COMP5111 – Fundamentals of Software Testing and Analysis

Overview of Test Coverage

Shing-Chi Cheung

Computer Science & Engineering

HKUST

https://www.youtube.com/watch?v=4bublRBCLVQ

Also available at: https://hkustconnect-my.sharepoint.com/:v:/g/personal/sccheung_connect_ust_hk/EeD98fs7o_VKpuqbn8AdJW8BAice4fvdsRLQz1ct7HF1fw?e=N4TVJB

Why coverage?

Project Coverage summary

Name	Classes		Conditionals		Files		Lines		Packages	
Cobertura Coverage Report	45%	23/51	74%	469/630	45%	23/51	28%	1450/5222	88%	7/8

Coverage Breakdown by Package

Name	Classes		Conditionals		Files		Lines	
Stop-tabac	0%	0/1	N/A		0%	0/1	0%	0/5
Stop-tabac.Classes	100%	1/1	47%	8/17	100%	1/1	27%	58/213
Stop-tabac.Classes.Controller	19%	3/16	73%	22/30	19%	3/16	5%	120/2665
Stop-tabac.Classes.Manager	100%	3/3	63%	20/32	100%	3/3	65%	103/158
Stop-tabac.Classes.Model	75%	6/8	81%	249/306	75%	6/8	77%	669/869
Stop-tabac.Classes.Service	56%	5/9	69%	163/235	56%	5/9	47%	388/830
Stop-tabac.Classes.Utils	60%	3/5	N/A		60%	3/5	59%	74/126
Stop-tabac.Classes.View	25%	2/8	70%	7/10	25%	2/8	11%	38/356

- Coverage provides a quantitative measurement of how well we have validated a piece of code.

Coverage Criteria

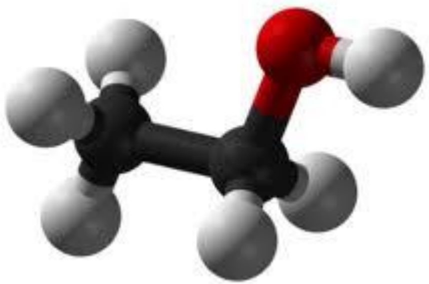
A tester's job is <u>simple</u> : Define software test requirements, then find ways to cover them

- <u>Test Requirements</u> : Specific software elements that a test case must satisfy or cover
- <u>Test Criterion</u> : A characterization of a set of test requirements
  - Four major models of coverage characterization

Testing researchers have defined dozens of criteria, but they are all criteria defined on top of four models …

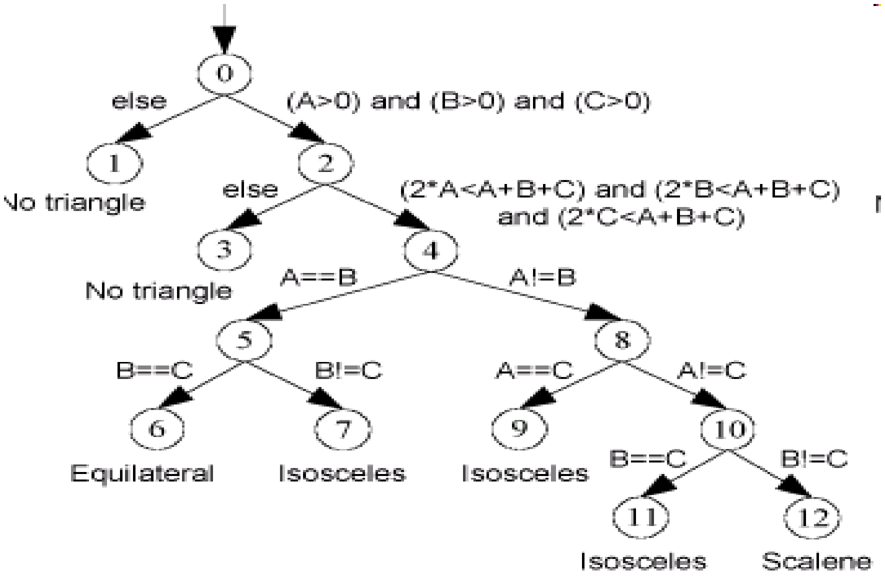
Four Primary Coverage Models

- Graph model
- Logical expression
- Input domain characterization
- Syntactic structure



Test Coverage Criteria (Branch Coverage)

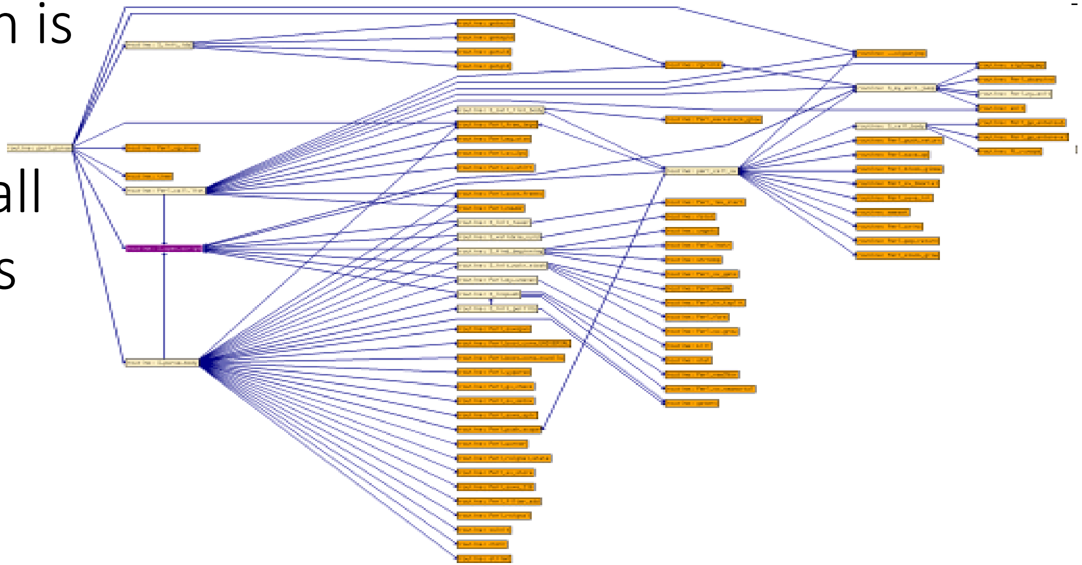
- Each outgoing edge of a decision node is a test requirement.
- There are 12 test requirements.
- Criterion that imposes these 12 test requirements on each test set is 'branch coverage'.



Flow Graph

Test Coverage Criteria (Call Coverage)

- Each method invocation is a test requirement.
- Criterion that imposes all these test requirements on a test set is 'call coverage'.



Method Call Graph

Test Coverage Criteria (Statement Coverage)

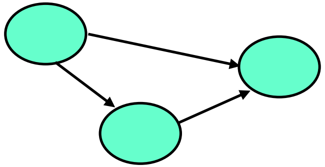
- Each statement invocation is a test requirement.
- Criterion imposes all these requirements on a test set is 'statement coverage'.

```
public static int numZero (int[] x) {
// Effects: if x == null throw NullPointerException
// else return the number of occurrences of 0 in x
   int count = 0;
   for (int i = 1; i < x.length; i++) {
       if (x[i] == 0) {
            count++;
       }
   }
   return count;
}
```

Summary: Criteria Based on Structures

Structures : Four ways to model software

1. Graphs



2. Logical Expressions

(not X or not Y) and A and B

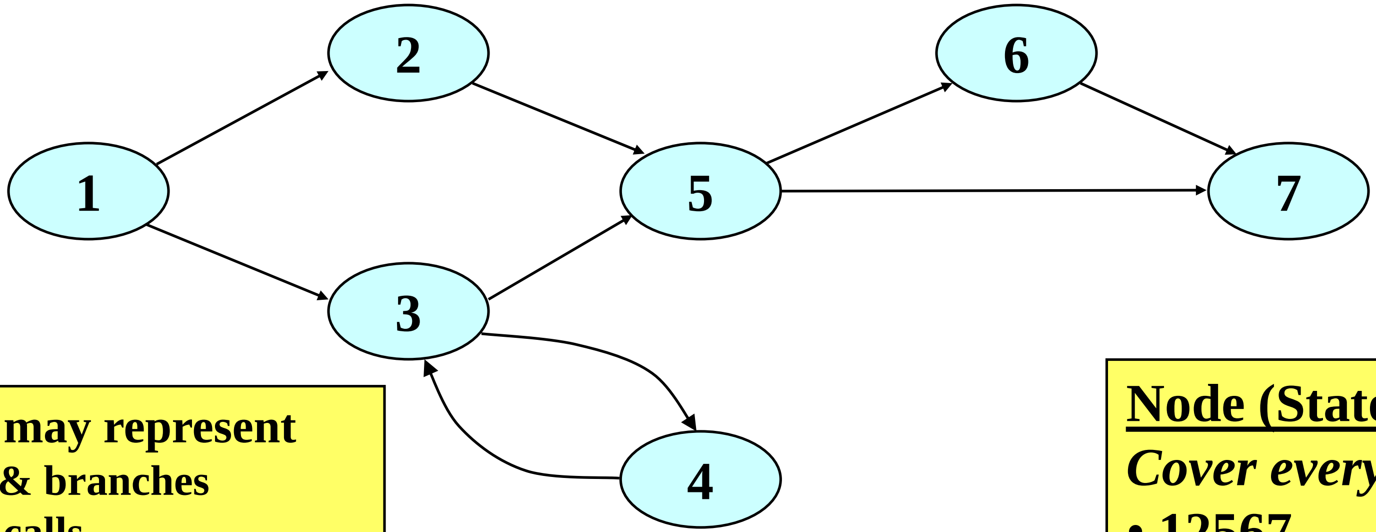
3. Input Domain Characterization

A: {0, 1, >1}
B: {600, 700, 800}
C: {swe, cs, isa, infs}

4. Syntactic Structures

```
if (x > y)
    z = x - y;
else
   z = 2 * x;
```

1. Graph Coverage – Structural



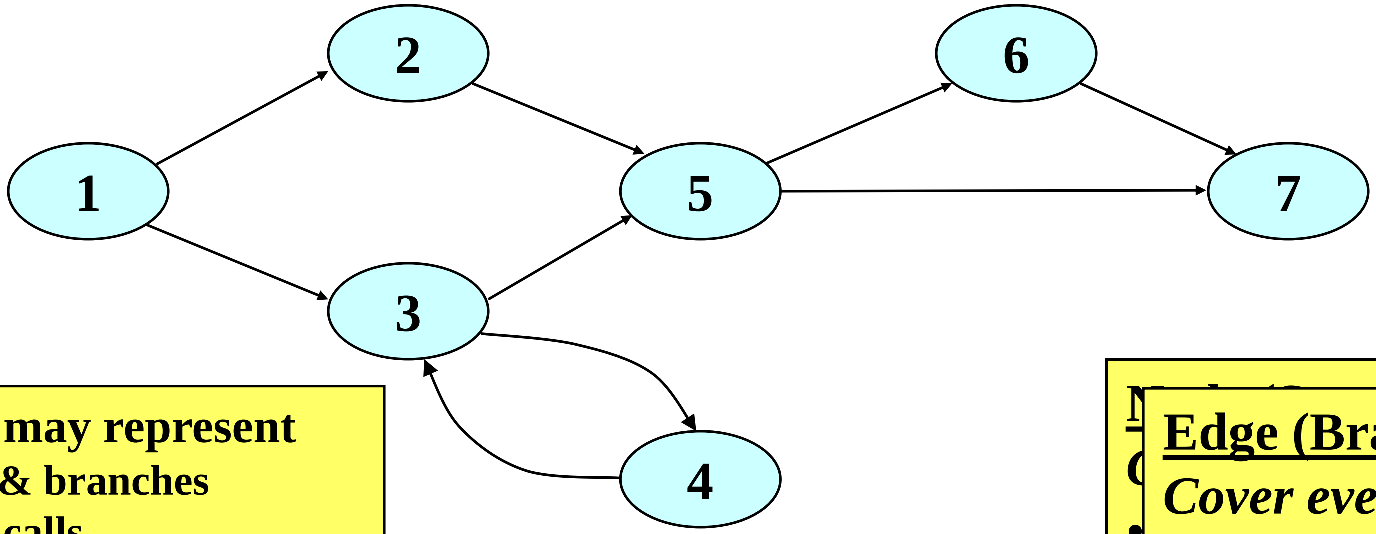
This graph may represent
- **statements & branches**
- **methods & calls**
- **components & signals**
- **states and transitions**

⋮

<u>Node (Statement)</u>

Cover every node
- **12567**
- **1343567**

1. Graph Coverage – Structural



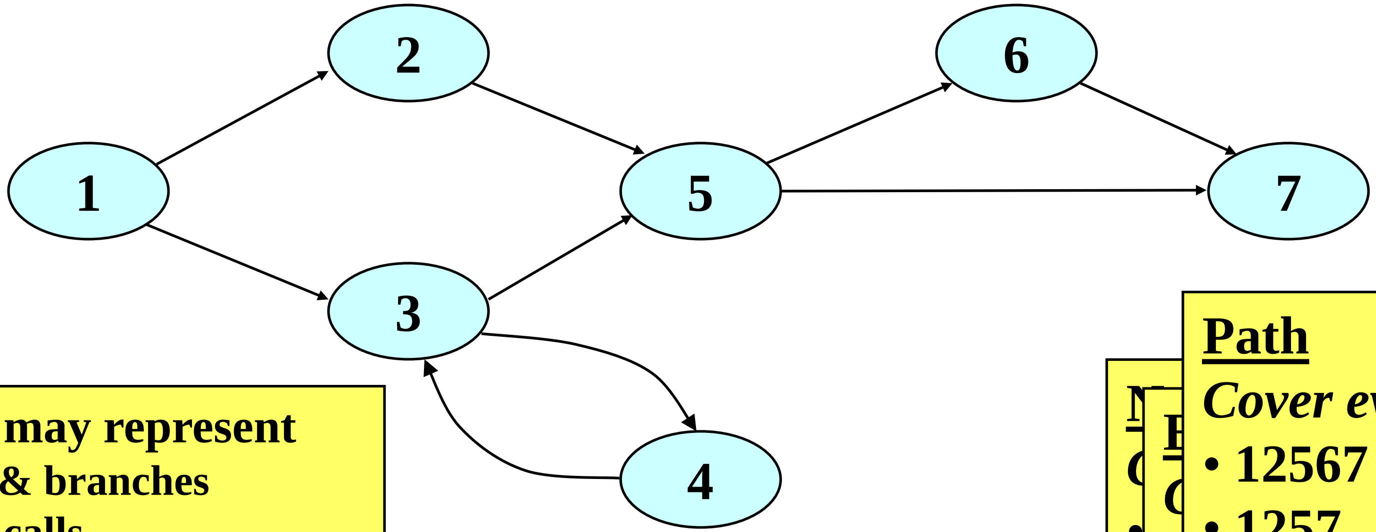
This graph may represent
- **statements & branches**
- **methods & calls**
- **components & signals**
- **states and transitions**

⋮

<u>**Edge (Branch)**</u>

Cover every edge
- **12567**
- **134357**

1. Graph Coverage – Structural



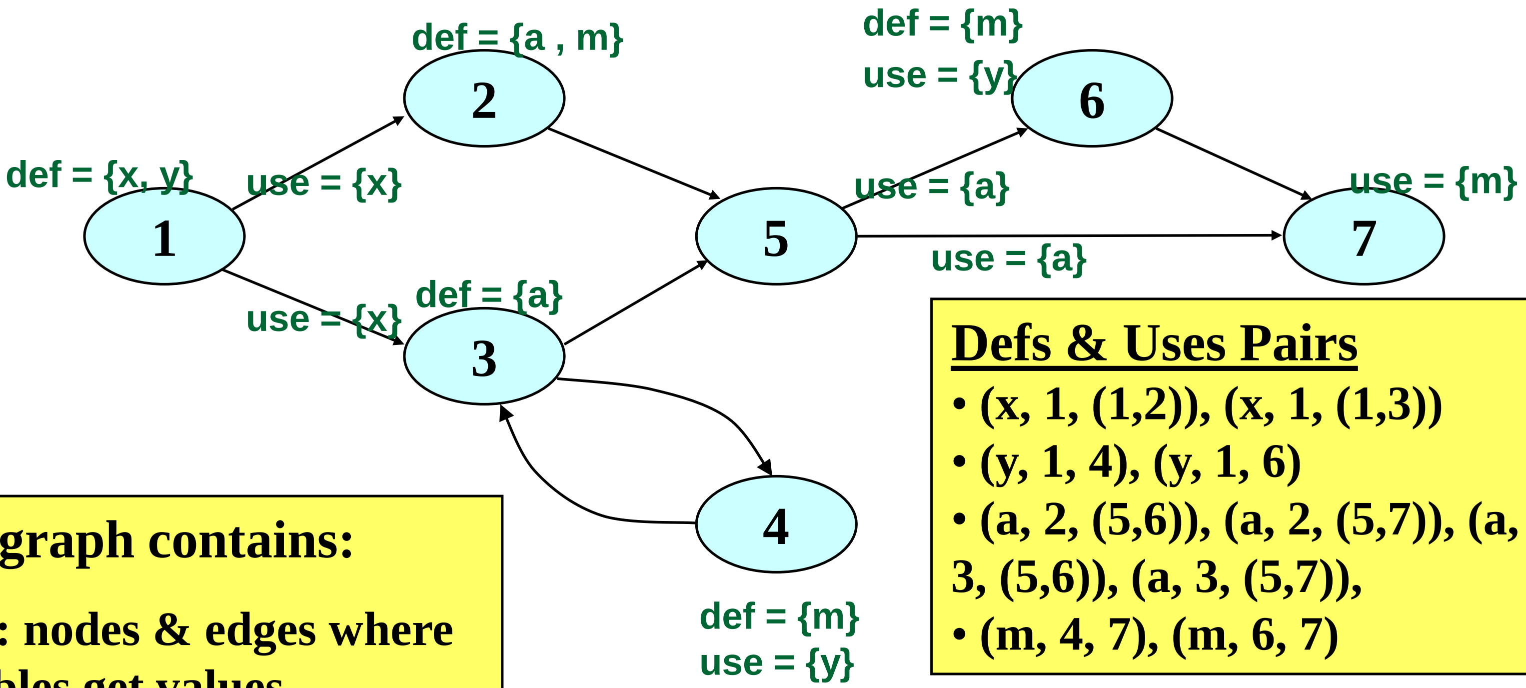
This graph may represent
- **statements & branches**
- **methods & calls**
- **components & signals**
- **states and transitions**

⋮

<u>Path</u>

Cover every path
- **12567**
- **1257**
- **13567**
- **1357**
- **1343567**
- **134357**
- **…**

1. Graph Coverage – Data Flow



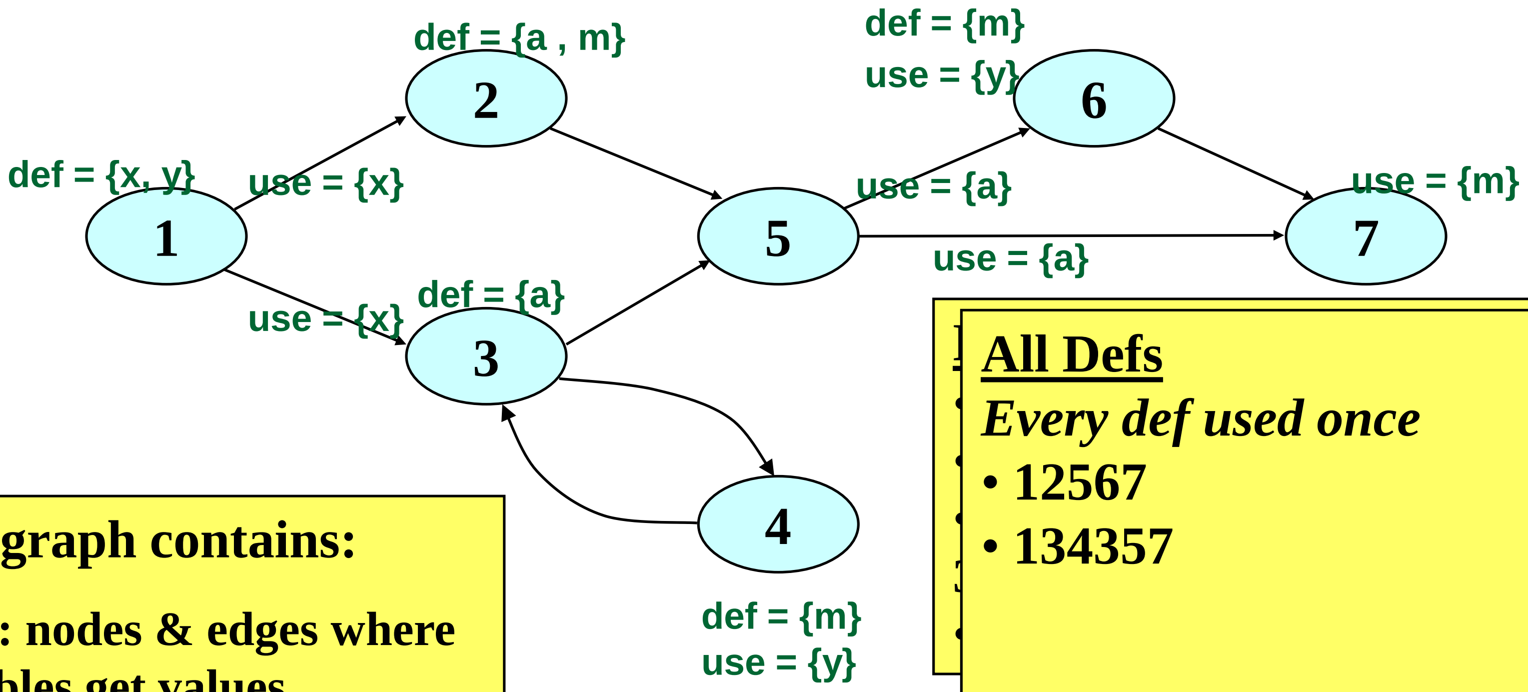
This graph contains:

- **<u>defs</u>: nodes & edges where variables get values**
- **<u>uses</u>: nodes & edges where values are accessed**

<u>Defs & Uses Pairs</u>

- **(x, 1, (1,2)), (x, 1, (1,3))**
- **(y, 1, 4), (y, 1, 6)**
- **(a, 2, (5,6)), (a, 2, (5,7)), (a, 3, (5,6)), (a, 3, (5,7)),**
- **(m, 4, 7), (m, 6, 7)**

1. Graph Coverage – Data Flow



This graph contains:

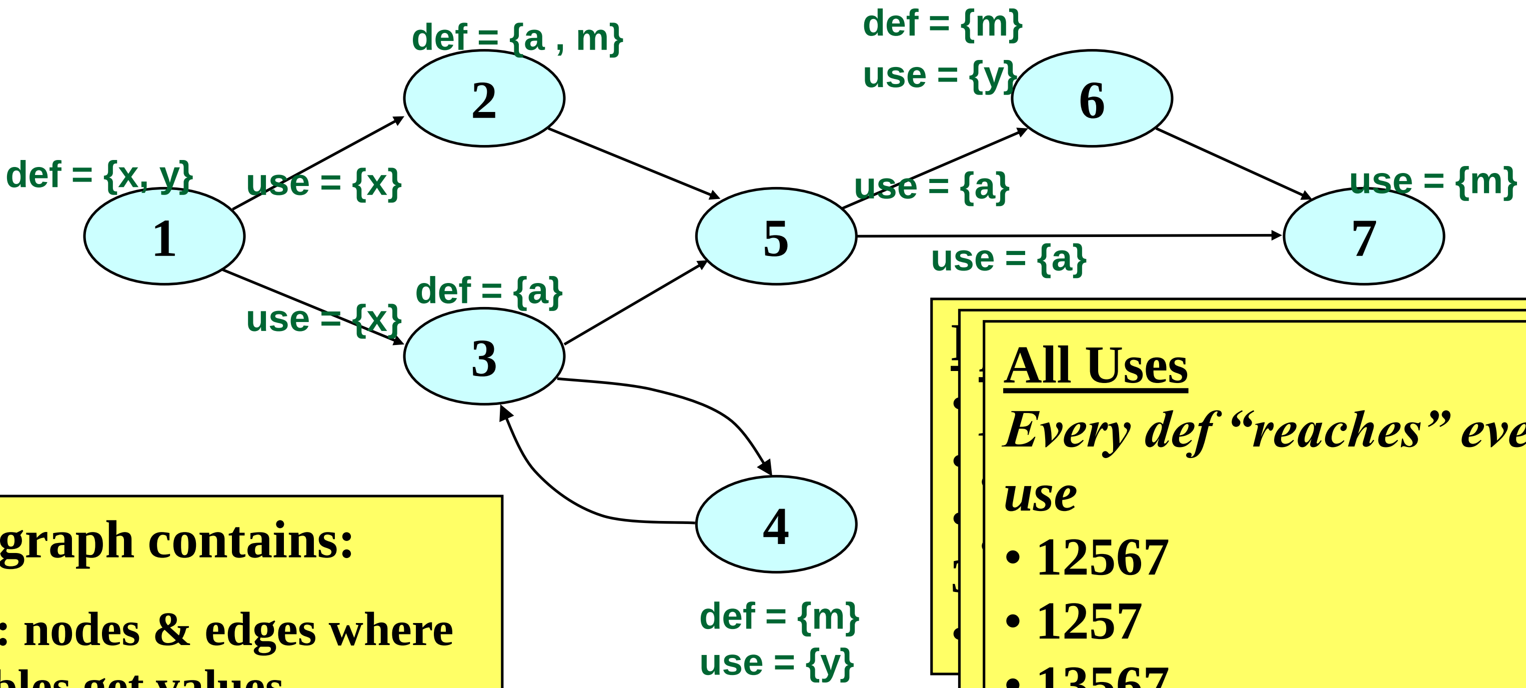
- **<u>defs</u>: nodes & edges where variables get values**
- **<u>uses</u>: nodes & edges where values are accessed**

<u>All Defs</u>

Every def used once

- **12567**
- **134357**

1. Graph Coverage – Data Flow



This graph contains:

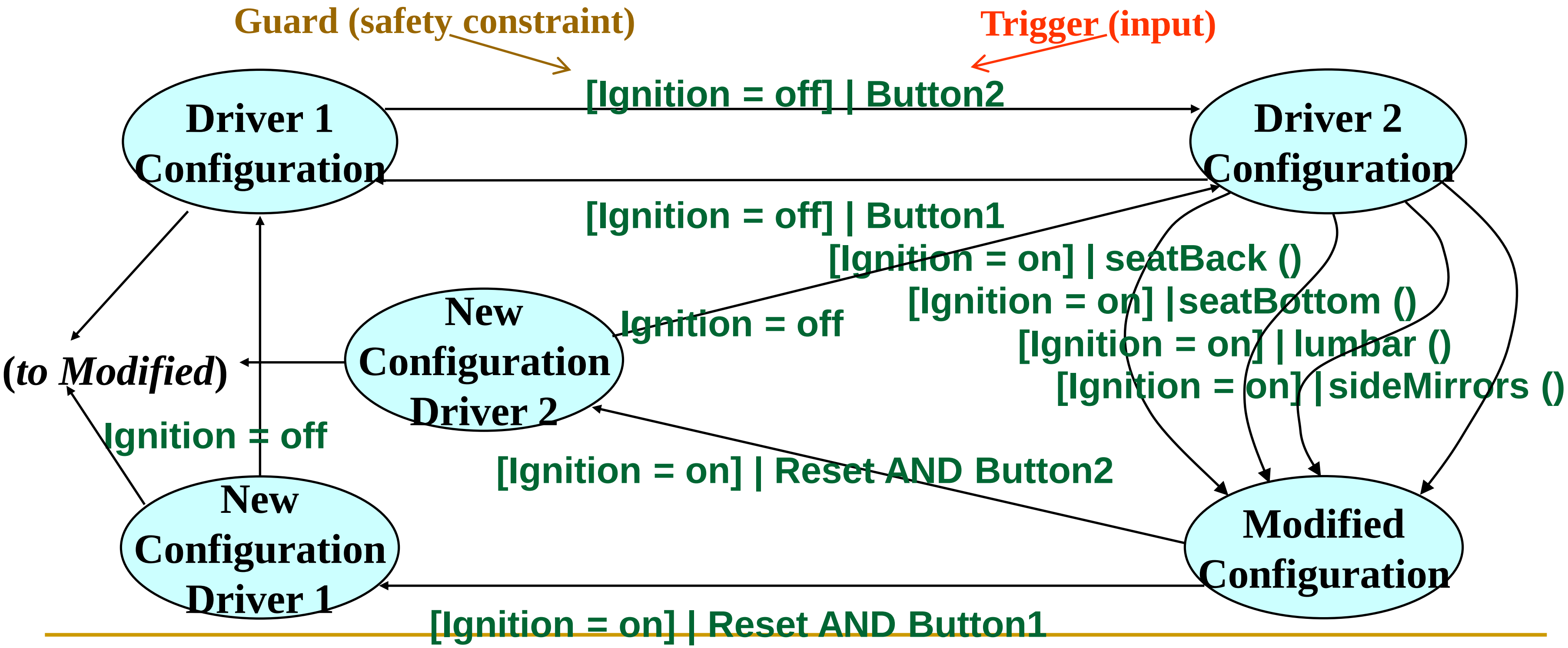
- <u>defs</u>: nodes & edges where variables get values
- <u>uses</u>: nodes & edges where values are accessed

<u>All Uses</u>

Every def "reaches" every use

- **12567**
- **1257**
- **13567**
- **1357**
- **134357**

1. Graph - FSM Example
Memory Seats in a Lexus ES 300



2. Logical Expressions

((a > b) or G) and (x < y)

Transitions

Program Decision Statements

Software Specifications

Logical Expressions

2. Logical Expressions

((a > b) or G) and (x < y)

- Predicate Coverage : Each predicate must be true and false
  - *((a>b) or G) and (x < y)* = True, False
- Clause Coverage : Each clause must be true and false
  - *(a > b)* = True, False
  - *G* = True, False
  - *(x < y)* = True, False
- Combinatorial Coverage : Various combinations of clauses
  - *Active Clause Coverage*: Each clause must determine the predicate’s result

2. Logic – Active Clause Coverage

((*a* > *b*) or *G*) and (*x* < *y*)

	$a > b$	G	$x < y$
1	T	F	T
2	F	F	T
3	F	T	T
4	F	F	T
5	T	T	T
6	T	T	F

With these values for G and (x<y), (a>b) determines the value of the predicate

duplicate

3. Input Domain Characterization

- Describe the input domain of the software
  - Identify inputs, parameters, or other categorization
  - Partition each input into finite sets of representative values
  - Choose combinations of values
- System level
  - Number of students { *0, 1, >1* }
  - Level of course { *600, 700, 800* }
  - Major { *swe, cs, isa, infs* }
- Unit level
  - Parameters *F (int X, int Y)*
  - Possible values X: *{ <0, 0, 1, 2, >2 }, Y : { 10, 20, 30 }*
  - Tests
    - *F (-5, 10), F (0, 20), F (1, 30), F (2, 10), F (5, 20)*

4. Syntactic Structures

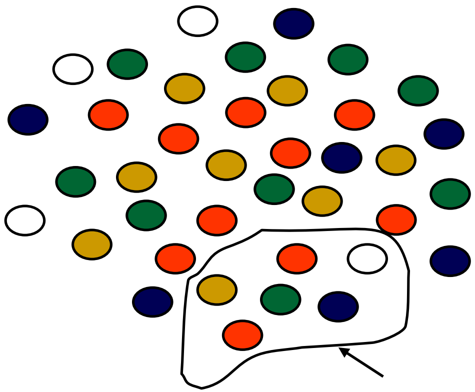
- Based on a grammar, or other syntactic definition
- Primary example is mutation testing
  1. Induce small changes to the program: mutants
  2. Find tests that cause the mutant programs to fail: killing mutants
  3. Failure is defined as different output from the original program
  4. Check the output of useful tests on the original program
- Example program and mutants

```
if (x > y)
    z = x - y;
else
   z = 2 * x;
```

```
if (x > y)
Δif (x >= y)
    z = x - y;
    Δ z = x + y;
    Δ z = x – m;
else
   z = 2 * x;
```

Coverage

Given a set of test requirements *TR* for coverage criterion *C*, a test set *T* satisfies *C* coverage if and only if for every test requirement *tr* in *TR*, there is at least one test *t* in *T* such that *t* satisfies *tr*

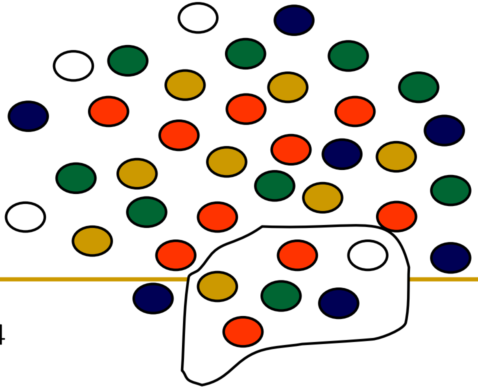


Test set satisfies TR

Suppose that TR consists of five test requirement, each represented in a different color.

Coverage

- Infeasible test requirements : test requirements that cannot be satisfied
  - No test case values exist that meet the test requirements
  - Dead code
  - Detection of infeasible test requirements is formally undecidable for most test criteria
- Thus, 100% coverage is impossible in practice



Suppose that TR consists of five test requirement, each represented in a different color.

Two Ways to Use Test Criteria

1. Directly generate test values to satisfy the criterion
   - Often assumed by the research community
   - Most obvious way to use criteria
   - Very hard without automated tools
2. Generate test values externally and measure against the criterion usually favored by industry
   - sometimes misleading
   - if tests do not reach 100% coverage, what does that mean?

Test criteria are sometimes called metrics

Generators and Recognizers

- Generator : A procedure that automatically generates values to satisfy a criterion (example: Evosuite)
- Recognizer : A procedure that decides whether a given set of test values satisfies a criterion (example: Randoop)
- Both problems are provably undecidable for most criteria
- It is possible to recognize whether test cases satisfy a criterion far more often than it is possible to generate tests that satisfy the criterion
- Coverage analysis tools are quite plentiful

Comparing Criteria with 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
- *Example* - Branch coverage subsumes statement coverage
  - If a test set covers every branch in a program (satisfies the branch coverage criterion), the test set is guaranteed to also cover every statement (satisfies the statement coverage criterion).

Test Coverage Criteria

- Traditional software testing is expensive and labor-intensive
- Coverage criteria are used to decide which test inputs to use
- More likely that the tester will find problems
- Greater assurance that the software is of high quality and reliability
- A goal or stopping rule for testing
- Criteria makes testing more efficient and effective

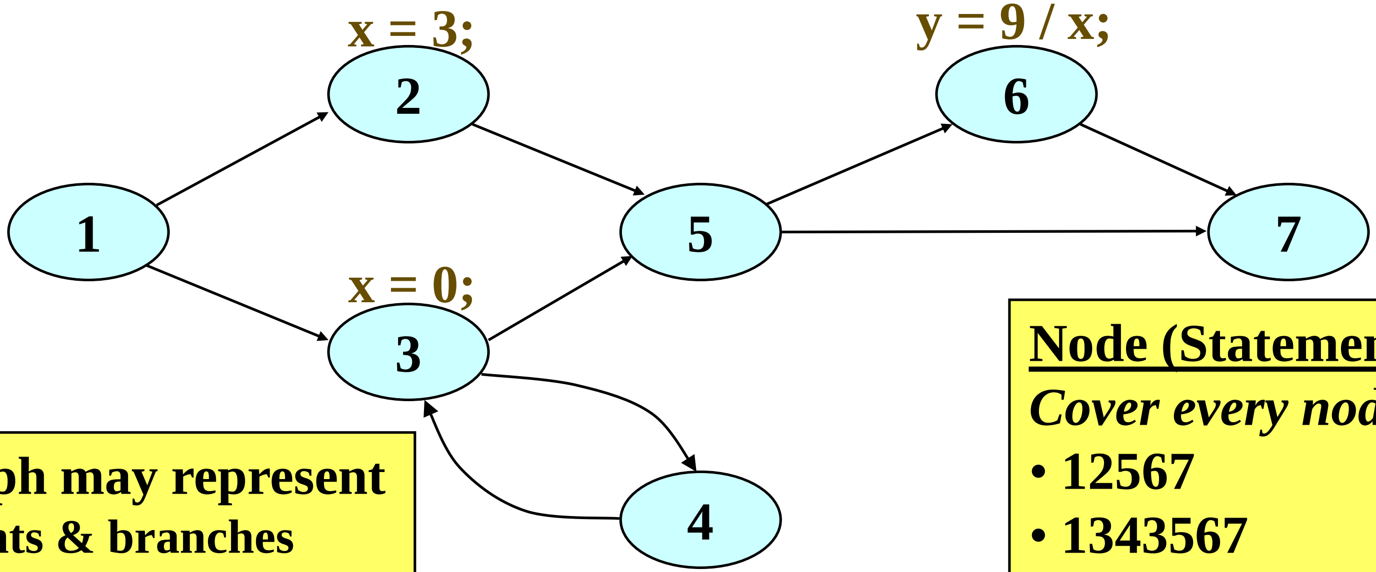
Are there issues we need to be aware of in practice?

Exercise

- Suppose that coverage criterion **C1 subsumes criterion C2**. Further suppose that test set **T1 satisfies C1** and test set **T2 satisfies C2** on the same program.
  - Does T1 necessarily satisfy C2?
  - Does T2 necessarily satisfy C1?
  - If T2 reveals a fault, does T1 necessarily reveal the fault?

C1 subsumes C2 $\wedge$
Cov(T1, C1) $\wedge$
Cov(T2, C2)

1. Graph Coverage – Structural



Node (Statement)

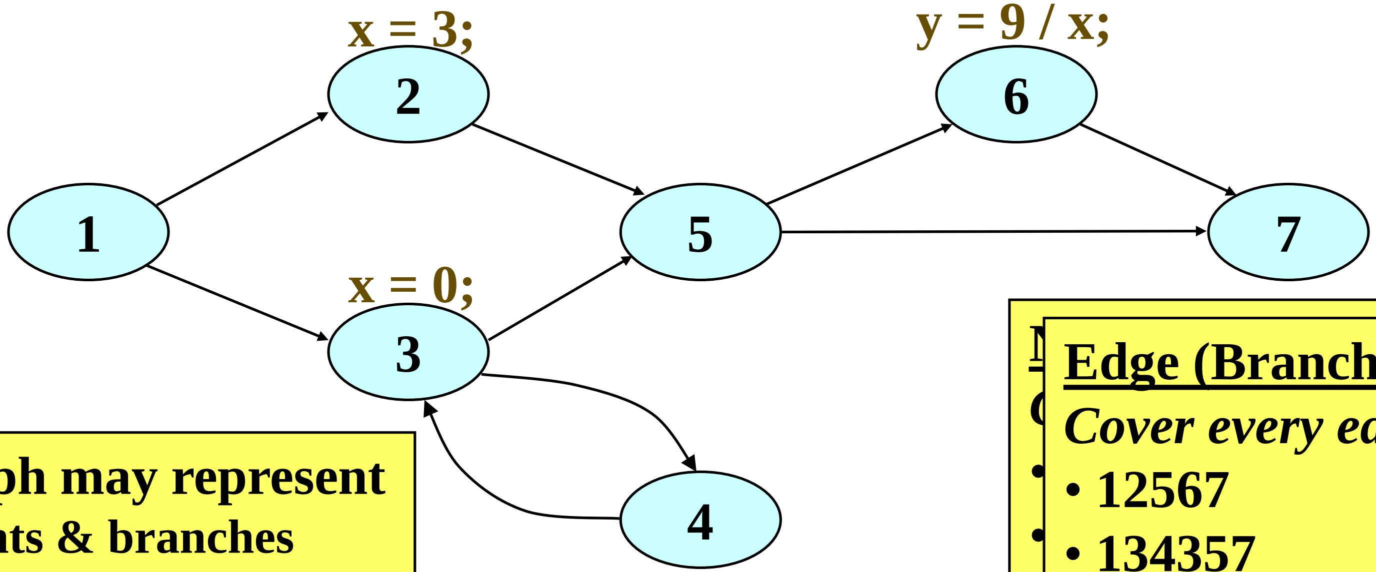
Cover every node

- **12567**
- **1343567**

This graph may represent
- **statements & branches**
- **methods & calls**
- **components & signals**
- **states and transitions**

⋮

1. Graph Coverage – Structural



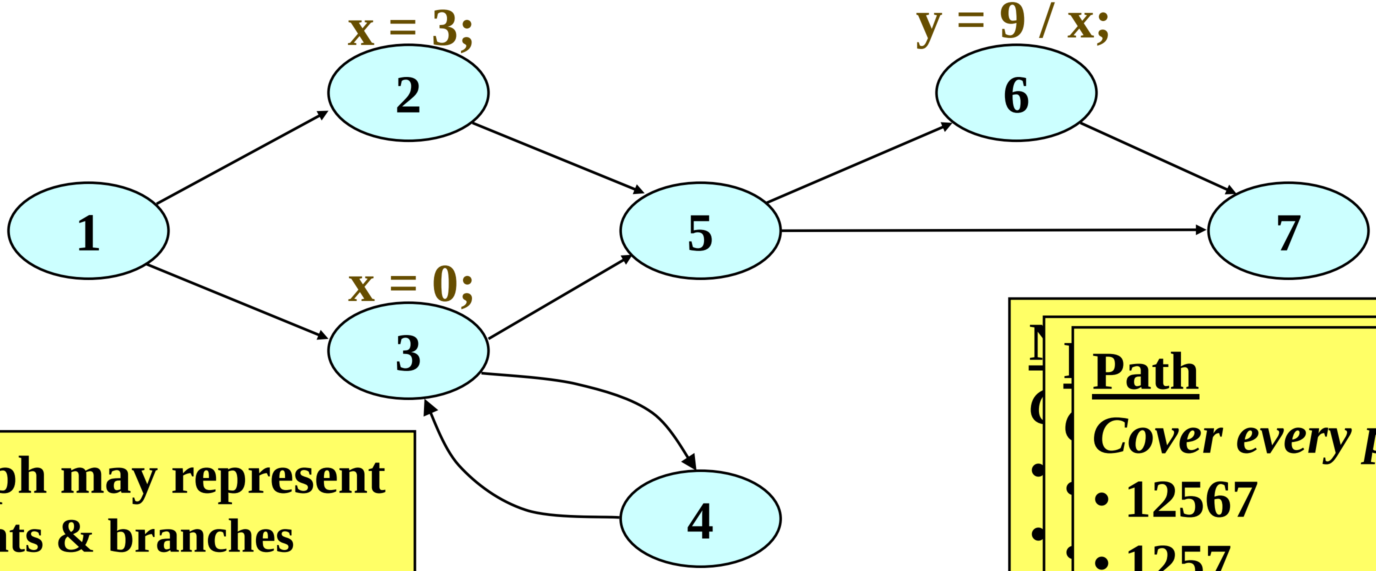
This graph may represent
- **statements & branches**
- **methods & calls**
- **components & signals**
- **states and transitions**

⋮

Edge (Branch)

Cover every edge
- **12567**
- **134357**

1. Graph Coverage – Structural



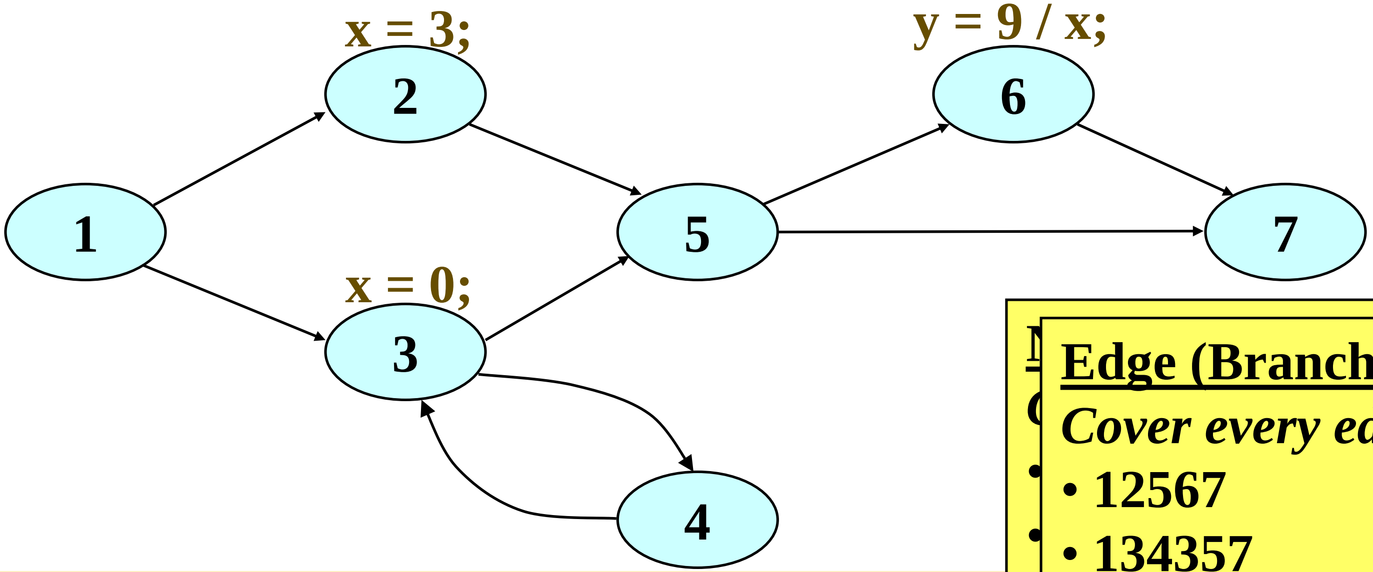
This graph may represent
- **statements & branches**
- **methods & calls**
- **components & signals**
- **states and transitions**

⋮

Path

Cover every path
- **12567**
- **1257**
- **13567**
- **1357**
- **1343567**
- **134357 …**

1. Graph Coverage – Structural



Edge (Branch)

Cover every edge

- **12567**
- **134357**

Both node and edge coverages are weakly correlated to fault detection capabilities

Can we have other coverages that are more correlated to fault detection?

Un-assessed Homework

- Do homework Exercise Test1 under self-assessment center after class.
- Please don't look at the solution before trying the problem yourself.

Testing - Basic

- Exercise
- Solution

Further Reading

- Ammann & Offutt, Chapter 1

Testing - Graph Coverage

- Exercise & Solution
- Exercise (Q1 on page 16) & Solution

Further Reading

- Ammann & Offutt, Chapter 2.1 & 2.2

GRAPH COVERAGE CRITERIA

Slides adapted from www.introsoftwaretesting.com by Paul Ammann & Jeff Offutt

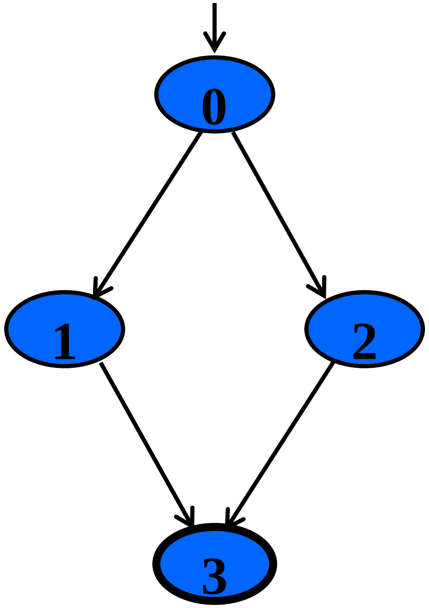
Covering Graphs

- 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 in some way

Definition of a Graph

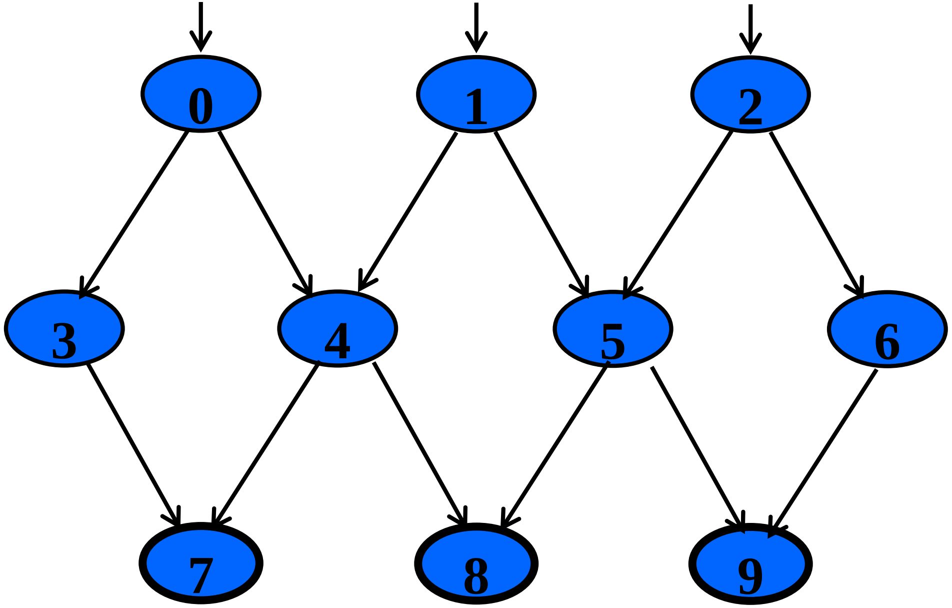
- A set N of nodes, N is not empty
- A set N_0 of initial nodes, N_0 is not empty
- A set N_f of final nodes, N_f is not empty
- A set E of edges, each edge from one node to another
  - (n_i, n_j), i is predecessor, j is successor

Three Example Graphs



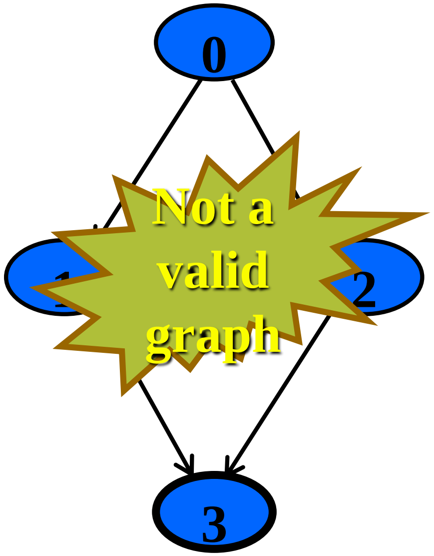
$N_0 = \{ 0 \}$

$N_f = \{ 3 \}$



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

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



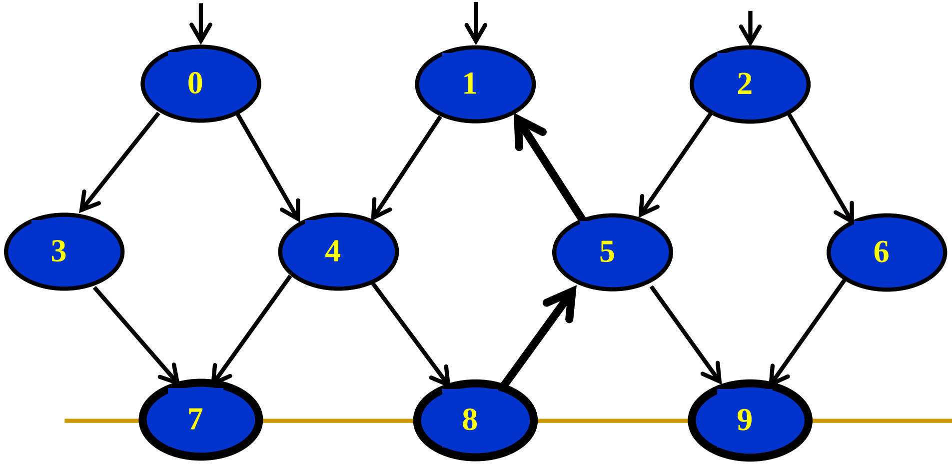
$N_0 = \{ \}$

$N_f = \{ 3 \}$

Paths in Graphs

- Path : A sequence of nodes – [n_1, n_2, …, 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*

Why are we interested in the **reach** relation?



Paths

[0, 3, 7]

[1, 4, 8, 5, 1]

[2, 6, 9]

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

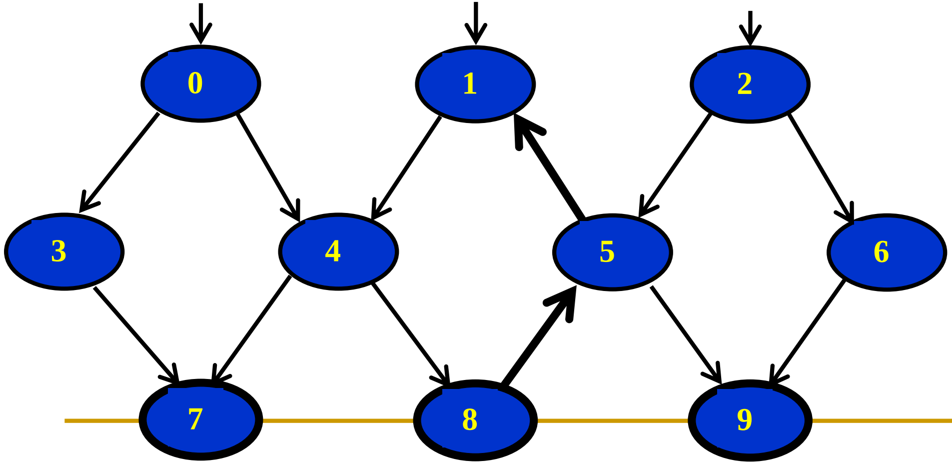
Reach ({0, 2}) = G

Reach([2,6]) = {2, 6, 9}

Why Coverage Works?

Recall the Reachability-Infection-Propagation (RIP) model
Basic assumption of coverage:
- To validate whether a test requirement is faulty is to cover it!

- 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 }

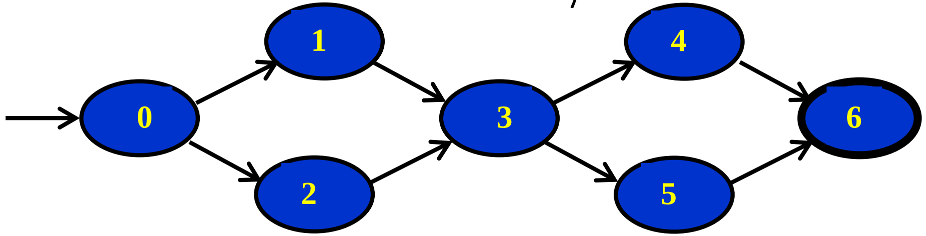
Reach ({0, 2}) = G

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
  - No and Nf have exactly one node

Structured Programming



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

- <u>Visit</u> : A test path *p* <u>*visits*</u> node *n* if *n* is in *p*

  A test path *p* <u>*visits*</u> edge *e* if *e* is in *p*

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

Path [0, 1, 3, 4, 6]

Visits nodes 0, 1, 3, 4, 6

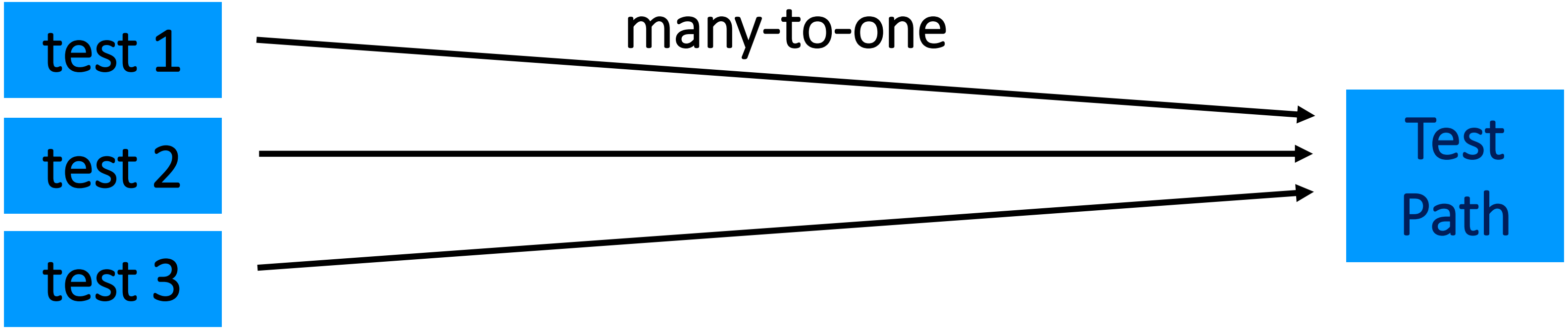
Visits edges (0, 1), (1, 3), (3, 4), (4, 6)

Tours subpaths (0, 1, 3), (1, 3, 4), (3, 4, 6), (0, 1, 3, 4), (1, 3, 4, 6)

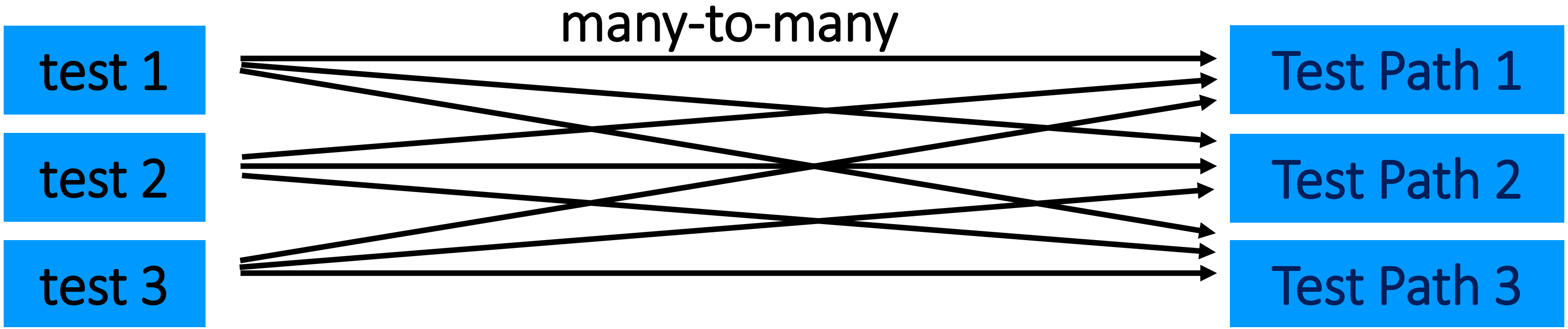
Tests and Test Paths

- 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

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

- 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
- 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

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

<u>Node Coverage (NC)</u> : 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

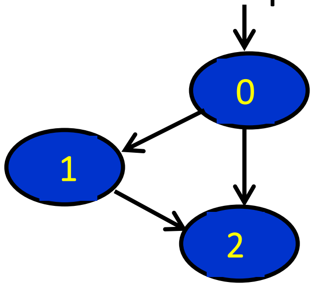
<u>Node Coverage (NC)</u> : TR contains each reachable node in G.

Node and Edge Coverage

- Edge coverage is slightly stronger than node coverage

Edge Coverage (EC) : 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)

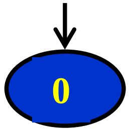


Node Coverage : TR = { 0, 1, 2 }
Test Path = [0, 1, 2]

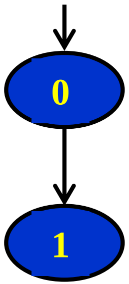
Edge Coverage : TR = { (0,1), (0, 2), (1, 2) }
Test Paths = [0, 1, 2], [0, 2]

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

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

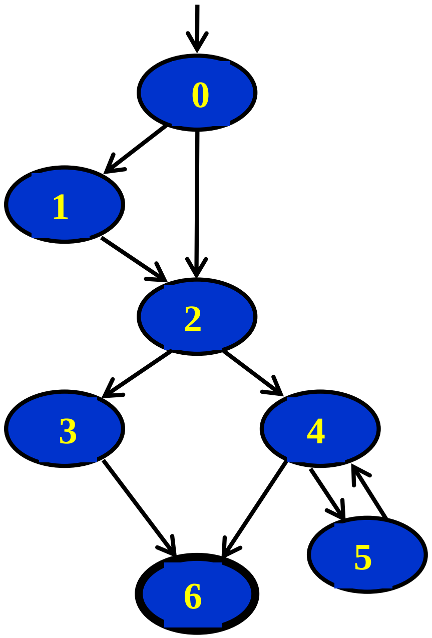
- The "length up to 2" is used to include graphs that have less than 2 edges
- The logical extension is to require all paths …

<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 is to make 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 = { 0, 1, 2, 3, 4, 5, 6 }

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

Edge Coverage

TR = { (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 = { [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

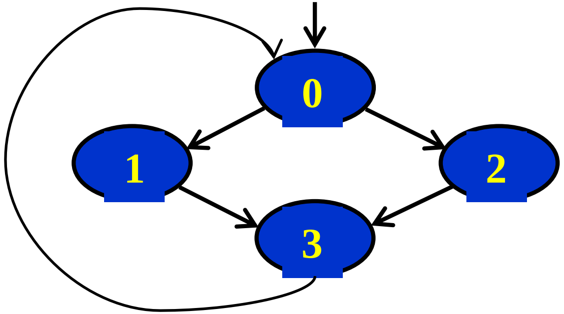
- If a graph contains a loop, it has an infinite number of paths
- Thus, CPC is not feasible
- SPC is not satisfactory because the results are subjective and vary with testers
- Attempts to "deal with" loops:
  - 1970s : Execute cycles once ([4, 5, 4] in previous example, informal)
  - 1980s : Execute 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
  - May include other subpaths
  - A loop is a simple path

Any paths can be created by composing simple paths!

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

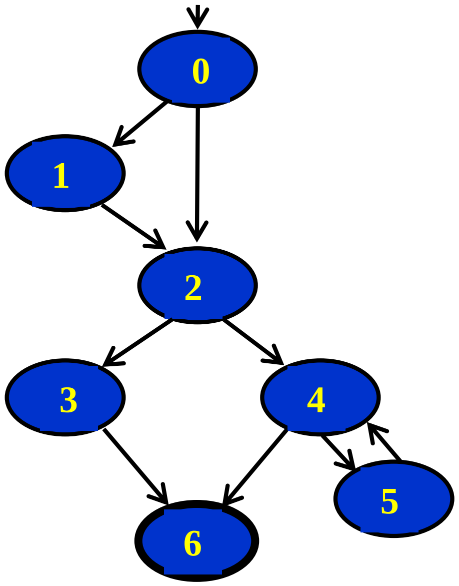


Simple Paths : [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1], [0, 1, 3], [0, 2, 3], [1, 3, 0], [2, 3, 0], [3, 0, 1], [3, 0, 2], [0, 1], [0, 2], [1, 3], [2, 3], [3, 0], [0], [1], [2], [3]

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]

More Prime Path Example

- The following graph has 38 simple paths
- Only nine *prime paths*



<u>Prime Paths</u>

[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

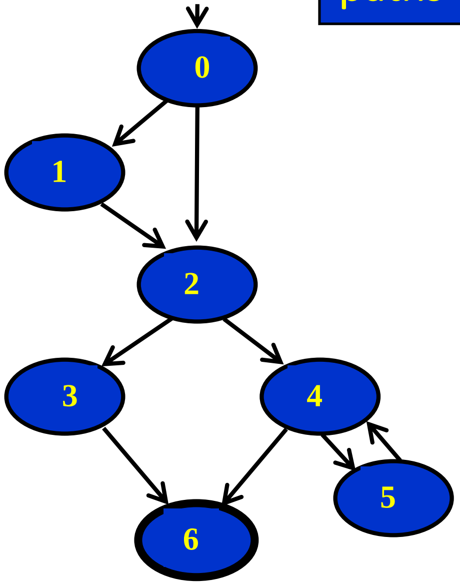
Execute loop at least once

Execute loop more than once

Finding Prime Paths

'!' means path terminates

'*' means path cycles



Simple paths

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

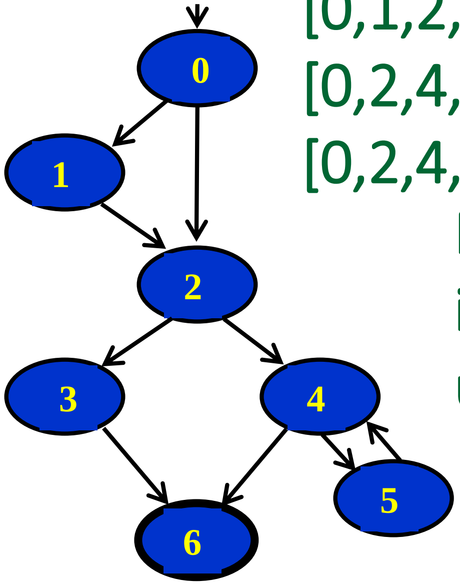
Prime Path Coverage

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

Prime Path Coverage (PPC) : 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**

Test Set Construction for Prime Path Coverage



T = {[0,1,2,3,6], [0,1,2,4,6], [0,1,2,4,5,4,6], [0,2,3,6], [0,2,4,6], [0,2,4,5,4,6], [0,2,4,5,4,5,4,6]}

Note that test set is not necessarily unique.

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

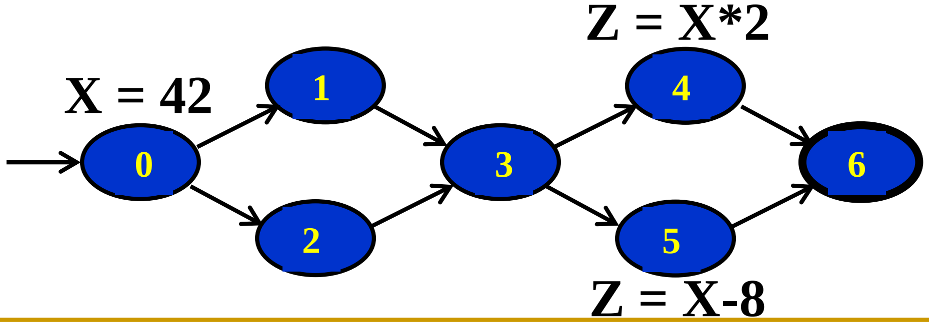
Infeasible Test Requirements

- 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

Data Flow Criteria

Goal: Try to ensure that values are computed and used correctly

- Definition (def) : A location where a value for a variable is stored into memory, e.g., x = 42;
- Use : A location where a variable’s value is accessed, e.g., z = x;
- 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

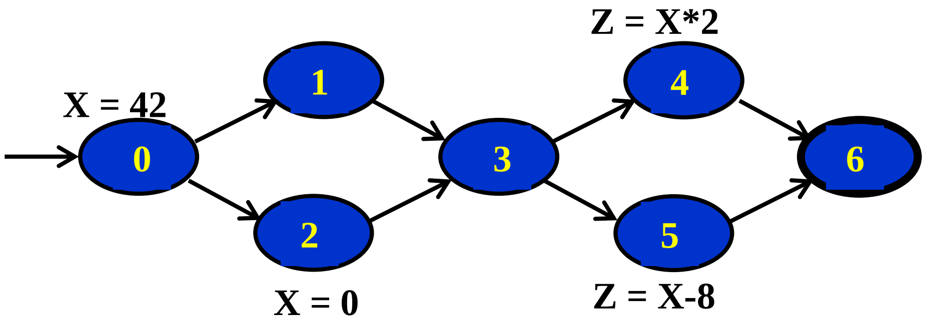


Defs: def (0) = {X}
def (4) = {Z}
def (5) = {Z}
Uses: use (4) = {X}
use (5) = {X}

DU Pairs and DU Paths

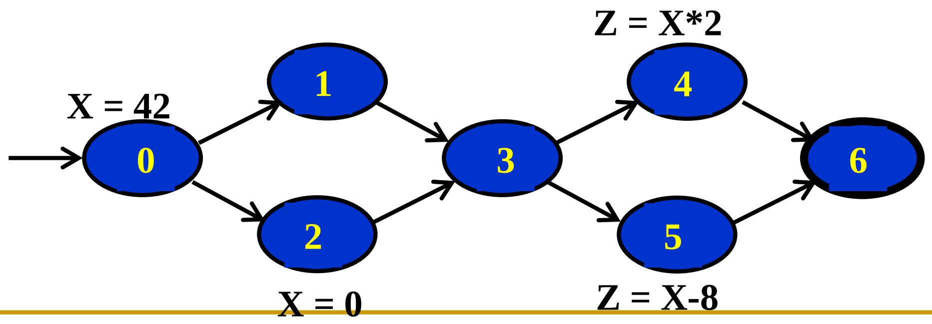
- DU pair : A pair of locations (l_i, l_j) such that a variable *v* is defined at l_i and used at l_j

- (0, 4), (0, 5), (2, 4), (2, 5) are DU pairs of X



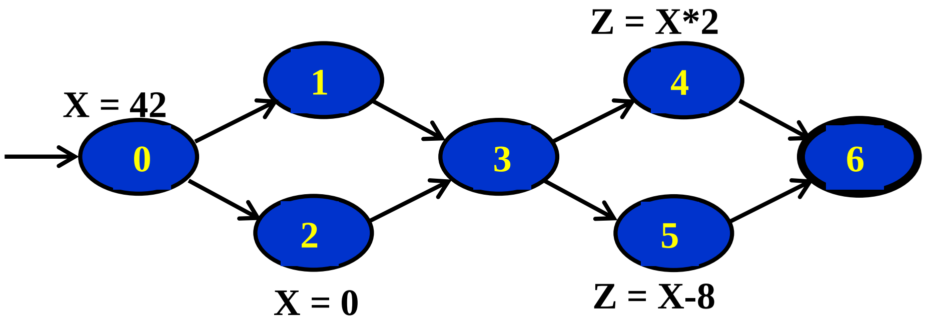
DU Pairs and DU Paths

- 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
- Examples: [0, 1, 3, 4], [2, 3, 5]
- Counter-examples: [0, 2, 3, 4], [0, 2, 3, 5]



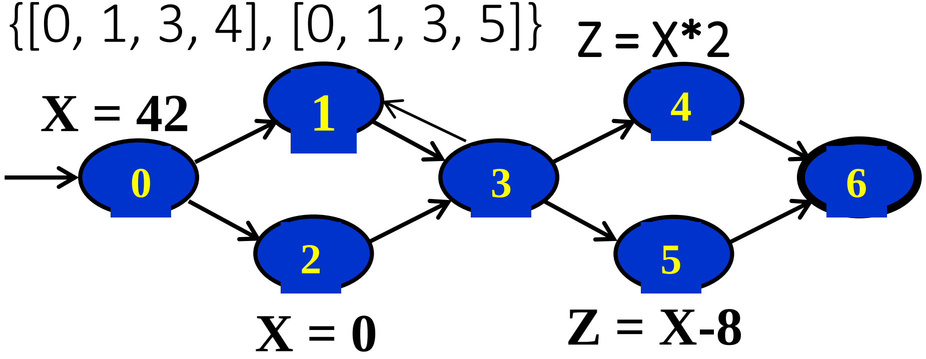
DU Pairs and DU Paths

- 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
- Examples: Reach(0) = {4, 5}; Reach(2) = {4, 5}



DU Pairs and DU Paths

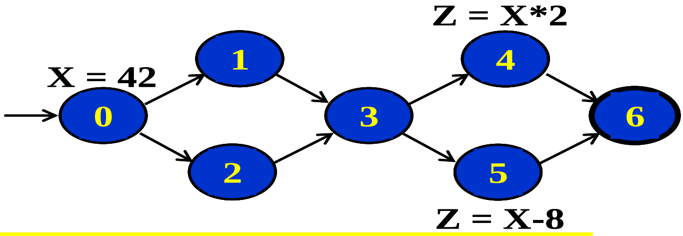
- du-path : A simple path that is def-clear with respect to *v* from a def of *v* to a use of *v*
- du (n_i, n_j, v) – the set of du-paths from n_i to n_j
  - du(0, 4, X) = [0, 1, 3, 4]; du(0, 5, X) = [0, 1, 3, 5]
- du (n_i, v) – the set of du-paths that start at n_i
  - du(0, X) = {[0, 1, 3, 4], [0, 1, 3, 5]}



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*
- 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

> <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

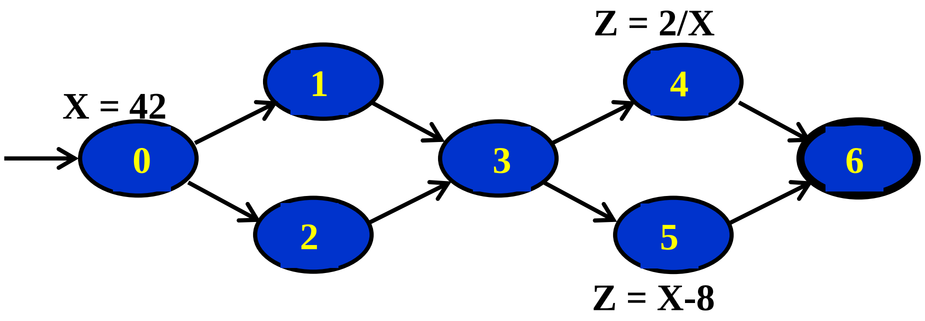
> <u>All-uses coverage (AUC)</u> : For each set of du-paths to uses *S* = *du* (n_i, n_j, *v*), TR contains at least one path *d* in *S*.

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

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

Data Flow Testing Example

DU pairs:
(X,0,4)
(X,0,5)



All-defs for *X*
[0, 1, 3, 4] or [0, 1, 3, 5] or …

Every def reaches a use

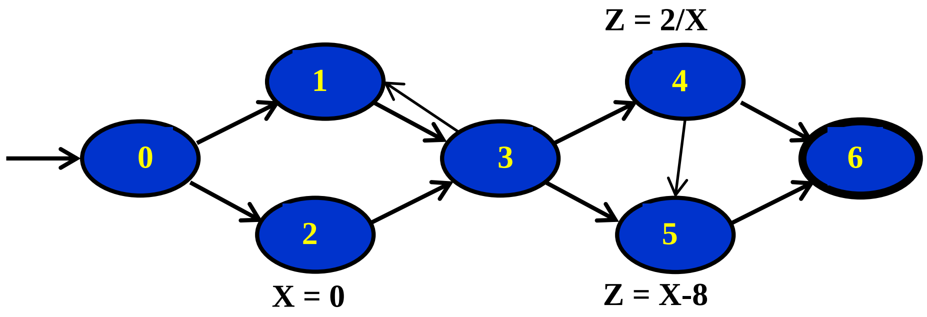
All-uses for *X*
[0, 1, 3, 4]
[0, 1, 3, 5]

Every def reaches all possible uses

All-du-paths for *X*
[0, 1, 3, 4]
[0, 2, 3, 4]
[0, 1, 3, 5]
[0, 2, 3, 5]

All possible paths between defs and their possible uses

Data Flow Testing Exercise

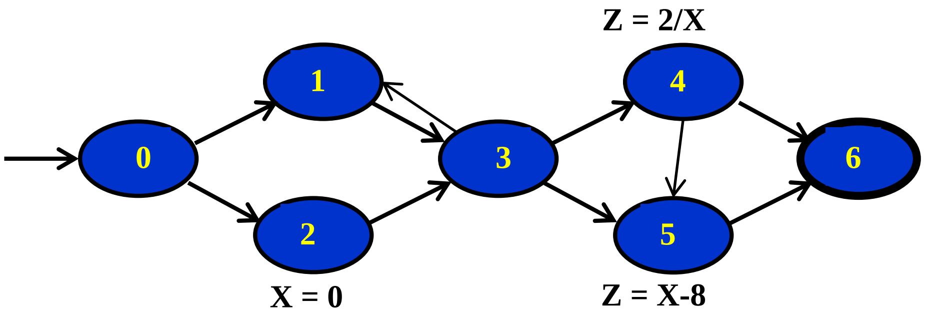


All-defs for *X*

All-uses for *X*

All-du-paths for *X*

Data Flow Testing Exercise

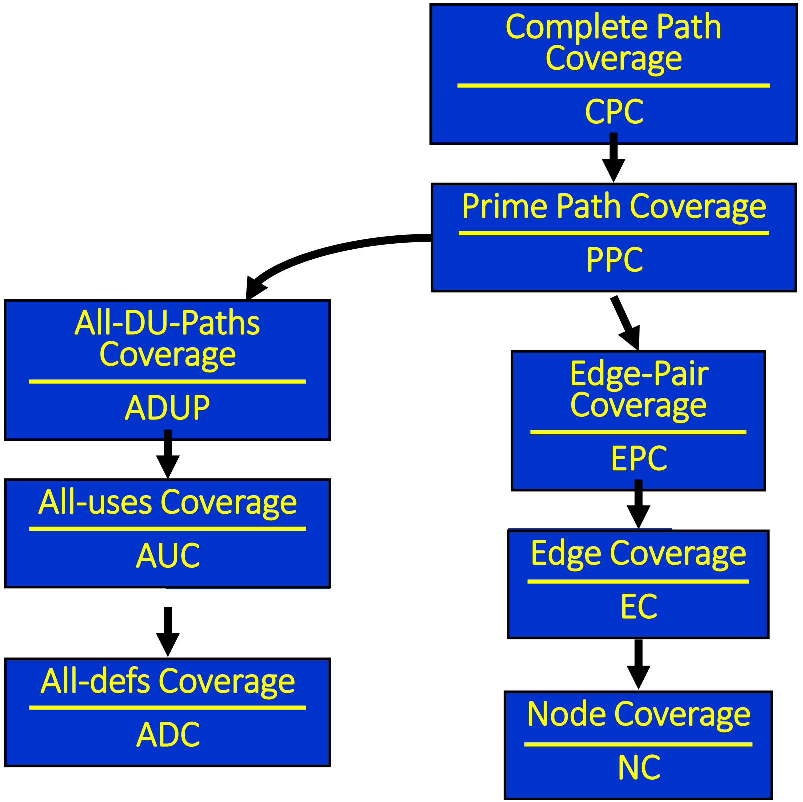


All-defs for *X*
[2, 3, 4]
or
[2, 3, 5]
or
[2, 3, 4, 5]

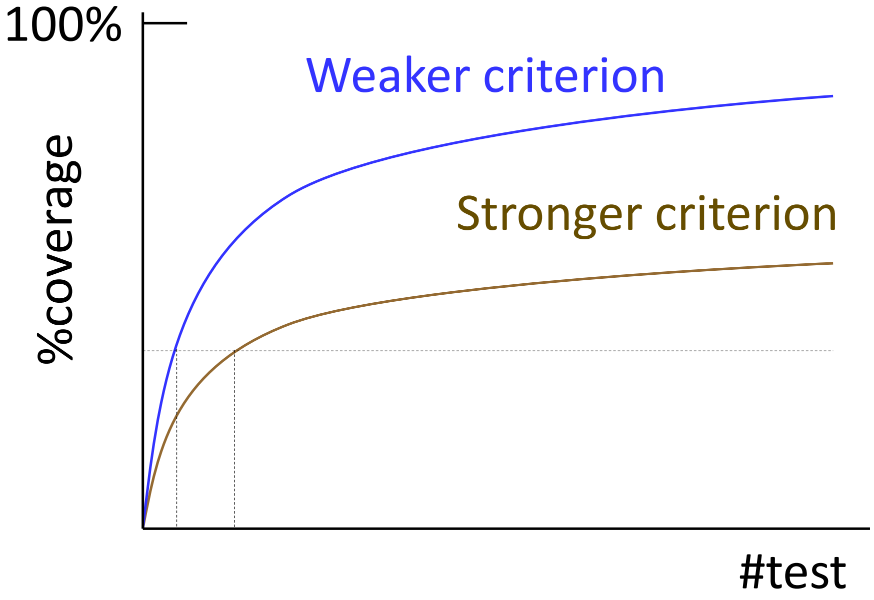
All-uses for *X*
[2, 3, 4]
[2, 3, 5]
or
[2, 3, 4, 5]

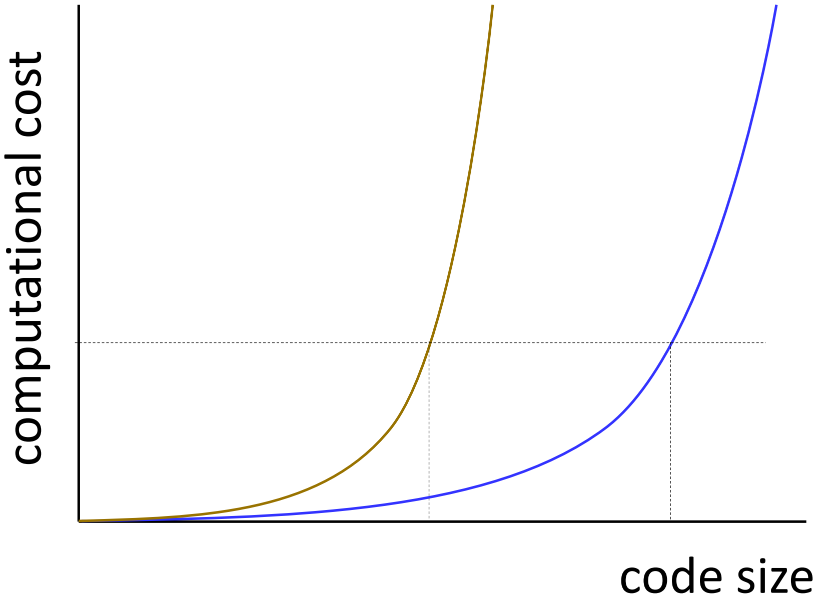
All-du-paths for *X*
~~[2, 3, 4]~~
[2, 3, 5]
[2, 3, 4, 5]

Graph Coverage Criteria Subsumption

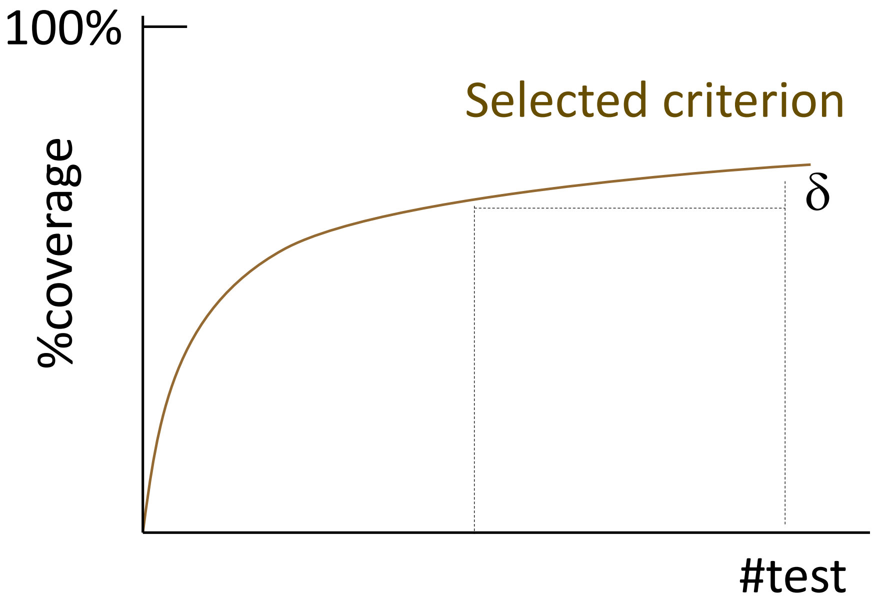


Saturation Coverage Policy



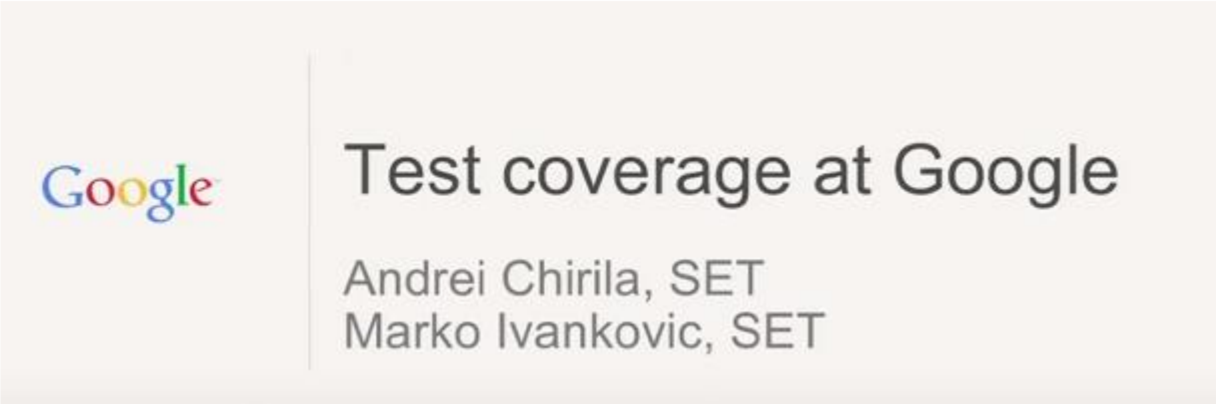


Saturation Coverage Policy



- Saturation coverage can be used as a termination criterion for testing
- Stop testing when cov(2n) - cov(n) < δ

Test Coverage



https://www.youtube.com/watch?v=4bublRBCLVQ

Also available at: https://hkustconnect-my.sharepoint.com/:v:/g/personal/sccheung_connect_ust_hk/EeD98fs7o_VKpuqbn8AdJW8BAice4fvdsRLQz1ct7HF1fw?e=N4TVJB