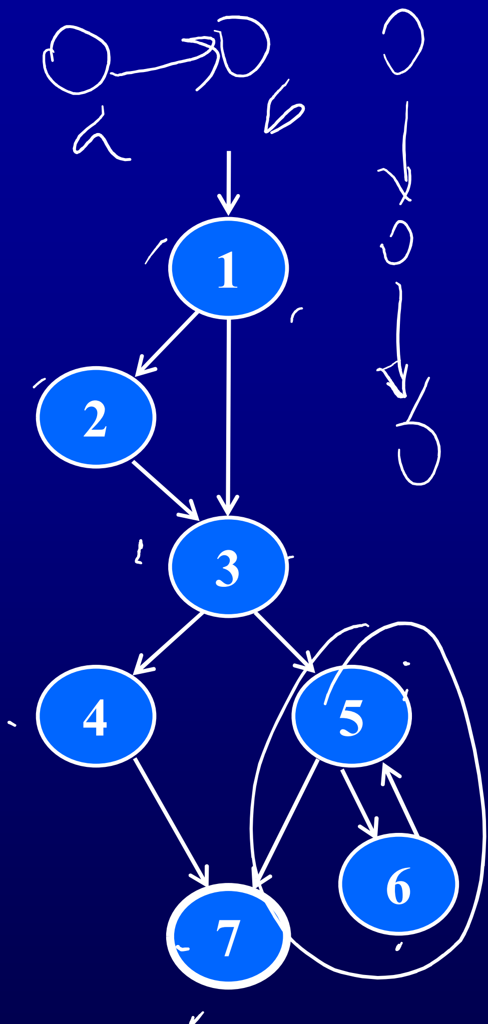
Covering Multiple Edges

<u>Complete Path Coverage (CPC)</u> : TR contains all paths in G.

Unfortunately, this is impossible if the graph has a loop, so a weak compromise makes the tester decide which paths:

<u>Specified Path Coverage (SPC)</u> : TR contains a set S of test paths, where S is supplied as a parameter.

Structural Coverage Example



Node Coverage

TR = { 1, 2, 3, 4, 5, 6, 7 }

Test Paths: [1, 2, 3, 4, 7] [1, 2, 3, 5, 6, 5, 7]

Edge Coverage

TR = { (1,2), (1, 3), (2, 3), (3, 4), (3, 5), (4, 7), (5, 6), (5, 7), (6, 5) }

Test Paths: [1, 2, 3, 4, 7] [1, 3, 5, 6, 5, 7]

Edge-Pair Coverage

TR = { [1,2,3], [1,3,4], [1,3,5], [2,3,4], [2,3,5], [3,4,7], [3,5,6], [3,5,7], [5,6,5], [6,5,6], [6,5,7] }

Test Paths: [1, 2, 3, 4, 7] [1, 2, 3, 5, 7] [1, 3, 4, 7] [1, 3, 5, 6, 5, 6, 5, 7] [1, 3, 5, 7]

Complete Path Coverage

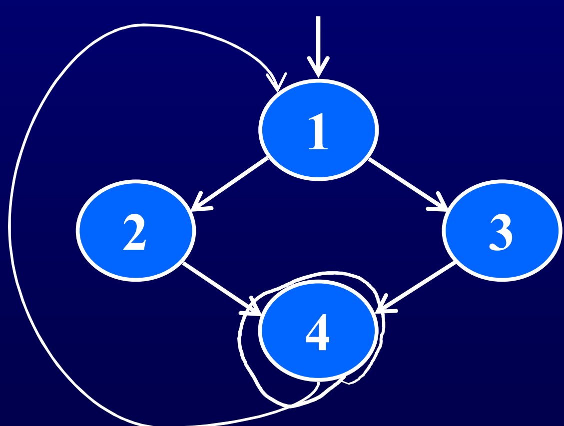
Test Paths: [1, 2, 3, 4, 7] [1, 2, 3, 5, 7] [1, 2, 3, 5, 6, 5, 7] [1, 2, 3, 5, 6, 5, 6, 5, 7] [1, 2, 3, 5, 6, 5, 6, 5, 6, 5, 7] …

Handling Loops in Graphs

- If a graph contains a loop, it has an infinite number of paths
- Thus, Complete Path Coverage 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 ([5, 6, 5] 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 (touring, sidetrips, and detours)

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 [2,5,6,5,7] ✗
  - A loop is a simple path [5,6,5] ✓
- Prime Path : *A simple path that does not appear as a proper subpath of any other simple path*



<u>Simple Paths</u> : [1,2,4,1], [1,3,4,1], [2,4,1,2], [2,4,1,3], [3,4,1,2], [3,4,1,3], [4,1,2,4], [4,1,3,4], [1,2,4], [1,3,4], [2,4,1], [3,4,1], [4,1,2], [4,1,3], [1,2], [1,3], [2,4], [3,4], [4,1], [1], [2], [3], [4]

<u>Prime Paths</u> : [2,4,1,2], [2,4,1,3], [1,3,4,1], [1,2,4,1], [3,4,1,2], [4,1,3,4], [4,1,2,4], [3,4,1,3]

Prime Path Coverage

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

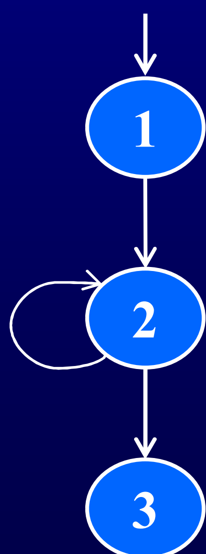
<u>Prime Path Coverage (PPC)</u> : TR contains each prime path in G.

- Will tour all paths of length 0, 1, …
- That is, it subsumes node and edge coverage
- PPC almost, but not quite, subsumes EPC …

Why does PPC not subsume EPC?

PPC Does Not Subsume EPC

- If a node *n* has an edge to itself (*self edge*), EPC requires [*n, n, m*] and [*m, n, n*]
- Neither [*n, n, m*] nor [*m, n, n*] are simple paths (not prime)



EPC Requirements : ?

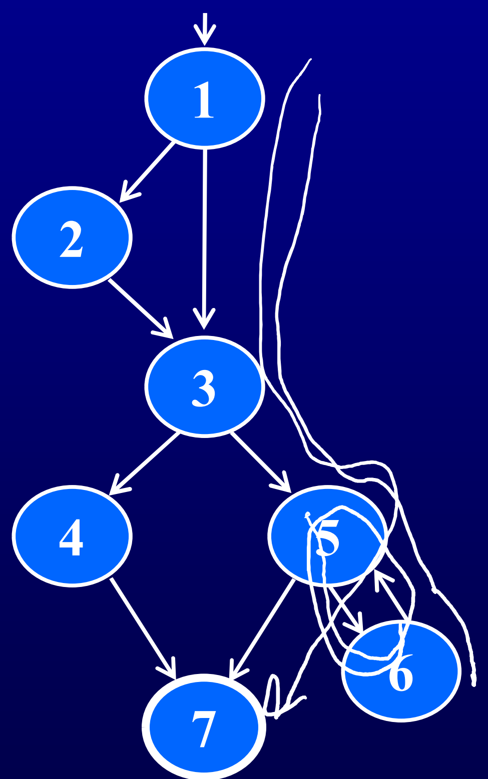
TR = { [1,2,3], [1,2,2], [2,2,3], [2,2,2] **}**

PPC Requirements : ?

TR = { [1,2,3], [2,2] **}**

Prime Path Example

- The previous example has 38 simple paths
- Only nine *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 loop 0 times

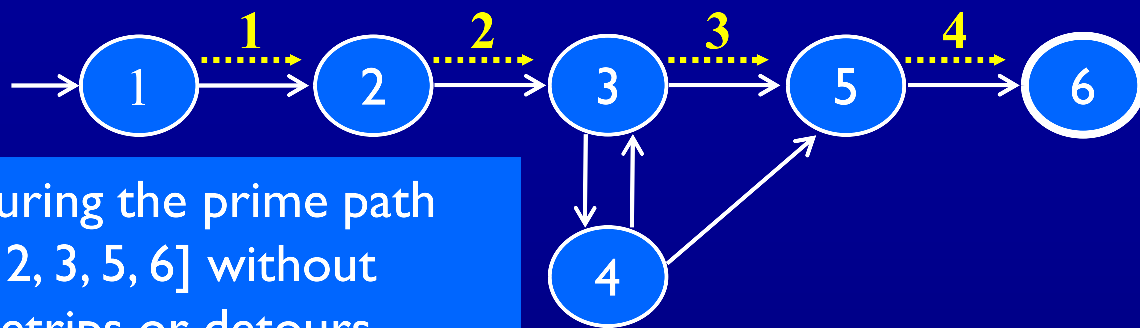
Execute loop once

Execute loop more than once

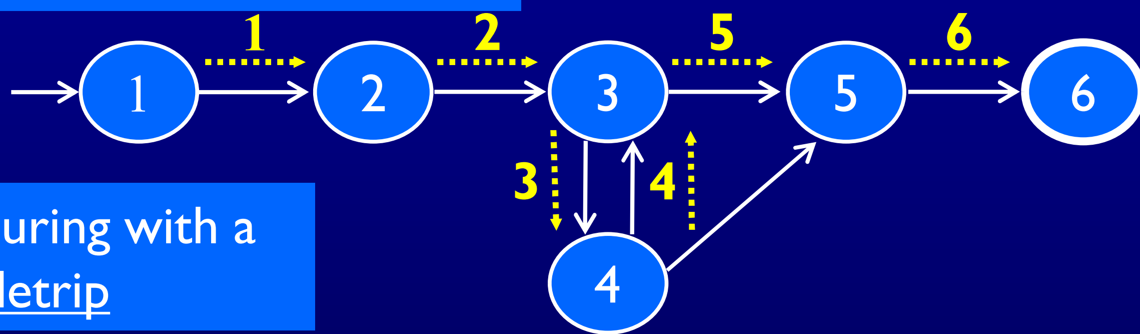
Touring, Sidetrips, and Detours

- Prime paths have no internal loops … test paths might
- Tour : *A test path p tours subpath q if q is a subpath of p*
- Tour With Sidetrips : *A test path p tours subpath q with sidetrips iff every edge in q is also in p in the same order*
  - Tour can have a sidetrip if it comes back to the same node
- Tour With Detours : *A test path p tours subpath q with detours iff every node in q is also in p in the same order*
  - Tour can have a detour from node n_i, if it returns to the prime path at a successor of n_i

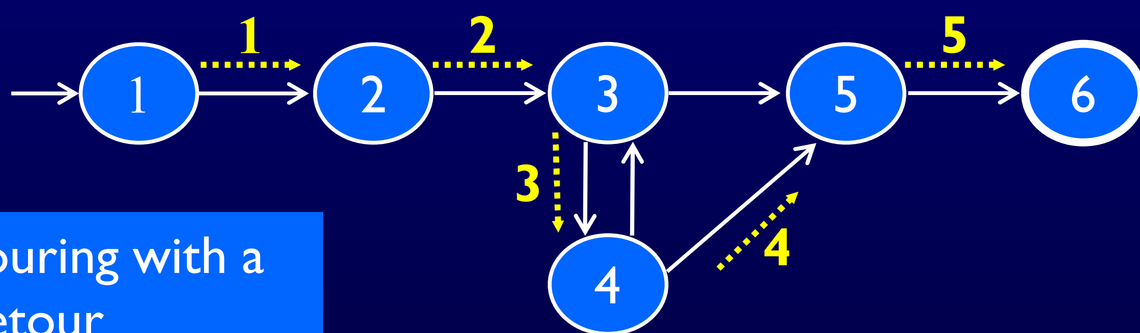
Sidetrips and Detours Example



Touring the prime path [1, 2, 3, 5, 6] without <u>sidetrips or detours</u>



Touring with a <u>sidetrip</u>



Touring with a <u>detour</u>

Dealing with Infeasible TRs

- An infeasible test requirement <u>cannot be satisfied</u>
  - Unreachable statement (dead code)
  - Subpath that can only be toured if a contradiction holds, e.g., if ($X > 0$ and $X < 0$)
- Most test criteria have some infeasible test requirements
- It is usually undecidable whether all test requirements are feasible

Infeasible TRs and Sidetrips

- When sidetrips are not allowed, many structural criteria have more infeasible test requirements
- However, always allowing sidetrips weakens the test criteria

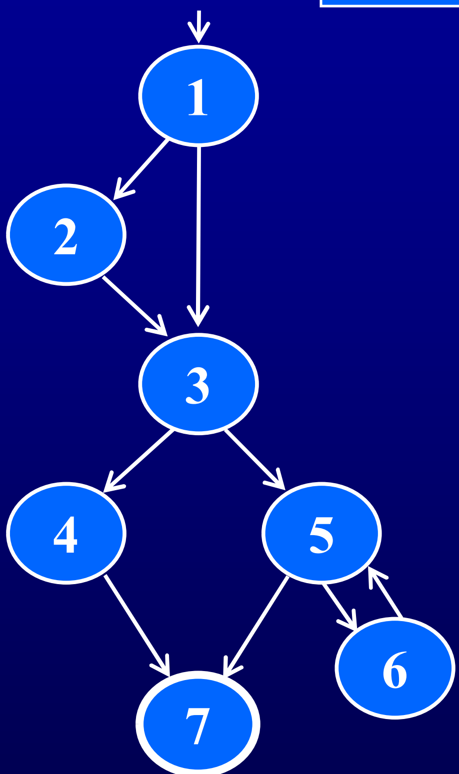
Practical recommendation—Best Effort Touring

- **First, satisfy as many test requirements as possible without sidetrips**
- **Then, allow sidetrips to try to satisfy remaining test requirements**

Simple path & prime path example

"!" Means "cannot be extended to a simple path"

Simple paths



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

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

Len 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] *

Len 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] !

'*' means path cycles

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

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.

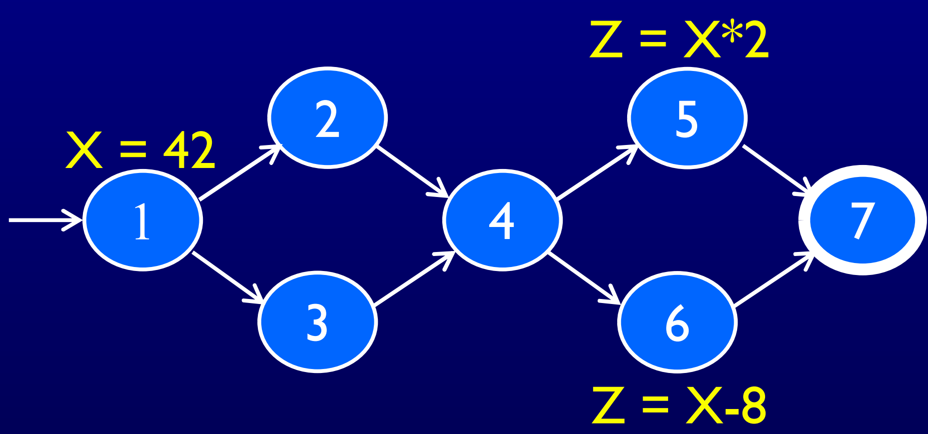
Complete Round Trip Coverage (CRTC) : TR contains all round-trip paths for each reachable node in G.

- The criteria omit nodes & edges that are not in round trips
- They do not subsume edge-pair, edge, or node coverage

Data Flow Criteria

<u>Goal</u> : Ensure that values are computed and used correctly

- Definition (def) : A location where a value for a variable is stored into memory
- Use : A location where a variable's value is accessed



<u>Defs</u>: def (1) = { **X** }

def (5) = { **Z** }

def (6) = { **Z** }

Fill in these sets

<u>Uses</u>: use (5) = { **X** }

use (6) = { **X** }

The values given in defs should reach at least one, some, or all possible uses

DU Pairs and DU Paths

- def (n) or def (e) : The set of variables that are defined by node n or edge e
- use (n) or use (e) : The set of variables that are used by node n or edge e

- 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 nodes 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

Touring DU-Paths

- A test path *p* *du-tours* subpath *d* with respect to *v* if *p* tours *d* and *d* 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

Data Flow Test Criteria

- First, we make sure every def reaches a use

All-defs coverage (ADC) : For each set of du-paths $S = du(n, v)$, TR contains at least one path d in S.

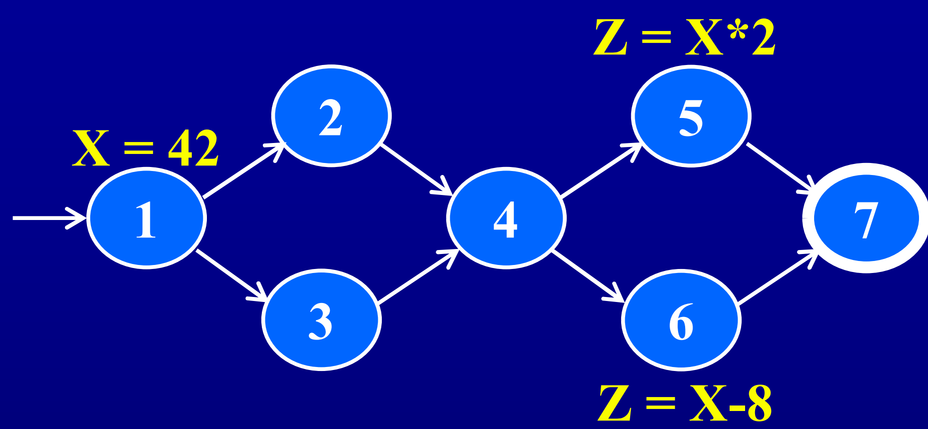
- Then we make sure that every def reaches all possible uses

All-uses coverage (AUC) : 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

All-du-paths coverage (ADUPC) : For each set $S = du(ni, nj, v)$, TR contains every path d in S.

Data Flow Testing Example



All-defs for X

[1, 2, 4, 5]

All-uses for X

[1, 2, 4, 5]

[1, 2, 4, 6]

All-du-paths for X

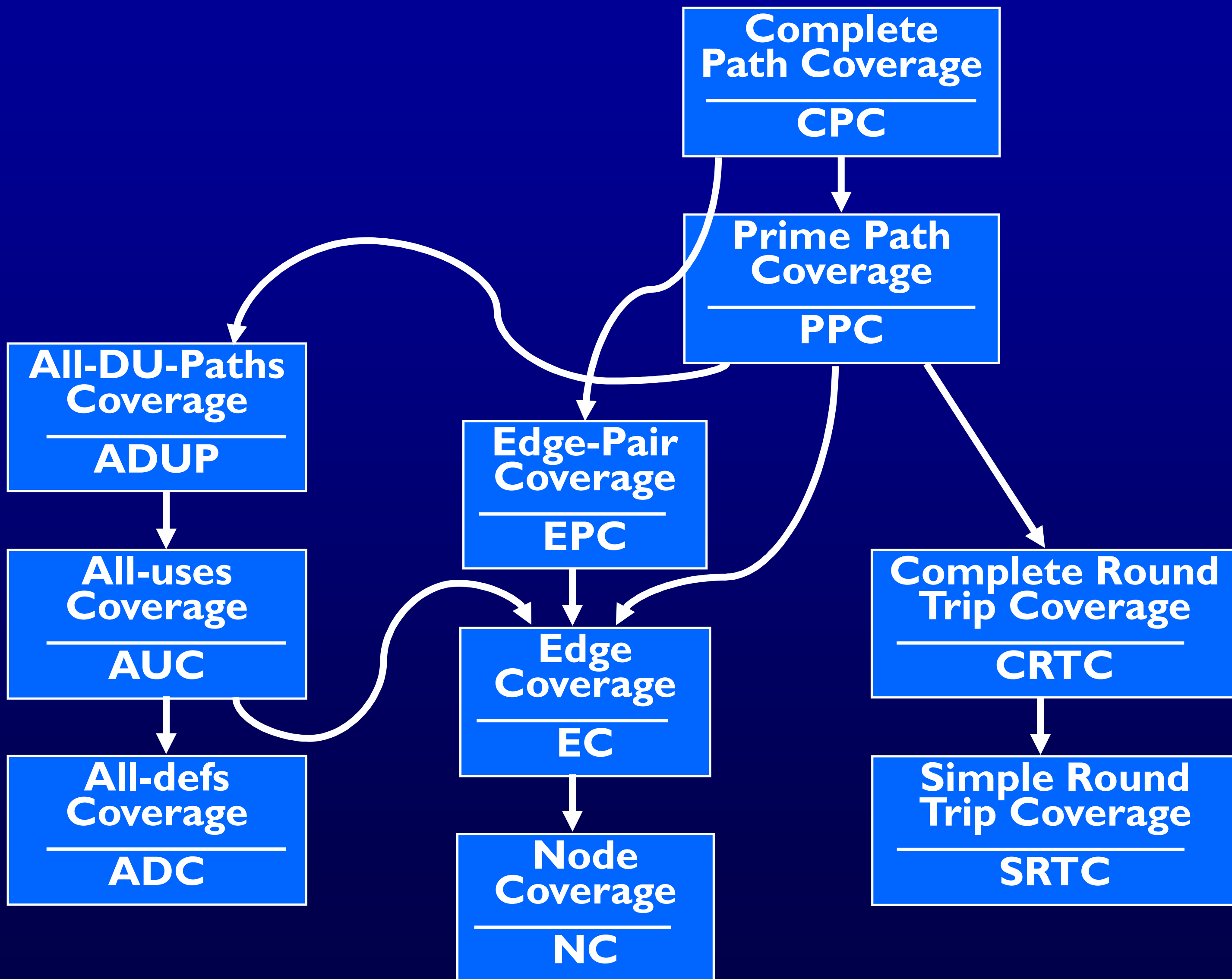
[1, 2, 4, 5]

[1, 3, 4, 5]

[1, 2, 4, 6]

[1, 3, 4, 6]

Graph-based criteria subsumption



Summary

- Graphs are a very powerful abstraction for designing tests
- The various criteria allow lots of cost / benefit tradeoffs
- These two sections are entirely at the "design abstraction level" from chapter 2
- Graphs appear in many situations in software
  – Next: we will apply these criteria to source code
  – Design, specs, and use cases are not covered in CS 5154