Introduction to Software Testing
(2nd edition)
Chapter 7

Graph Coverage Criteria

Instructor: Morteza Zakeri

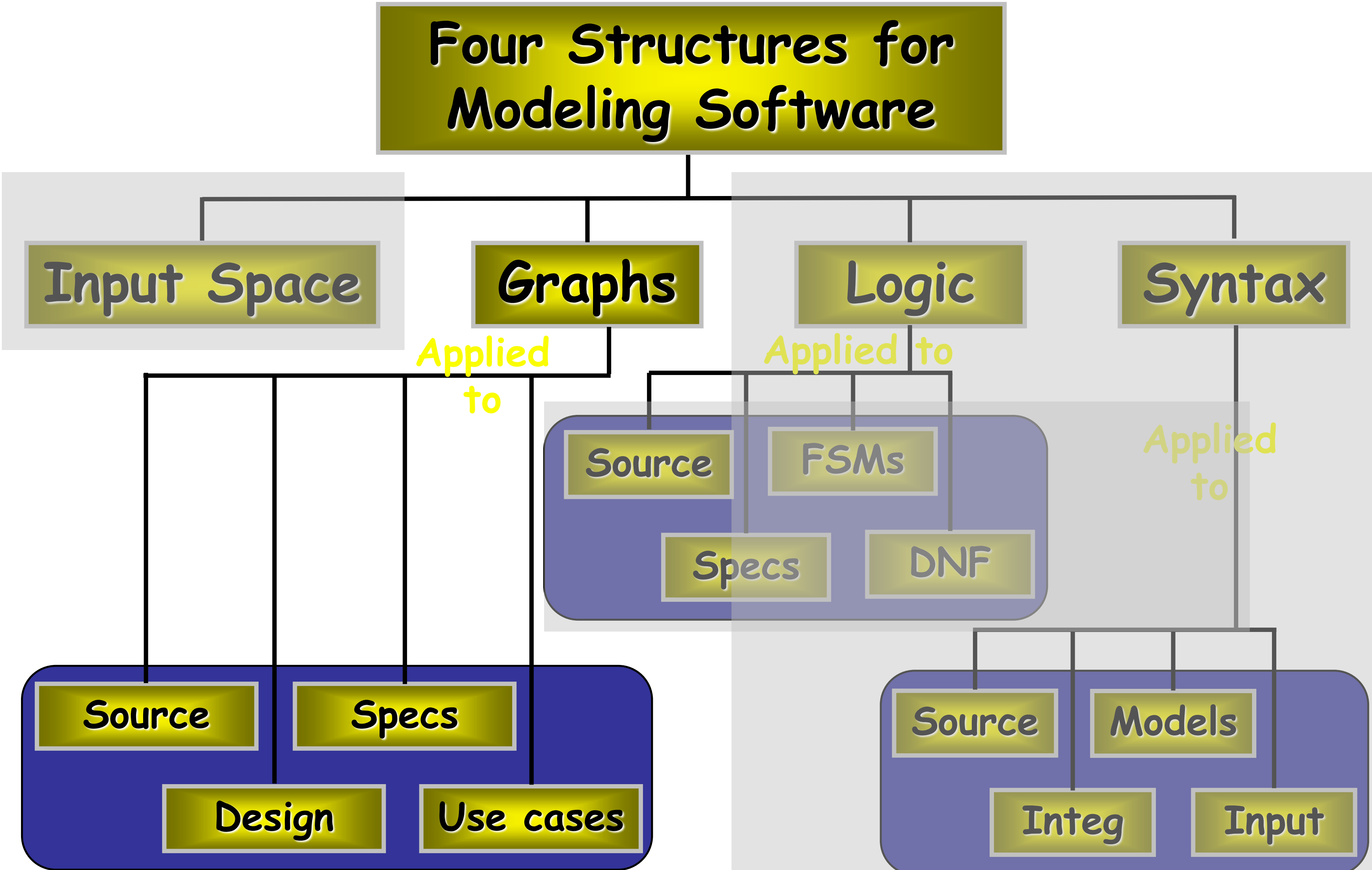
Slides by: **Paul Ammann & Jeff Offutt**

http://www.cs.gmu.edu/~offutt/softwaretest/

Modified by: **Morteza Zakeri**

March 2024

Ch. 7 : Graph Coverage



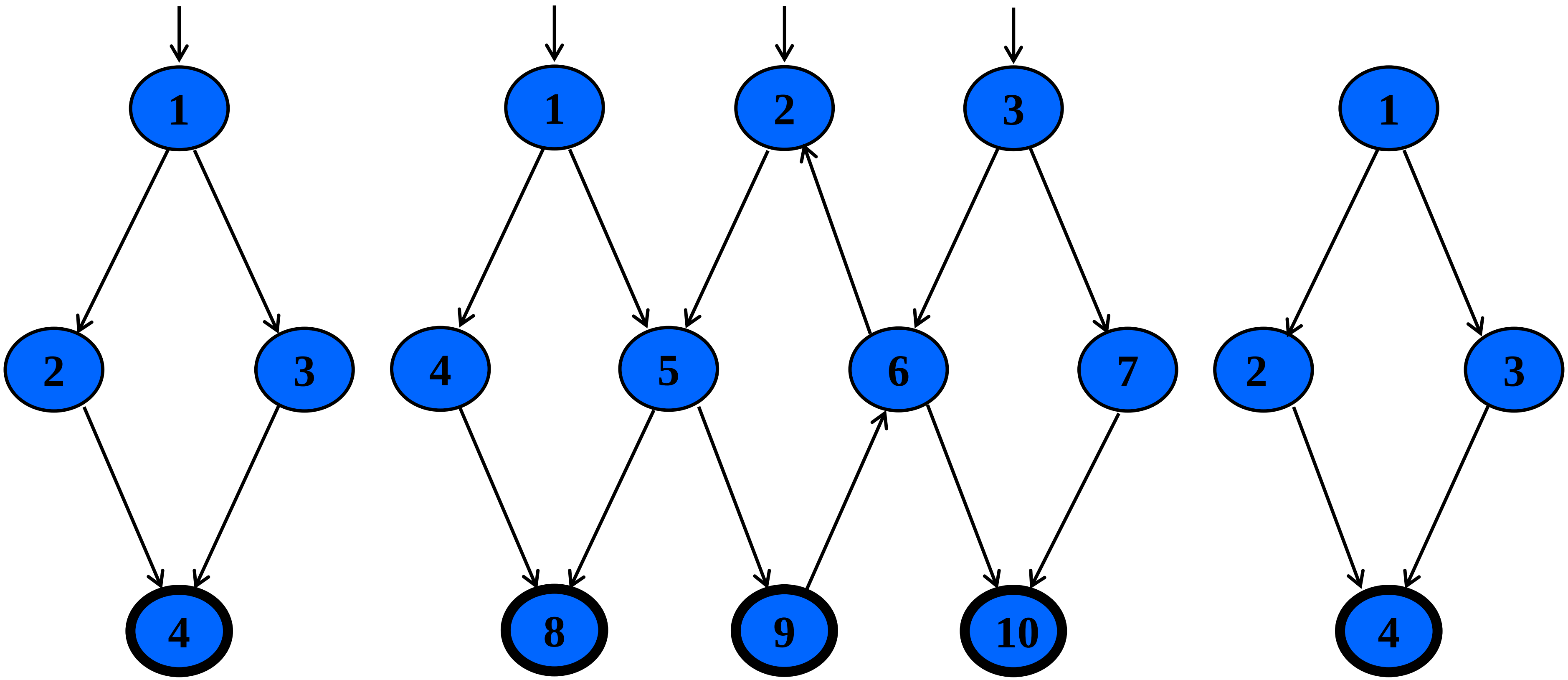
Covering Graphs (7.1)

- Graphs are the most commonly used structure for testing
- Graphs can come from many sources
  - Control Flow Graphs (CFGs)
  - Design structure (UML Class Diagram)
  - FSMs and State Charts
  - 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

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 }

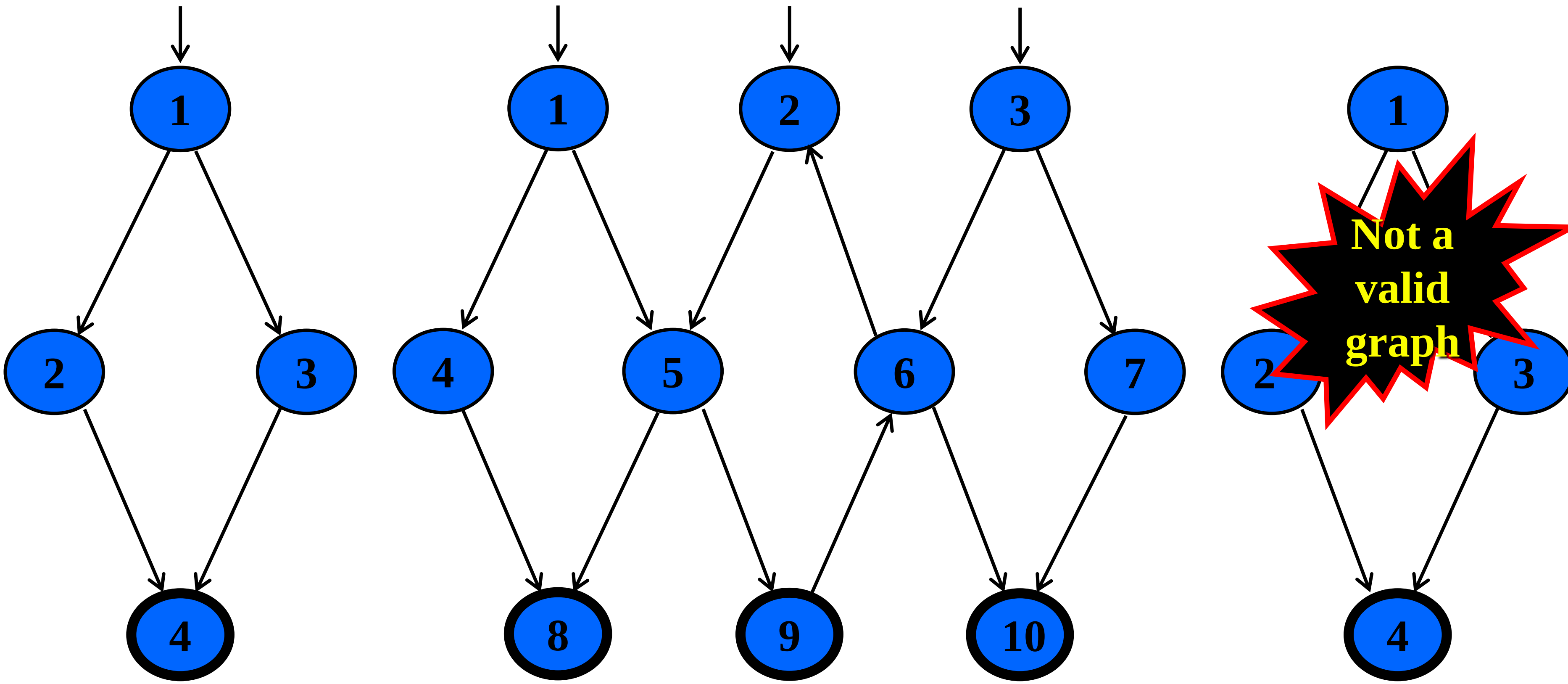
E = { (1,4), (1,5), (2,5), (3,6), (3, 7), (4, 8), (5,8), (5,9), (6,2), (6,10), (7,10) (9,6) }

N_0 = { }

N_f = { 4 }

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

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) }

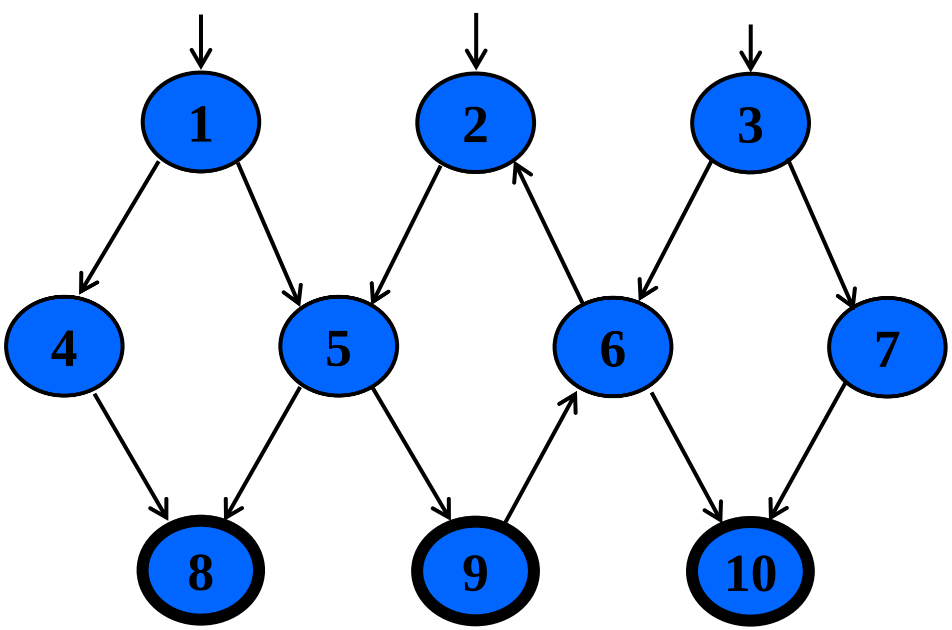
$N_0 = \{ \}$

$N_f = \{ 4 \}$

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

Paths in Graphs

- Path: A sequence of nodes, *i.e.*, [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*



A Few Paths

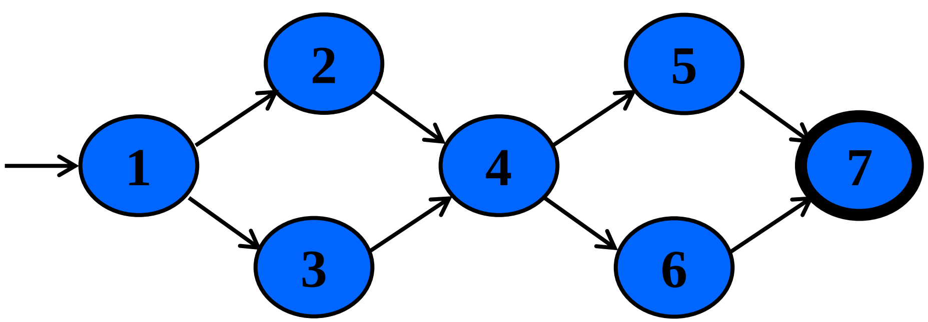
[1, 4, 8]

[2, 5, 9, 6, 2]

[3, 7, 10]

Test Paths and SESEs

- Test Path (execution 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

[1, 2, 4, 5, 7]

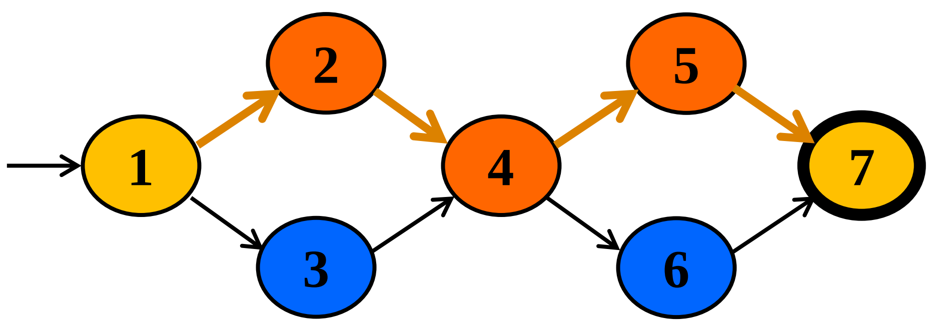
[1, 2, 4, 6, 7]

[1, 3, 4, 5, 7]

[1, 3, 4, 6, 7]

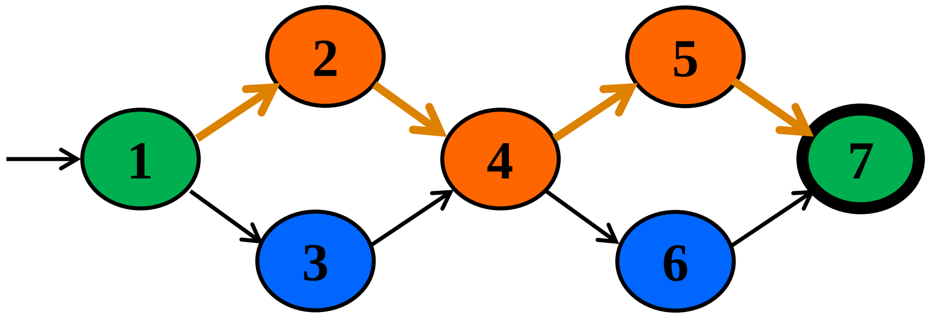
Visiting and Touring

- **Visit (cover):**
  - 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 = {[1, 2, 4, 5, 7]}

Visiting and Touring Example

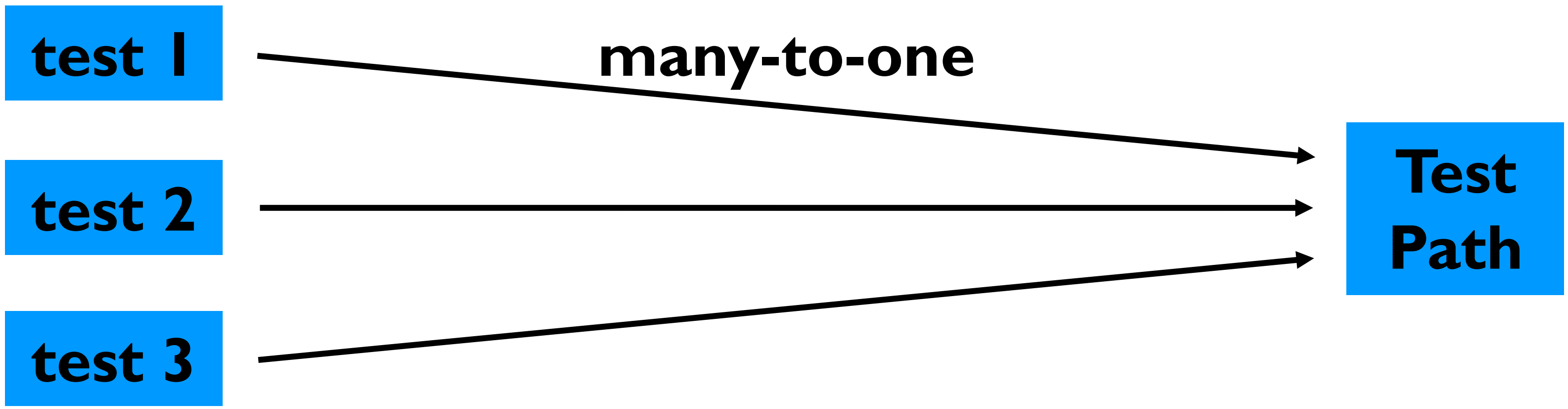


- **Path = {[1, 2, 4, 5, 7]}**
- Visits nodes = {1, 2, 4, 5, 7}
- Visits edges = {(1, 2), (2, 4), (4, 5), (5, 7)}
- Tours subpaths = {[1, 2, 4], [2, 4, 5], [4, 5, 7], [1, 2, 4, 5], [2, 4, 5, 7], [1, 2, 4, 5, 7]}
  - *(Also, each edge, e.g., [1,2], is technically a subpath)*

Tests and Test Paths

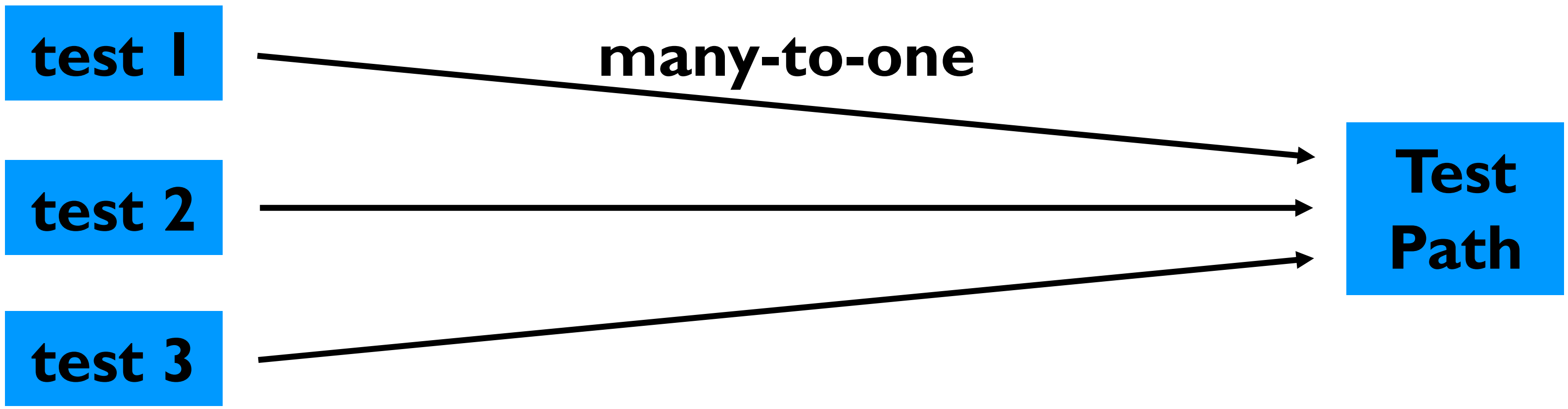
- path(*t*): The test path executed by test *t*
- path(*T*): The set of test paths executed by the set of tests *T*
- Each test executes one and only one test path
  – Complete execution from a start node to an final node
- A location in a graph (node or edge) can be reached from another location if 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 will become important in **section 7.3**

Tests and Test Paths

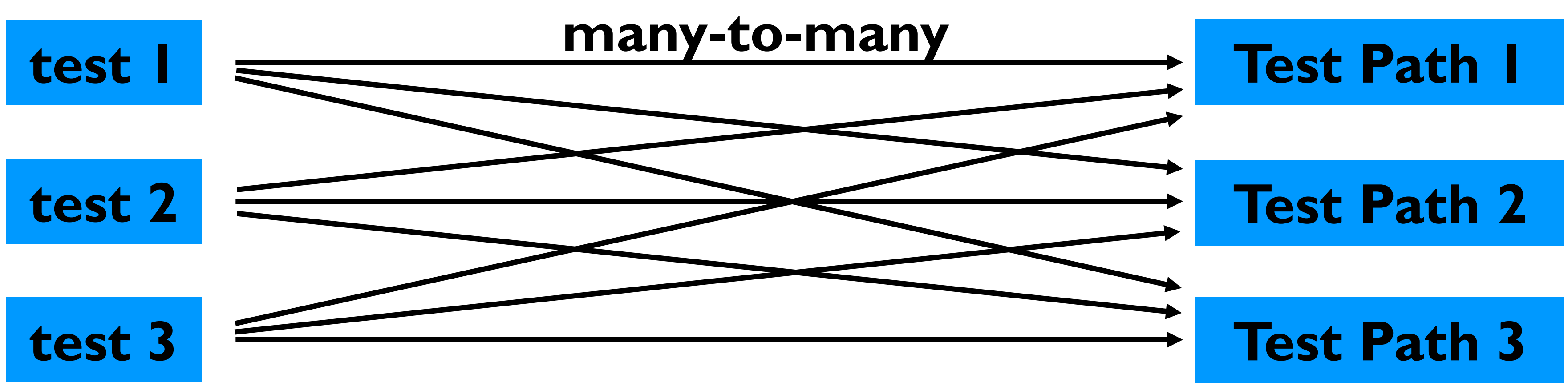


Deterministic software: Always executes the same test path

Tests and Test Paths



Deterministic software: Always executes the same test path



Non-deterministic software: The same test can execute different test paths

Testing and Covering Graphs (7.2)

- We use graphs in testing as follows :
  – Develop a model of the software as a graph
  – Require 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

- 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 in terms of the set of test requirements

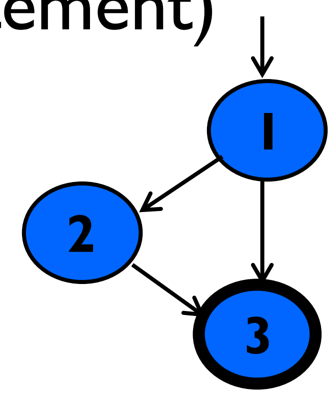
> **<u>Node Coverage (NC)</u>: TR contains each reachable node in G.**

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 phrase "*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)

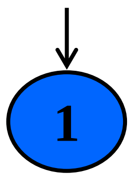


<u>Node Coverage</u>: TR = { 1, 2, 3 }
Test Path = { [1, 2, 3] }

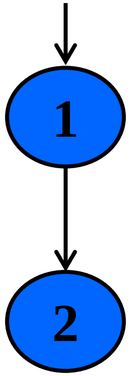
<u>Edge Coverage</u>: TR = { (1, 2), (1, 3), (2, 3) }
Test Paths = { [1, 2, 3], [1, 3] }

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
- 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 …

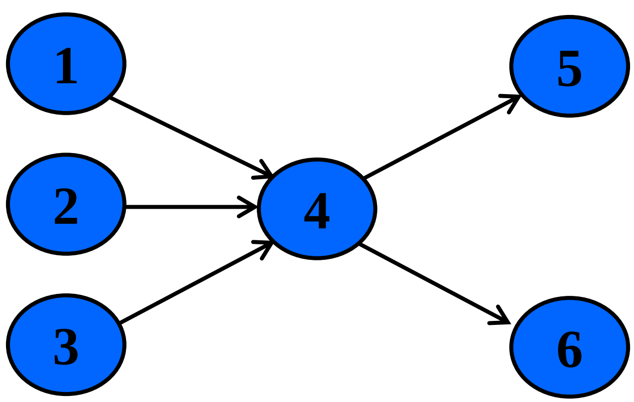


Covering Multiple Edges

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



<u>Edge-Pair Coverage</u>:

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

- The logical extension is to require all paths …

Covering Multiple Edges

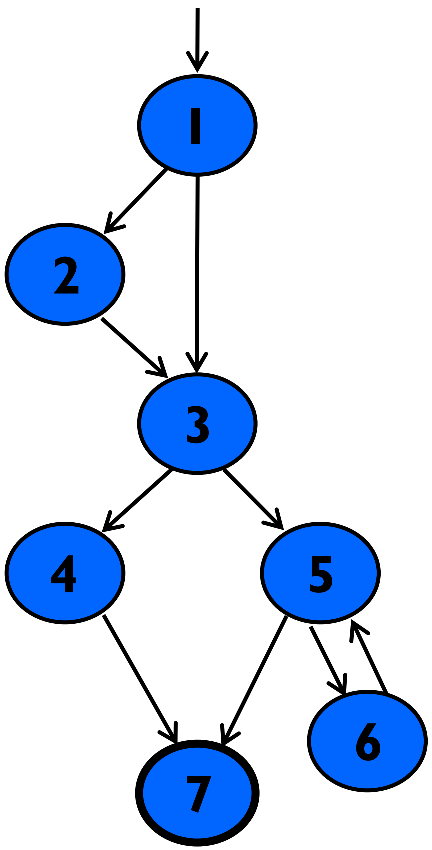
Complete Path Coverage (CPC): **TR contains all paths in G.**

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

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

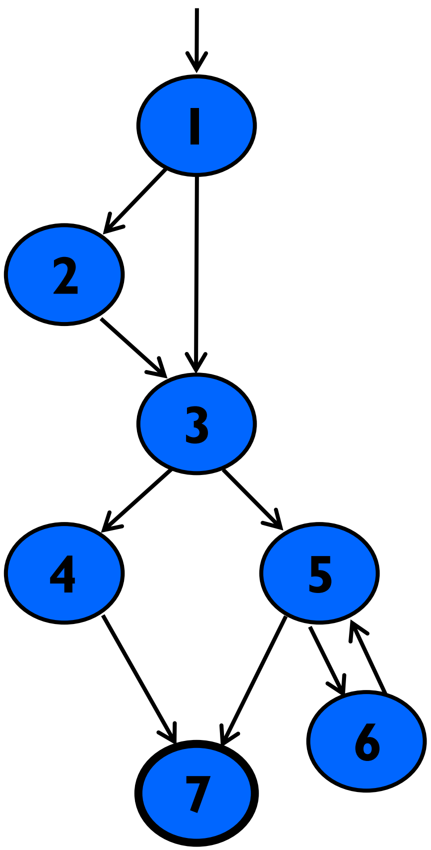
Example: Structural Coverage

List test requirements and their corresponding test paths for NC, EC, EPC, CPC.



Example: Structural Coverage

List test requirements and their corresponding test paths for NC, EC, EPC, CPC.



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]}

Complete Path Coverage

Test Paths: {[1, 2, 3, 4, 7], [1, 2, 3, 5, 7], [1, 2, 3, 5, 6, 5, 6], [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, CPC is not feasible.
- Specified Path Coverage (SPC) is also not satisfactory because the results are subjective and vary with the tester.

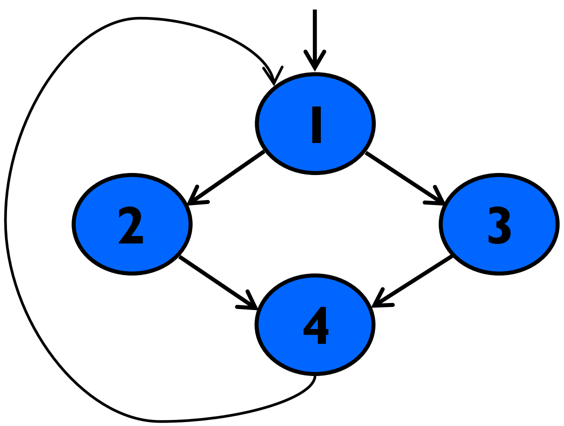
- Attempts to “deal with” loops:
  - 1970s : Execute cycles once ([6, 5, 6] in previous example, informal)
  - 1980s : Execute each loop, exactly once (formalized)
  - 1990s : Execute loops 0 times, once, more than once (informal description)
  - 2000s : Prime paths (Touring, Sidetrips, and Detours)

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
  - – 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 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
  - 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 = {[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]}

Prime Paths = {[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

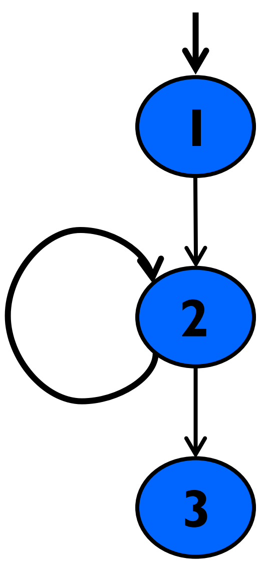
- A simple, elegant and finite criterion that requires loops to be executed as well as skipped

Prime Path Coverage (PPC): 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 …

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*]
- [*n*, *n*, *m*] is not prime
- Neither [*n*, *n*, *m*] nor [*m*, *n*, *n*] are simple paths (not prime)



EPC Requirements:

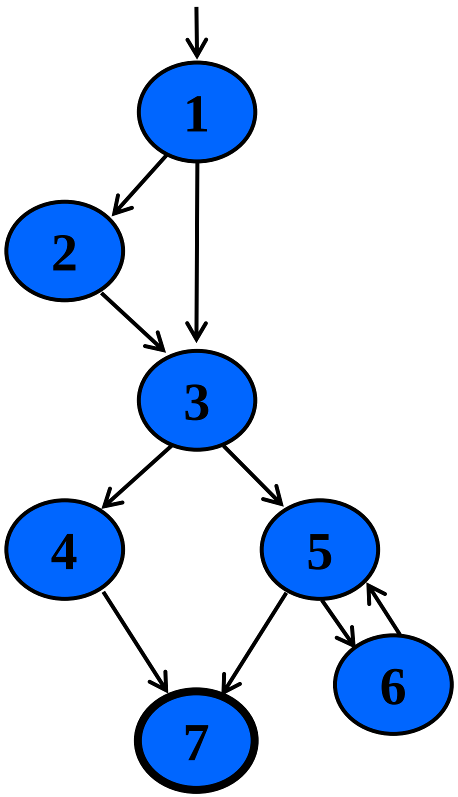
TR = { [1,2,3], [1,2,2], [2,2,3], [2,2,2] }

PPC Requirements:

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

Prime Path Example

- The following 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]

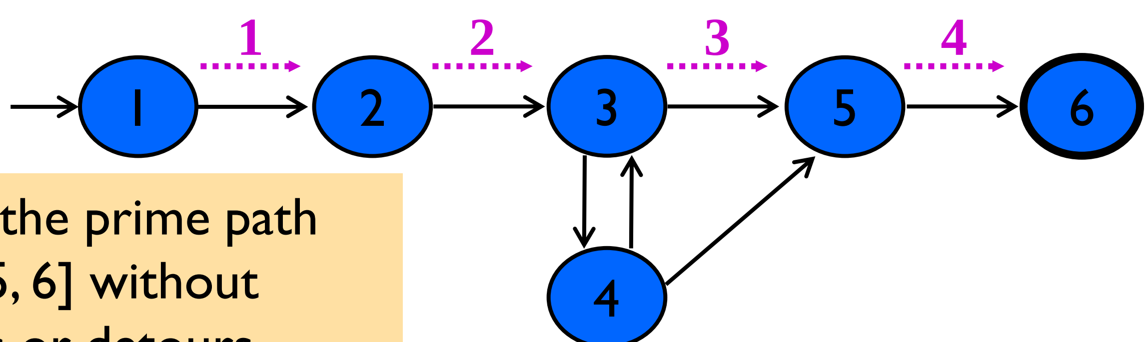
Execute loop 0 times

Execute loop once

Execute loop more than once

Touring, Sidetrips, and Detours

- Prime paths do not have internal loops … test paths <u>might</u>
- **Tour:** *A test path p tours subpath q if q is a subpath of p*
- If we are required to tour subpath q = [2, 3, 5], the strict definition of tour prohibits us from meeting the requirement with any path that contains 4, such as:
  - p = [1, 2, 3, 4, 3, 5, 6]
  - Indeed, we do not visit 2, 3, and 5 in exactly the same order.

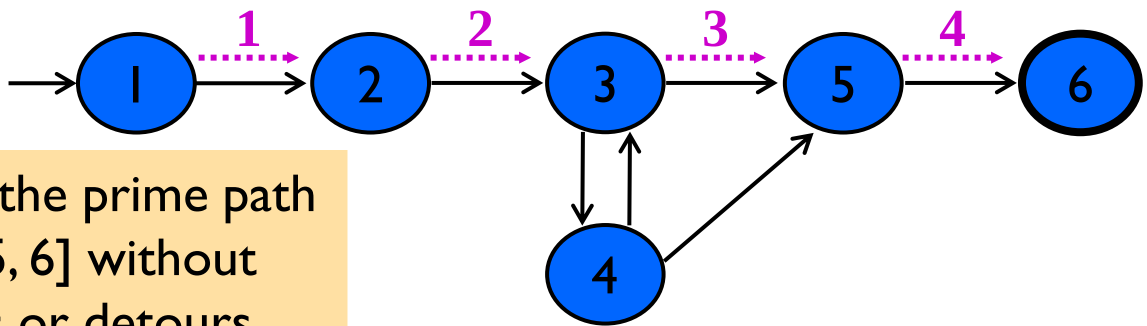


Touring the prime path [1, 2, 3, 5, 6] without <u>sidetrips or detours</u>

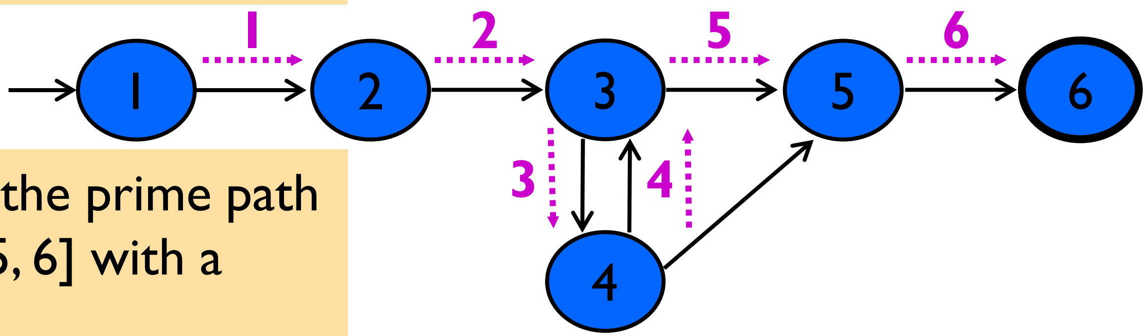
Touring, Sidetrips, and Detours

- Prime paths do not have internal loops … test paths <u>might</u>
- **Tour:** *A test path p tours subpath q if q is a subpath of p*
- We relax the **tour definition** in two ways:
- The first allows the tour to include "***sidetrips***," where we **can leave the path temporarily** from a node and then return to the same node.
- The second allows the tour to include more general "***detours***" where we can leave the path from a node and then return to the **next node** on the path (**skipping an**
- **edge**).

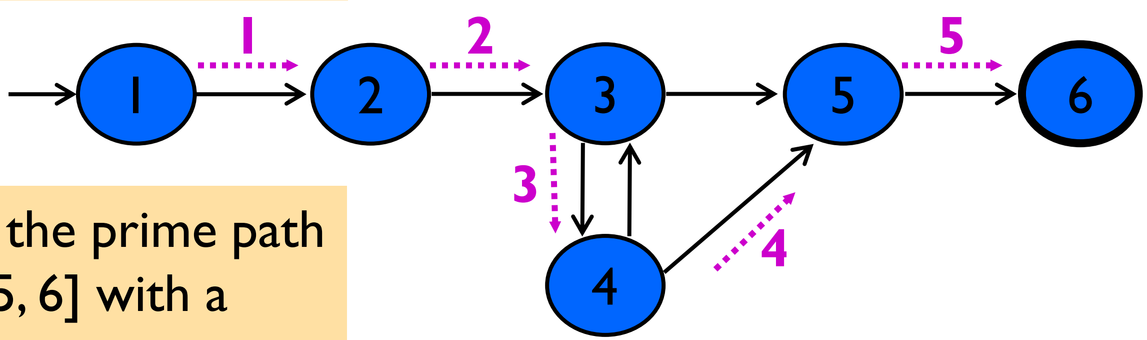
Sidetrips and Detours Example



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



Touring the prime path [1, 2, 3, 5, 6] with a <u>sidetrip</u>



Touring the prime path [1, 2, 3, 5, 6] with a <u>detour</u>

Touring, Sidetrips, and Detours

- Prime paths do not have 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*
  - The tour can include a sidetrip, as long as 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.*
  - The tour can include a detour from node n_i, as long as it comes back to the prime path at a successor of n_i.

Infeasible Test Requirements

- An infeasible test requirement cannot be satisfied.
  - Unreachable statement (dead code)
  - Subpath that can only be executed with a contradiction ($X > 0$ and $X < 0$)
- Most test criteria have some infeasible test requirements.
- It is 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

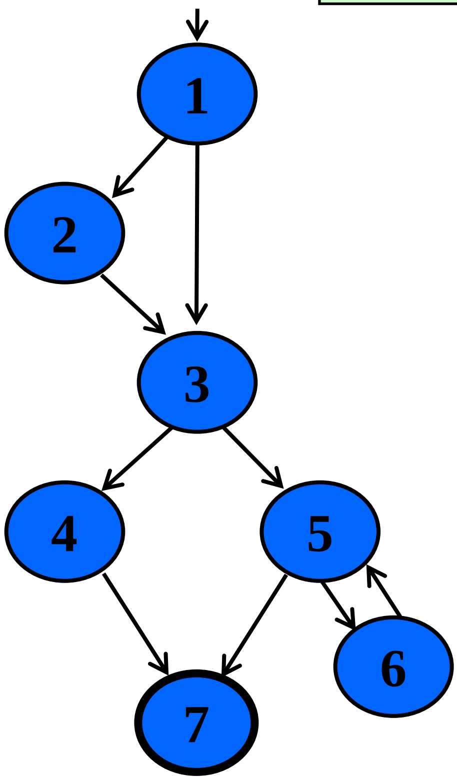
- **Satisfy as many test requirements as possible without sidetrips**
- **Allow sidetrips to try to satisfy remaining test requirements**

Finding Prime Test Paths

- It turns out to be relatively simple to find **all prime paths** in a graph, and test paths to tour the prime paths can be constructed **automatically**.
- The following websites contains a graph coverage web application tool that will compute prime paths (and other criteria) on general graphs.
  - https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage
- The **CodA** tool
  - https://m-zakeri.github.io/CodA/
  - To be completed by You!

Simple paths → **Prime paths** → **Test paths** → **Minimum cost test paths (MCTC)**

Simple & Prime Path Example

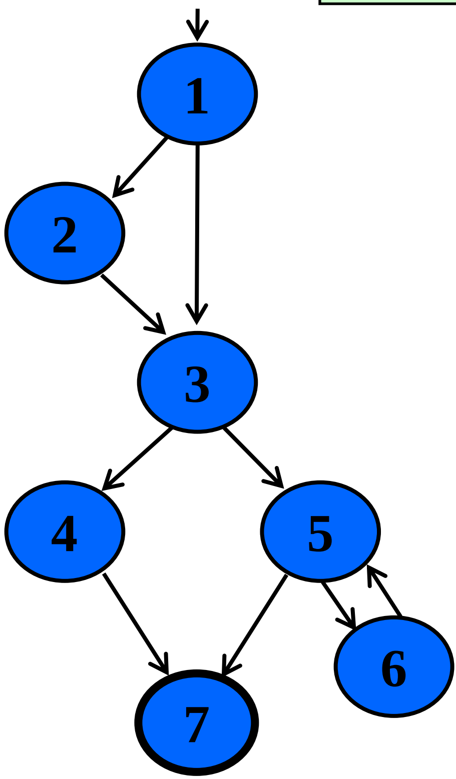


Simple paths

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

'!' means path terminates

Simple & Prime Path Example



Simple paths

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

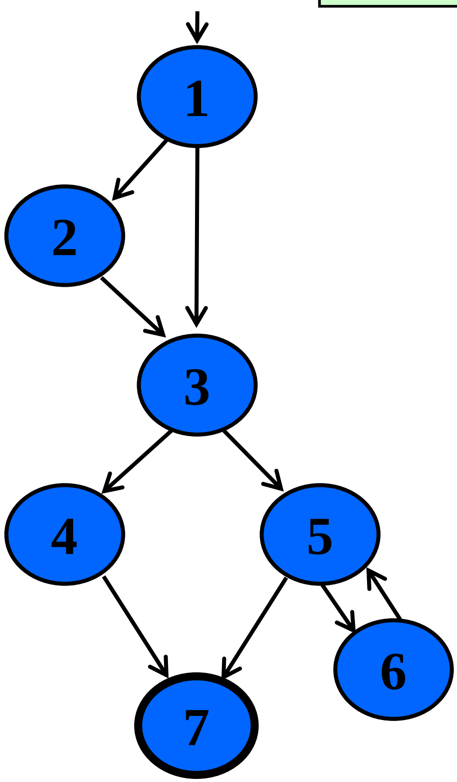
Len 1
[1, 2]
[1, 3]
[2, 3]
[3, 4]
[3, 5]
[4, 7] !
[5, 7] !
[5, 6]
[6, 5]

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] *

‘!’ means path … **es**

‘*’ means path cycles

Simple & Prime Path Example



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]

'!' means path
es

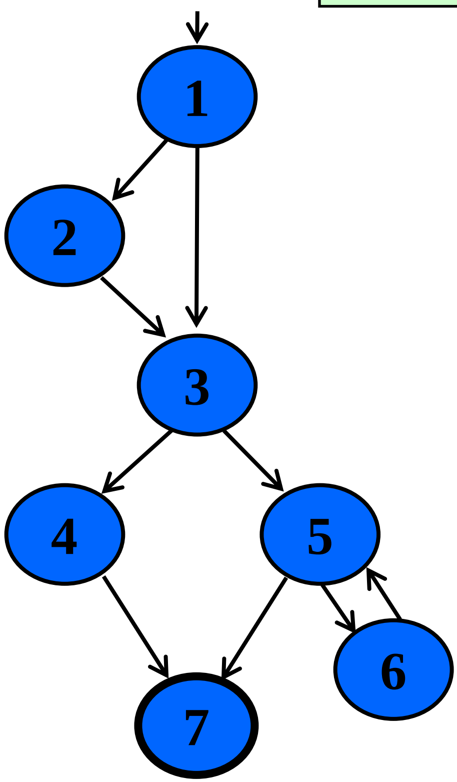
Len 2
[1, 2, 3]
[1, 3, 4]
[1, 3, 5]
[2, 3, 4]
[2, 3, 5]
[3, 4, 7] !
[3, 5, 7] !
[3, 5, 6] !
[5, 6, 5] *
[6, 5, 7] !
[6, 5, 6] *

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] !

ns path
les

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

Simple & Prime Path Example



Simple paths

'!' means path ... es

... s path ... les

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] !

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

Prime Paths

Finding Prime Test Paths Automatically

- This process is guaranteed to terminate because the length of the **longest possible prime path is the number of nodes.**
- Although graphs often have many simple paths they can usually be **toured** with far fewer **test paths**.
- Many possible algorithms can **find test paths to tour the prime paths**.
  - N. Li, F. Li and J. Offutt, "**Better Algorithms to Minimize the Cost of Test Paths**," *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, Montreal, QC, 2012, pp. 280-289, doi: 10.1109/ICST.2012.108.
  - Ebrahim Fazli, Mohsen Afsharchi, "**A time and space-efficient compositional method for prime and test paths generation**", *IEEE Access*, vol.7, pp.134399-134410, 2019.
  - Parampreet Kaur, Ashish Kr. Luhach, "**An approach to improve test path generation: Inclination towards automated model-based software design and testing**", *2016 5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*, pp.156-162, 2016.

Finding Prime Test Paths Automatically

- Develop your own algorithm:
  - We recommend starting with the longest prime paths and extending them to the beginning and end nodes in the graph.
  - Visit our **Domain Coverage** and **CodA** projects:
  - https://github.com/m-zakeri/DomainCoverage
    - *https://github.com/m-zakeri/DomainCoverage/tree/main/code/src/code_coverage*
  - https://github.com/m-zakeri/CodA/
- Test engineer can evaluate the **tradeoffs** between more but shorter test paths and fewer but longer test paths and choose the appropriate algorithm.

Round Trips

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

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

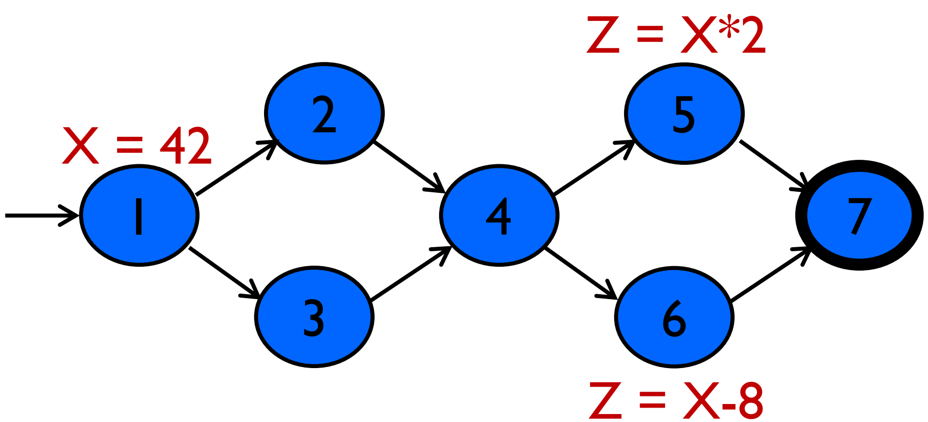
> **<u>Complete Round Trip Coverage (CRTC)</u>: TR contains all round-trip paths for each reachable node in G.**

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

Data Flow Criteria

<u>Goal</u>: Try to 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}

<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 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

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.

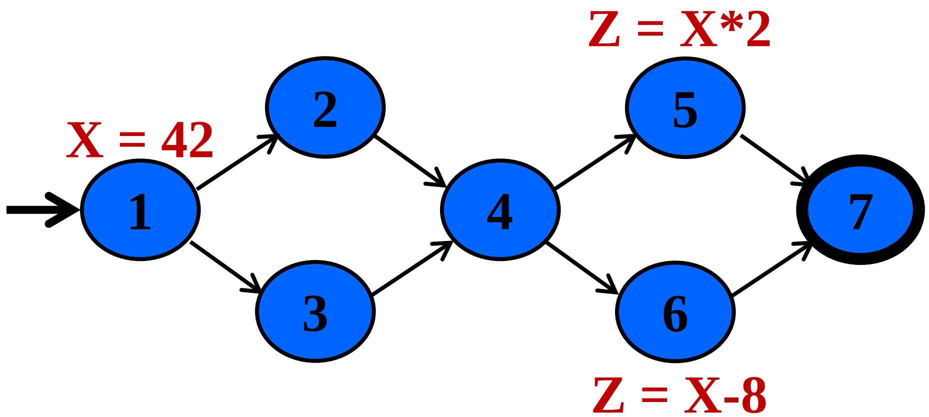
> **<u>All-uses coverage (AUC)</u>: For each set of du-paths to uses $S = du(n_i, n_j, v)$, TR contains at least one path d in S.**

- 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.**

Data Flow Testing Example

Write down the TRs and Test Paths for these criteria.

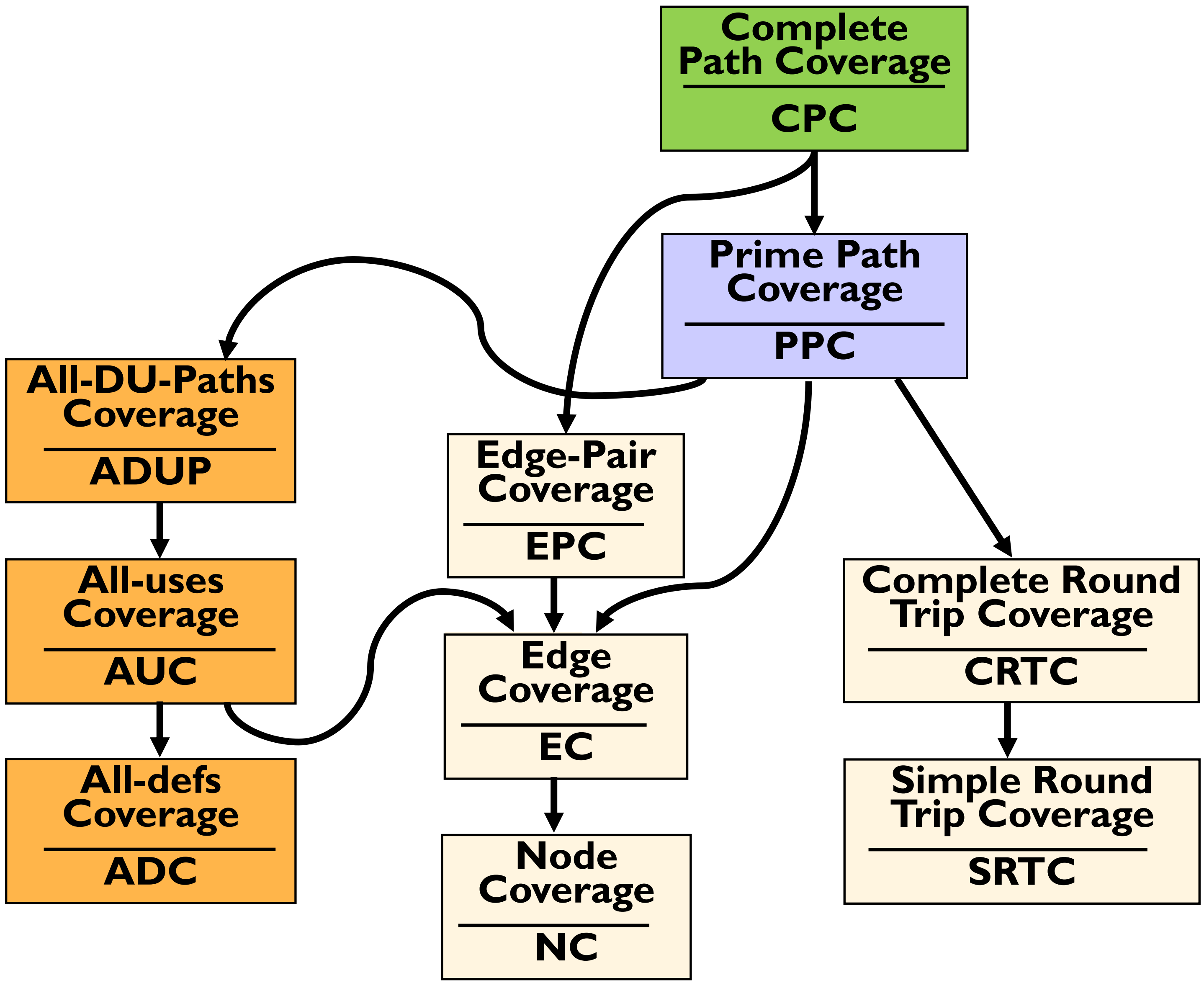


All-defs for *X*
[1, 2, 4, 5]

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 Coverage Criteria Subsumption



Summary 7.1-7.2

- 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
  - As discussed in the rest of chapter 7

Graph Coverage for Source Code (7.3)

https://github.com/m-zakeri/CodA/

https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage

Overview

- A common application of graph criteria is to program source
- **Graph:** Usually the control flow graph (CFG) of a function or method.
- **Node coverage:** Execute every statement (statement or line or basic block coverage)
- **Edge coverage:** Execute every branch (branch coverage)
- **Loops:** Looping structures such as for loops, while loops, etc.
- **Data flow coverage**: **Augment the CFG**
  – defs are statements that assign values to variables
  – uses are statements that use variables

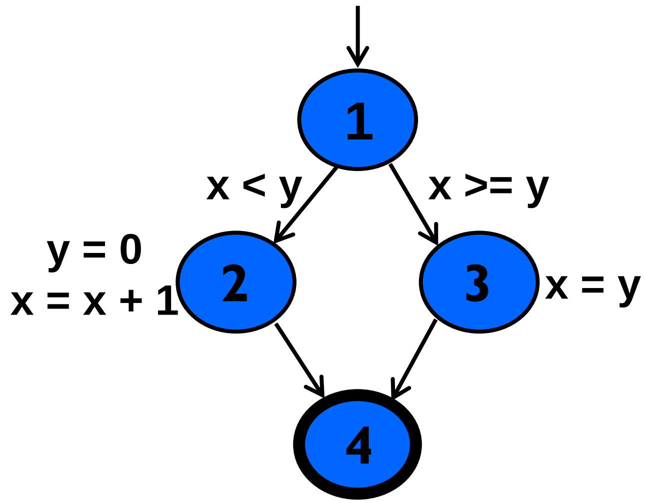
Control Flow Graphs

- A CFG models all executions of a method by describing control structures
- **Nodes:** Statements or sequences of statements (**basic blocks**)
- **Edges:** Transfers of control
- **Basic Block:** A sequence of statements such that if the first statement is executed, all statements will be (no branches)
- CFGs are sometimes annotated with extra information
  - branch predicates, defs, uses
- Rules for translating statements into graphs …

CFG: The *if* Statement

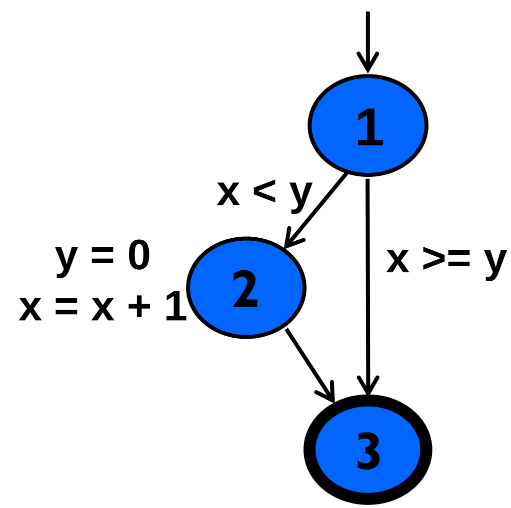
Draw the graph. Label the edges with the Java statements.

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
```



Draw the graph and label the edges.

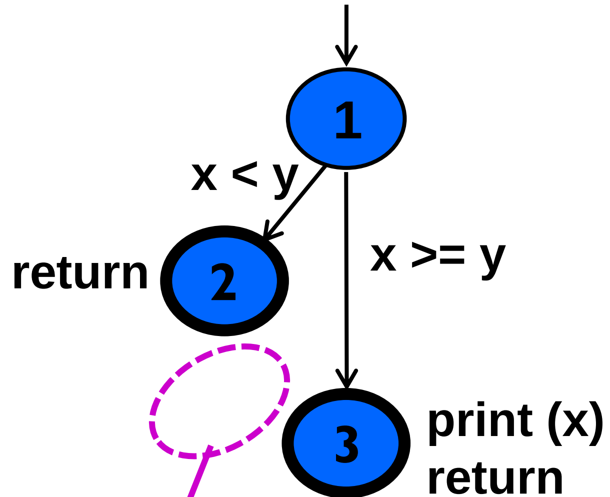
```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```



CFG: The *if-Return* Statement

Draw the graph and label the edges.

```
if (x < y)
{
   return;
}
print (x);
return;
```



No edge from node 2 to 3.
The return nodes must be distinct.

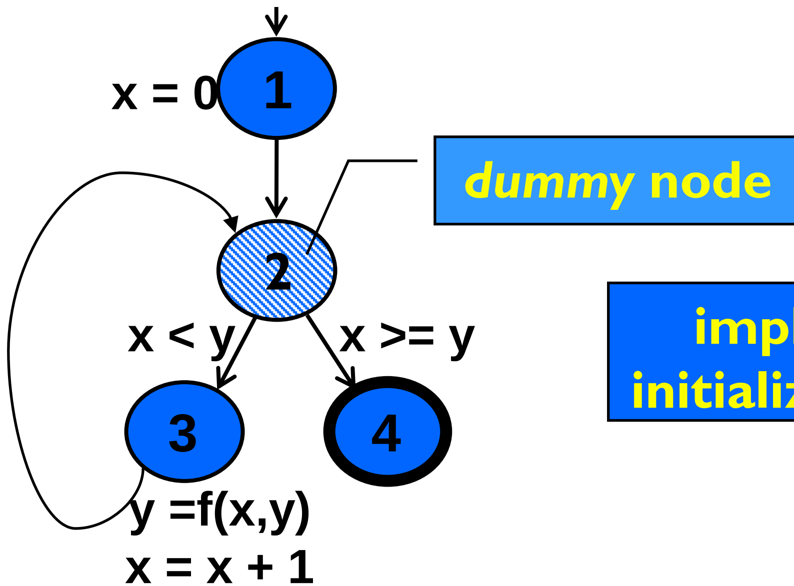
Loops

- Loops require "*extra*" nodes to be added
- Nodes that do not represent statements or basic blocks

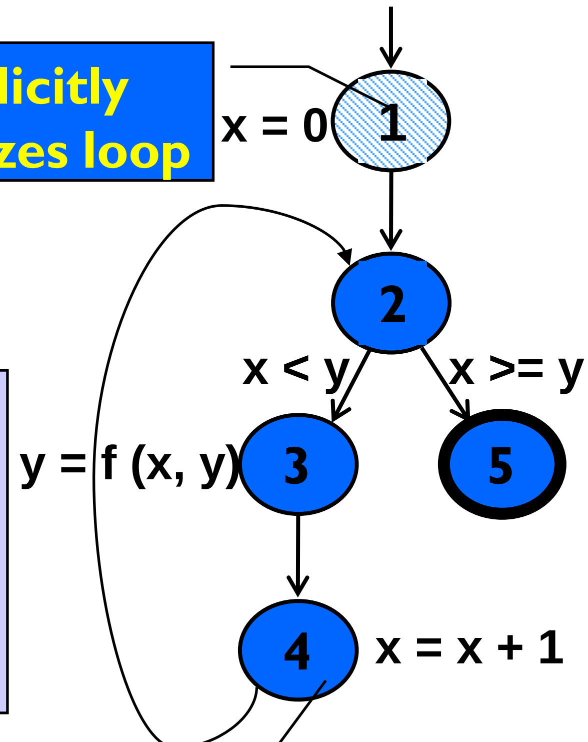
CFG : while and for Loops

Draw the graph and label the edges.

```
x = 0;
while (x < y)
{
   y = f (x, y);
   x = x + 1;
}
return (x);
```



implicitly initializes loop



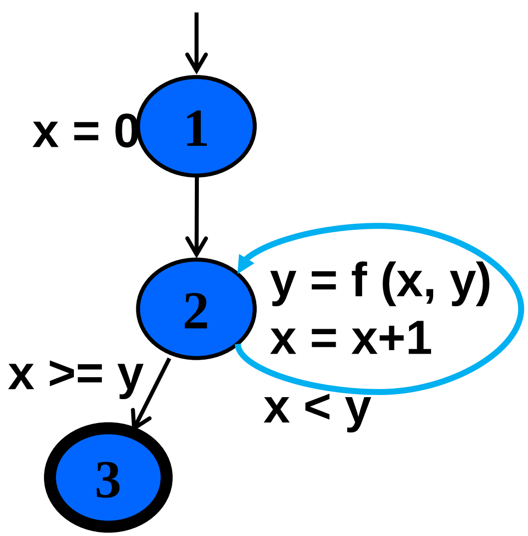
Draw the graph and label the edges.

```
for (x = 0; x < y; x++)
{
   y = f (x, y);
}
return (x);
```

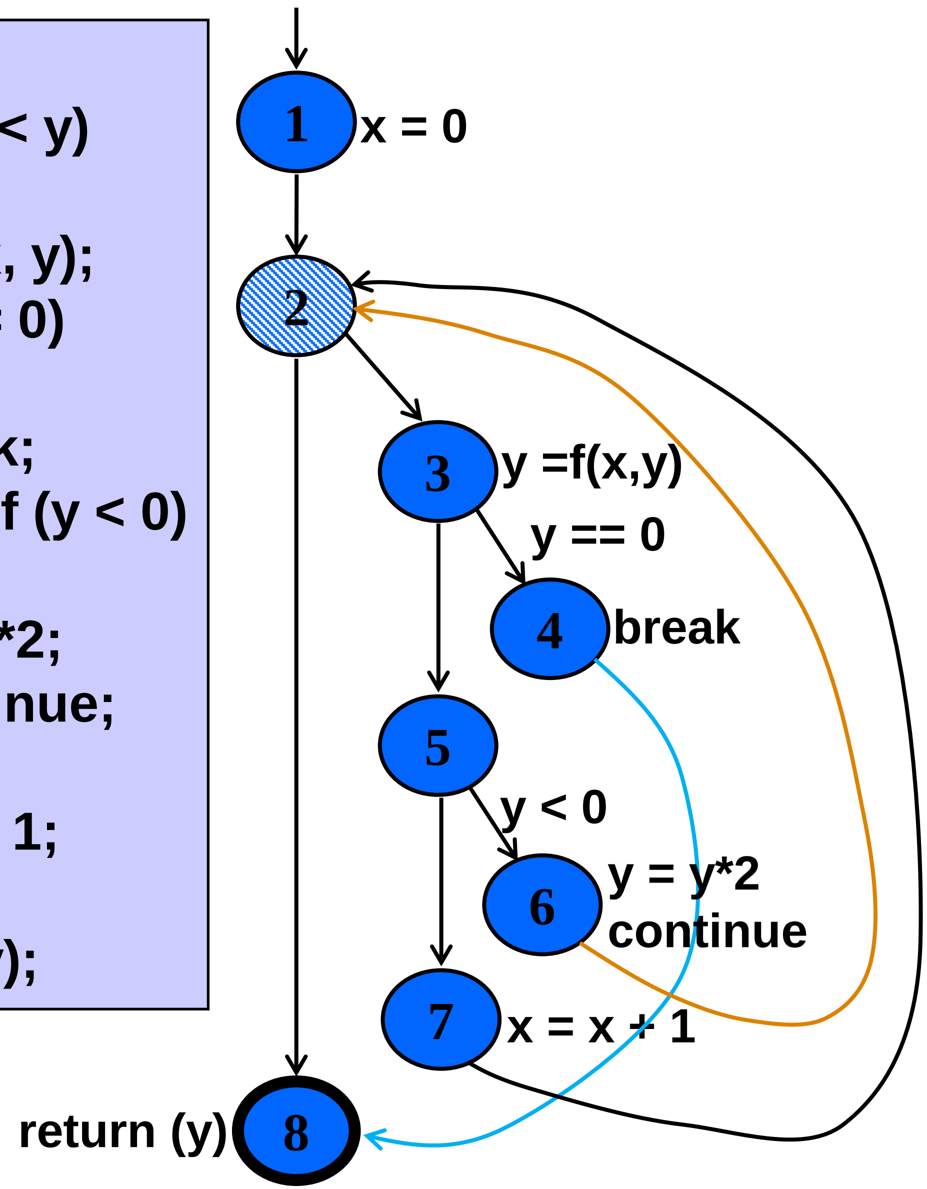
implicitly increments loop

CFG: do Loop, break, and continue

```
x = 0;
do
{
   y = f (x, y);
   x = x + 1;
} while (x < y);
return (y);
```



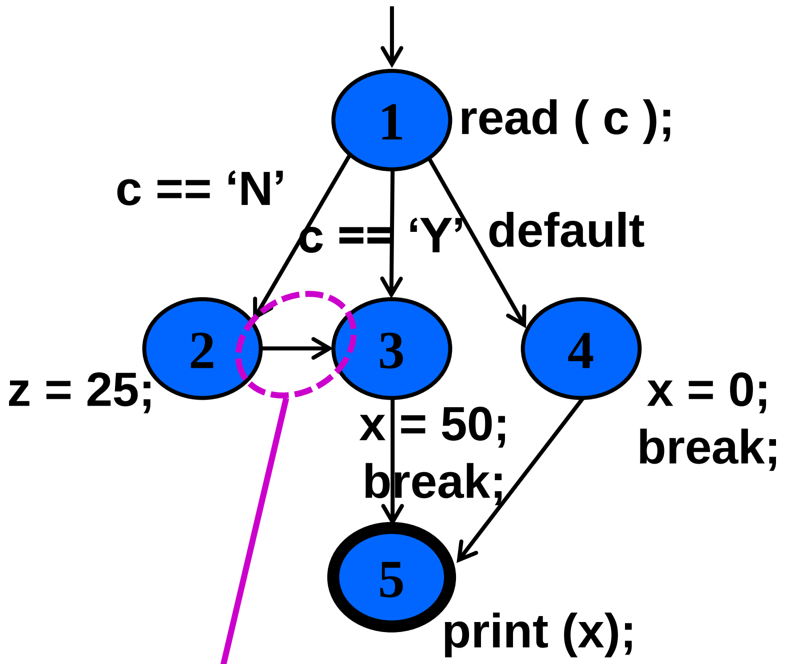

```
x = 0;
while (x < y)
{
   y = f (x, y);
   if (y == 0)
   {
      break;
   } else if (y < 0)
   {
      y = y*2;
      continue;
   }
   x = x + 1;
}
return (y);
```



CFG: The case (switch) Structure

Draw the graph and label the edges.

```
read ( c) ;
switch ( c )
{
   case 'N':
      z = 25;
   case 'Y':
      x = 50;
      break;
   default:
      x = 0;
      break;
}
print (x);
```

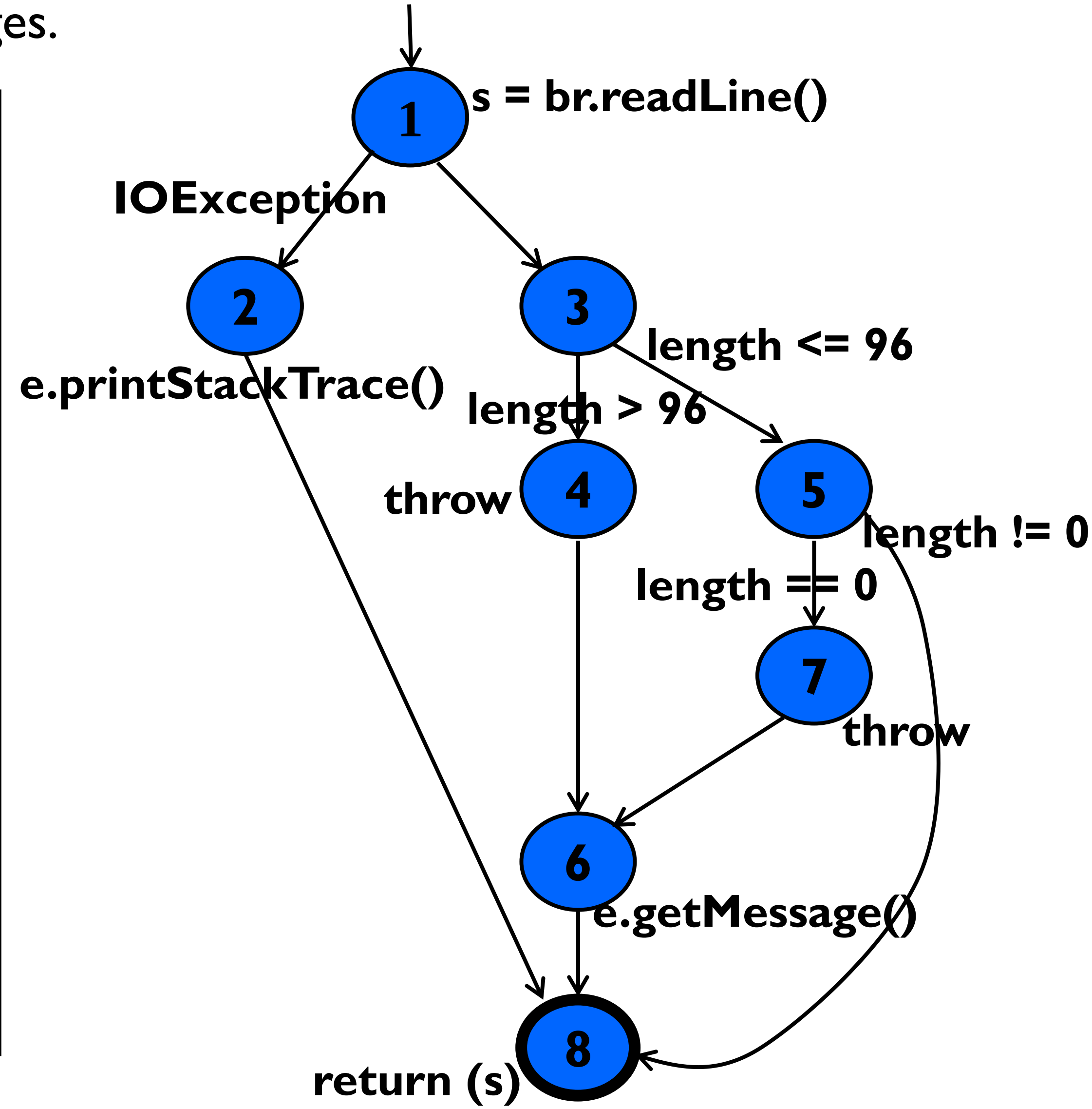


Cases without breaks fall through to the next case

CFG: Exceptions (try-catch)

Draw the graph and label the edges.

```
try
{
   s = br.readLine();
   if (s.length() > 96)
      throw new Exception
         ("too long");
   if (s.length() == 0)
      throw new Exception
         ("too short");
} (catch IOException e) {
   e.printStackTrace();
} (catch Exception e) {
   e.getMessage();
}
return (s);
```



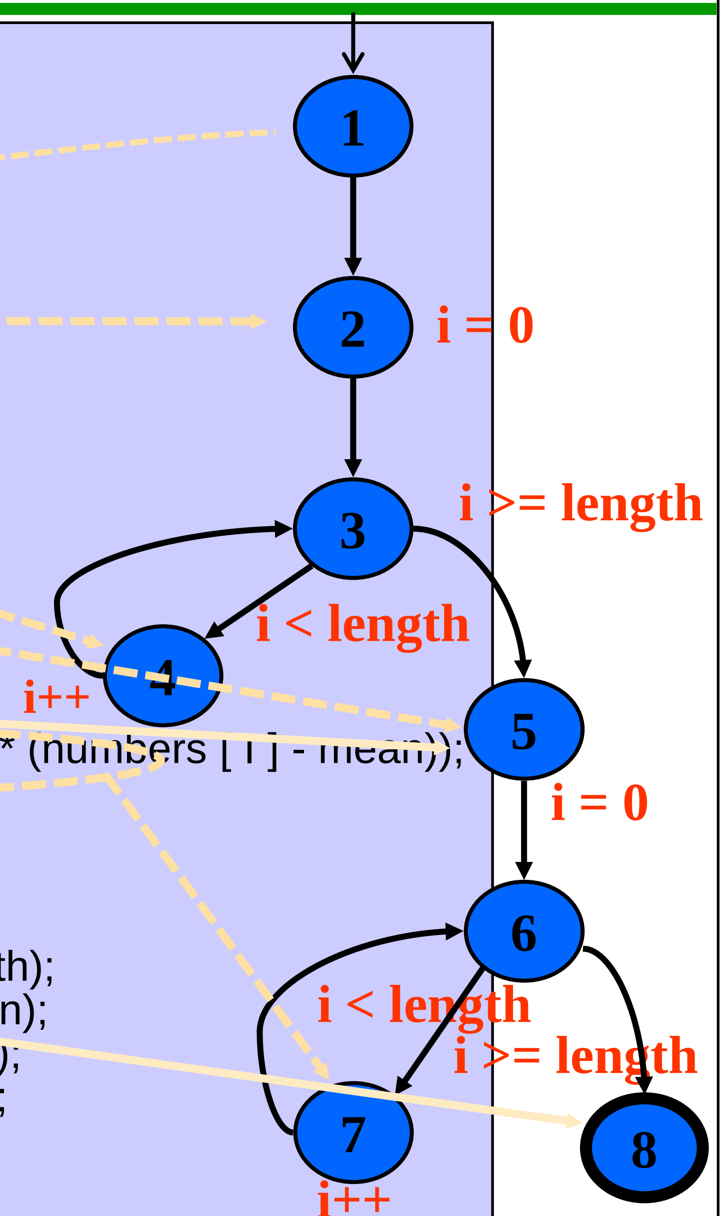
Example Control Flow – Stats

```
public static void computeStats (int [ ] numbers)
{
     int length = numbers.length;
     double med, var, sd, mean, sum, varsum;

     sum = 0;
     for (int i = 0; i < length; i++)
     {
          sum += numbers [ i ];
     }
     med   = numbers [ length / 2];
     mean = sum / (double) length;

     varsum = 0;
     for (int i = 0; i < length; i++)
     {
          varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
     }
     var = varsum / ( length - 1.0 );
     sd  = Math.sqrt ( var );

     System.out.println ("length:                    " + length);
     System.out.println ("mean:                      " + mean);
     System.out.println ("median:                    " + med);
     System.out.println ("variance:                  " + var);
     System.out.println ("standard deviation: " + sd);
}
```

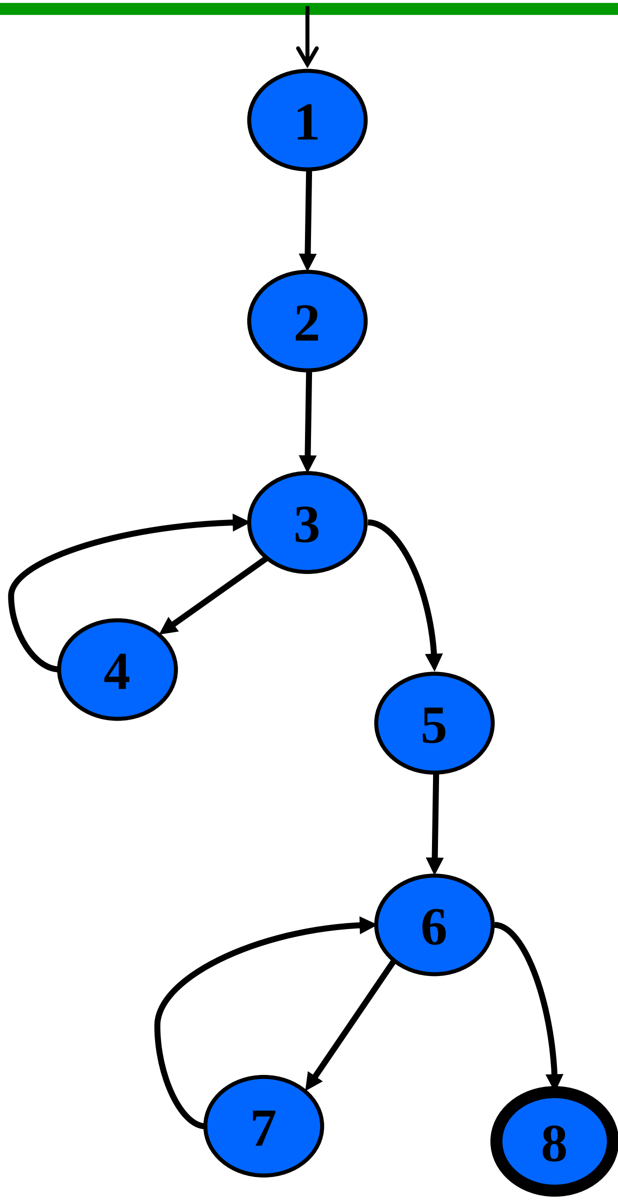
Draw the graph and label the edges.

Control Flow Graph for Stats

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2];
    mean = sum / (double) length;

    varsum = 0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ I ] - mean) * (numbers [ I ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                    " + length);
    System.out.println ("mean:                      " + mean);
    System.out.println ("median:                    " + med);
    System.out.println ("variance:                  " + var);
    System.out.println ("standard deviation: " + sd);
}
```

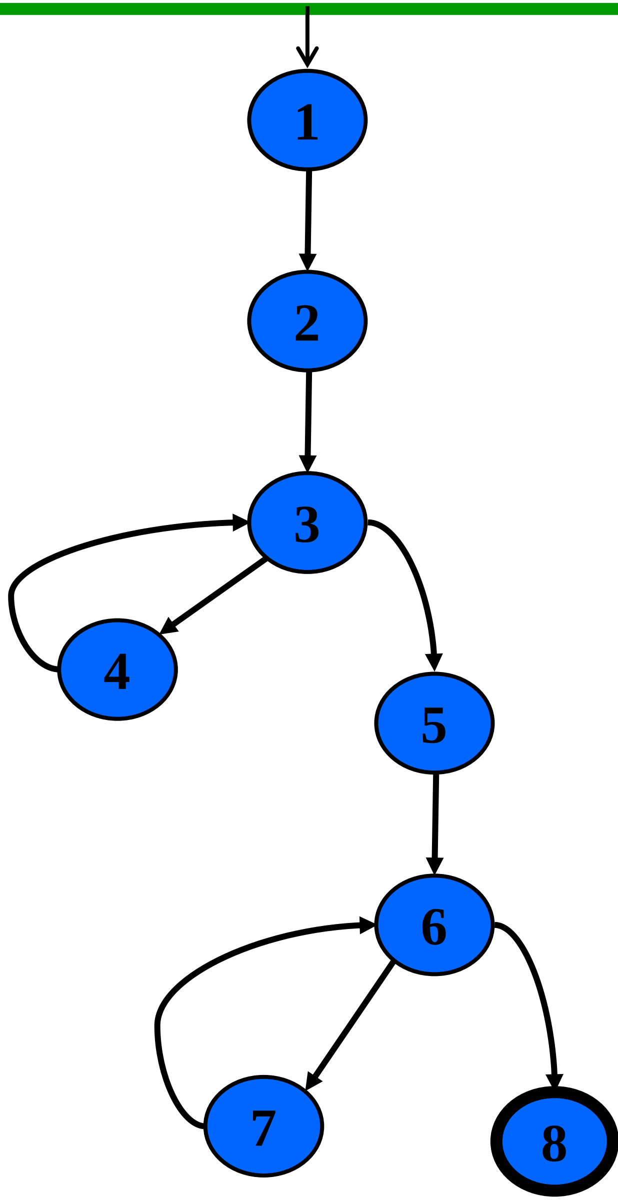


Control Flow TRs and Test Paths—EPC



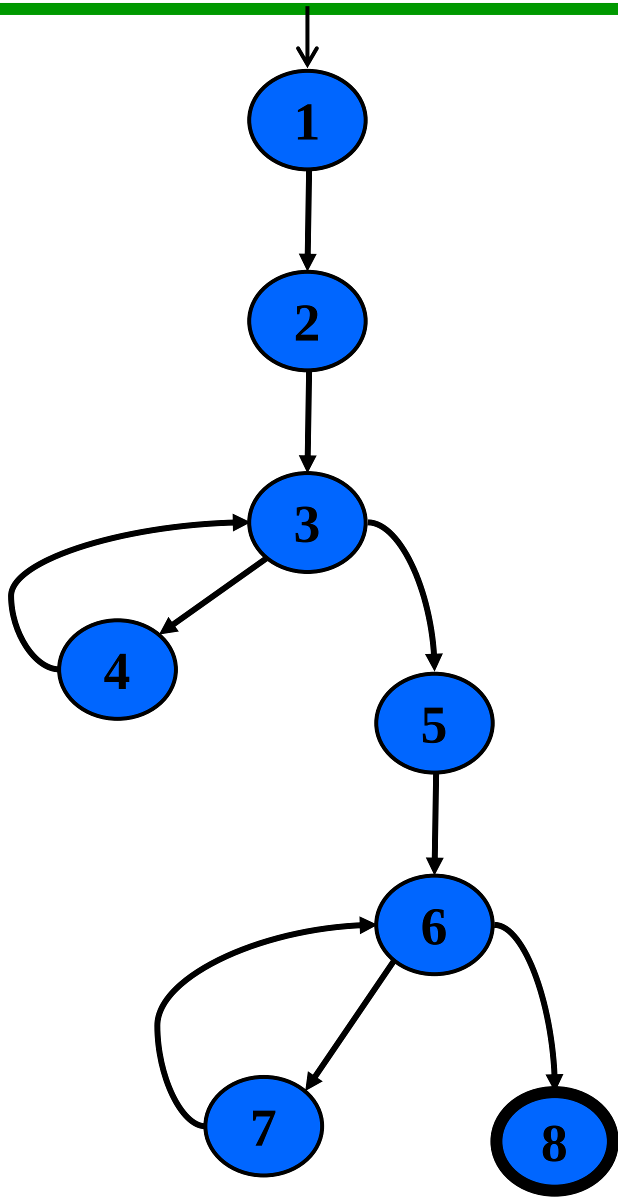
Edge-Pair Coverage	
TR	**Test Paths**
Write down TRs for EPC.	*Write down test paths that tour all edge pairs.*

Control Flow TRs and Test Paths—EPC



Edge-Pair Coverage	
TR	**Test Paths**
A. [1, 2, 3] B. [2, 3, 4] C. [2, 3, 5] D. [3, 4, 3] E. [3, 5, 6] F. [4, 3, 5] G. [5, 6, 7] H. [5, 6, 8] I. [6, 7, 6] J. [7, 6, 8] K. [4, 3, 4] L. [7, 6, 7]	i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8] ii. [1, 2, 3, 5, 6, 8] iii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]

Control Flow TRs and Test Paths—EPC



Edge-Pair Coverage

TR

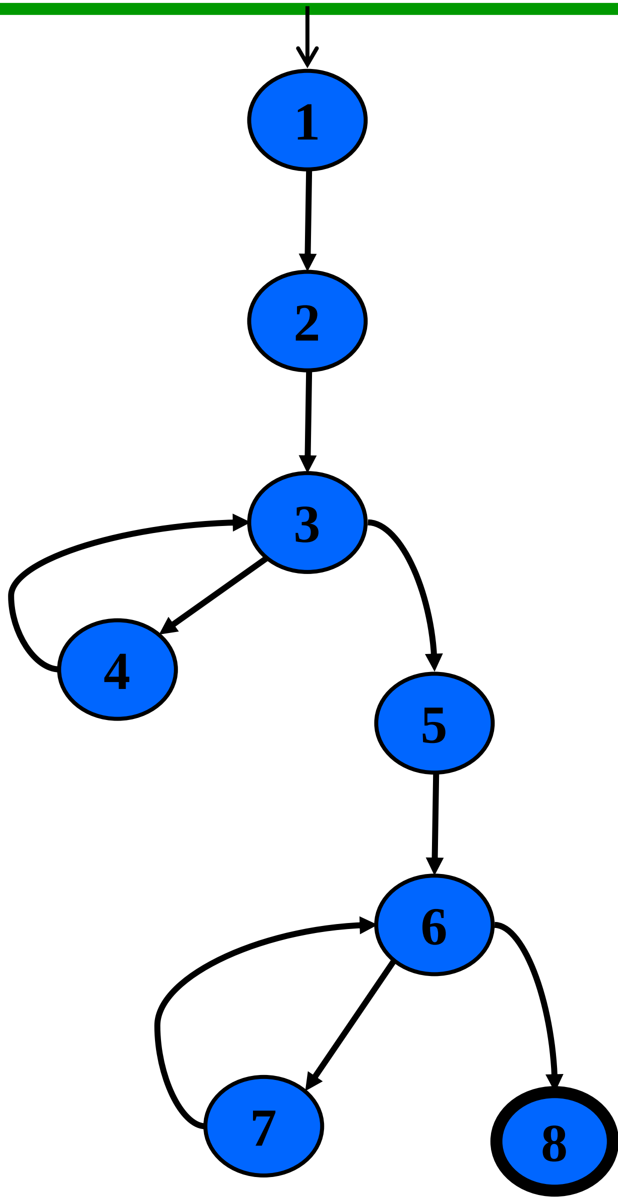
A. [1, 2, 3]
B. [2, 3, 4]
C. [2, 3, 5]
D. [3, 4, 3]
E. [3, 5, 6]
F. [4, 3, 5]
G. [5, 6, 7]
H. [5, 6, 8]
I. [6, 7, 6]
J. [7, 6, 8]
K. [4, 3, 4]
L. [7, 6, 7]

Test Paths

i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
ii. [1, 2, 3, 5, 6, 8]
iii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]

TP	TRs toured	*sidetrips*
i	A, B, D, E, F, G, I, J	C, H
ii	A, C, E, H	
iii	A, B, D, E, F, G, I, J, K, L	C, H

Control Flow TRs and Test Paths—EPC



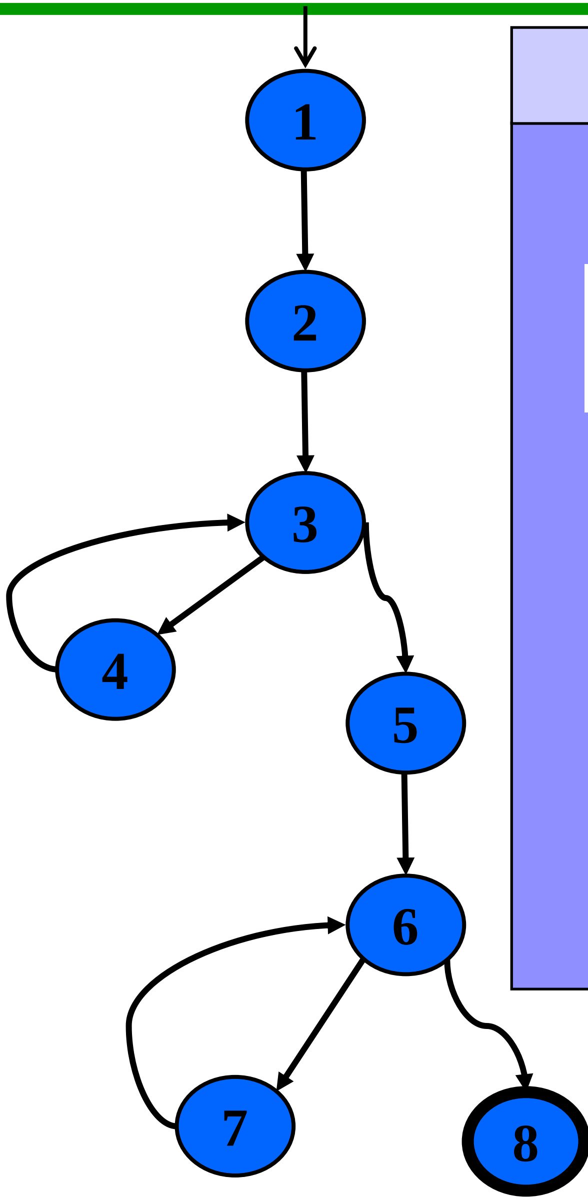
Edge-Pair Coverage

TR	Test Paths
A. [1, 2, 3] B. [2, 3, 4] C. [2, 3, 5] D. [3, 4, 3] E. [3, 5, 6] F. [4, 3, 5] G. [5, 6, 7] H. [5, 6, 8] I. [6, 7, 6] J. [7, 6, 8] K. [4, 3, 4] L. [7, 6, 7]	i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8] ii. [1, 2, 3, 5, 6, 8] iii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]

TP	TRs toured	*sidetrips*
~~i~~	~~A, B, D, E, F, G, I, J~~	~~C, H~~
ii	A, C, E, H	
iii	A, B, D, E, F, G, I, J, K, L	C, H

TP iii makes TP i redundant. A *minimal* set of TPs is cheaper.

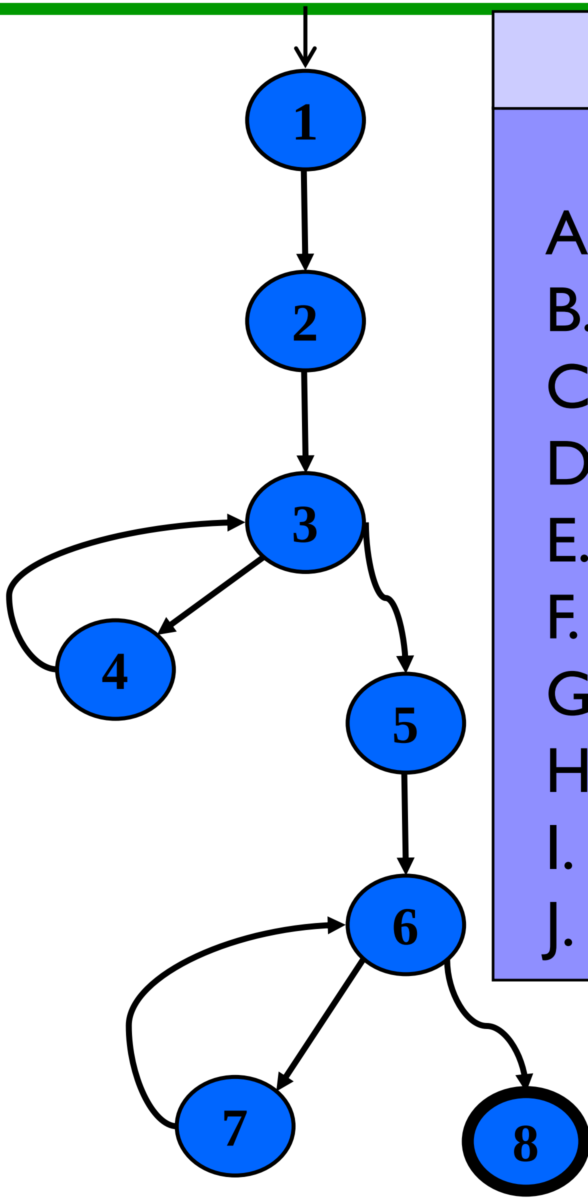
Control Flow TRs and Test Paths—PPC



Prime Path Coverage

TR	Test Paths
Write down TRs for PPC.	*Write down test paths that tour all prime paths.*

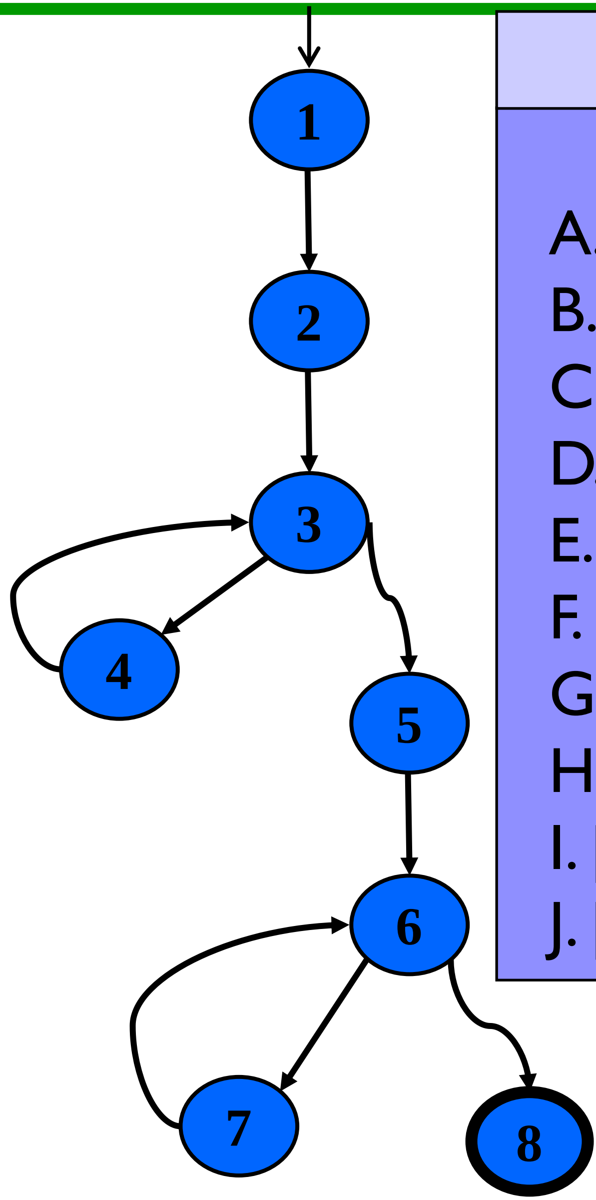
Control Flow TRs and Test Paths—PPC



Prime Path Coverage

TR	Test Paths
A. [3, 4, 3]	i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [4, 3, 4]	ii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]
C. [7, 6, 7]	iii. [1, 2, 3, 4, 3, 5, 6, 8]
D. [7, 6, 8]	iv. [1, 2, 3, 5, 6, 7, 6, 8]
E. [6, 7, 6]	v. [1, 2, 3, 5, 6, 8]
F. [1, 2, 3, 4]	
G. [4, 3, 5, 6, 7]	
H. [4, 3, 5, 6, 8]	
I. [1, 2, 3, 5, 6, 7]	
J. [1, 2, 3, 5, 6, 8]	

Control Flow TRs and Test Paths—PPC

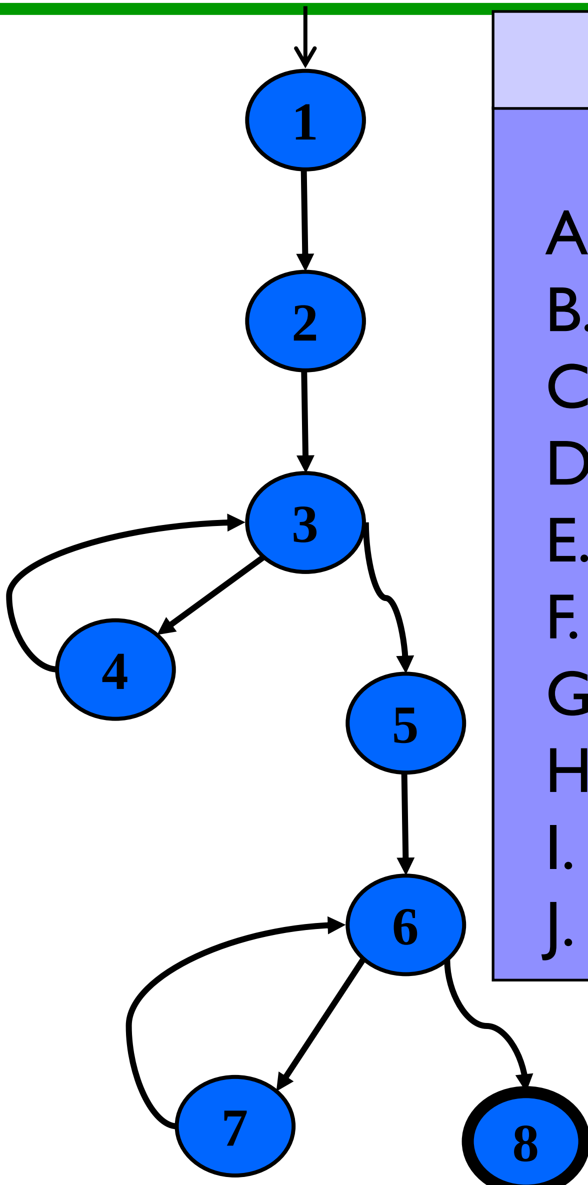


Prime Path Coverage

TR	Test Paths
A. [3, 4, 3]	i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [4, 3, 4]	ii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]
C. [7, 6, 7]	iii. [1, 2, 3, 4, 3, 5, 6, 8]
D. [7, 6, 8]	iv. [1, 2, 3, 5, 6, 7, 6, 8]
E. [6, 7, 6]	v. [1, 2, 3, 5, 6, 8]
F. [1, 2, 3, 4]	
G. [4, 3, 5, 6, 7]	
H. [4, 3, 5, 6, 8]	
I. [1, 2, 3, 5, 6, 7]	
J. [1, 2, 3, 5, 6, 8]	

TP	TRs toured	*sidetrips*
i	A, D, E, F, G	H, I, J
ii	A, B, C, D, E, F, G,	H, I, J
iii	A, F, H	J
iv	D, E, F, I	J
v	J	

Control Flow TRs and Test Paths—PPC



Prime Path Coverage

TR	Test Paths
A. [3, 4, 3]	i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [4, 3, 4]	ii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]
C. [7, 6, 7]	iii. [1, 2, 3, 4, 3, 5, 6, 8]
D. [7, 6, 8]	iv. [1, 2, 3, 5, 6, 7, 6, 8]
E. [6, 7, 6]	v. [1, 2, 3, 5, 6, 8]
F. [1, 2, 3, 4]	
G. [4, 3, 5, 6, 7]	
H. [4, 3, 5, 6, 8]	
I. [1, 2, 3, 5, 6, 7]	
J. [1, 2, 3, 5, 6, 8]	

TP	TRs toured	*sidetrips*
~~i~~	~~A, D, E, F, G~~	~~H, I, J~~
ii	A, B, C, D, E, F, G,	H, I, J
iii	A, F, H	J
iv	D, E, F, I	J
v	J	

TP ii makes TP i redundant.

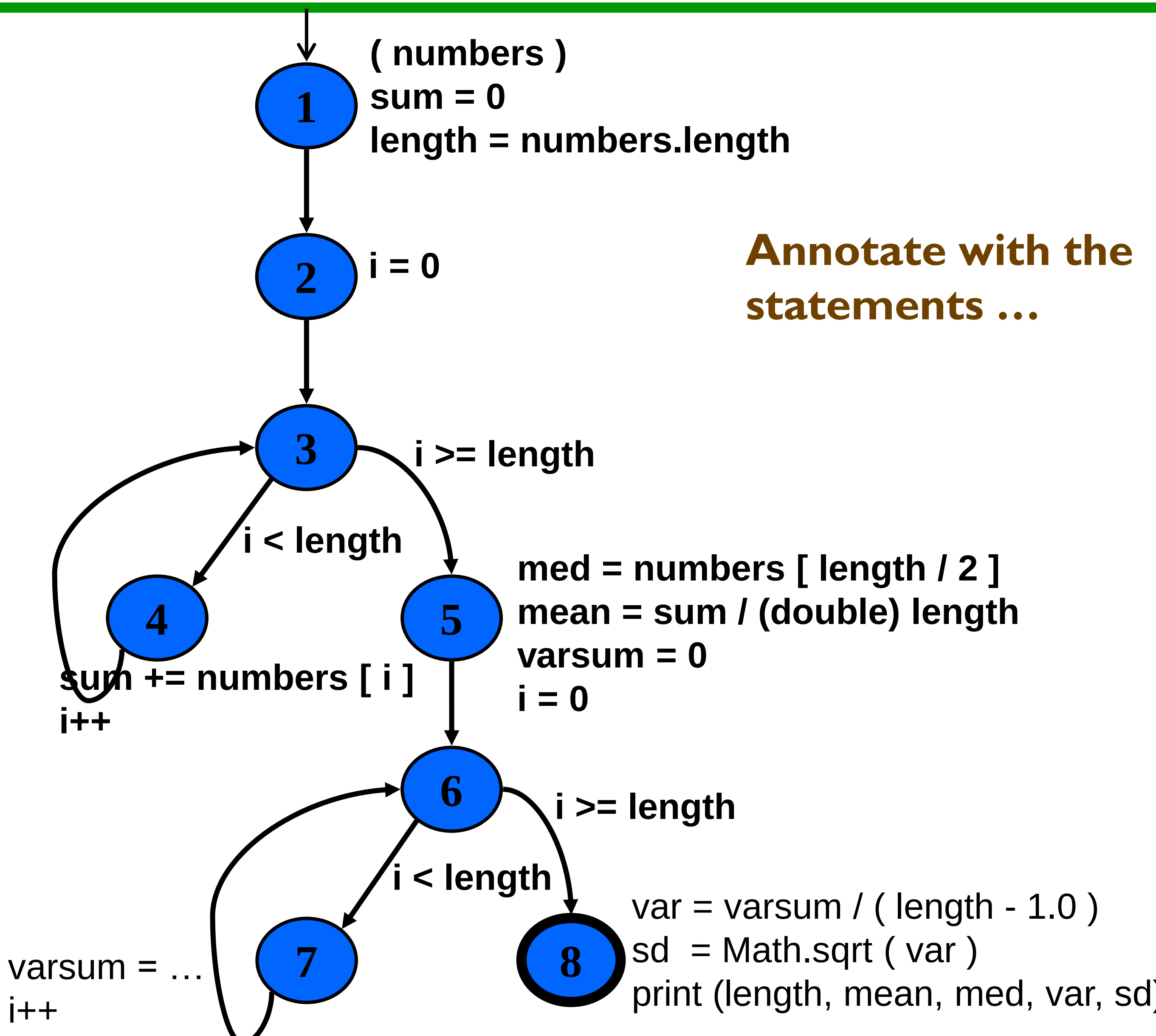
Data Flow Coverage for Source

- **def:** a location where a value is stored into memory
  - x appears on the left side of an assignment (x = 44;)
  - x is an actual parameter in a call and the method changes its value
  - x is a formal parameter of a method (implicit def when method starts)
  - x is an input to a program
- **use:** a location where variable's value is accessed
  - x appears on the right side of an assignment
  - x appears in a conditional test
  - x is an actual parameter to a method
  - x is an output of the program
  - x is an output of a method in a return statement
- If a def and a use appear on the same node, then it is only a DU-pair if the def occurs after the use and the node is in a loop

Example Data Flow – Stats

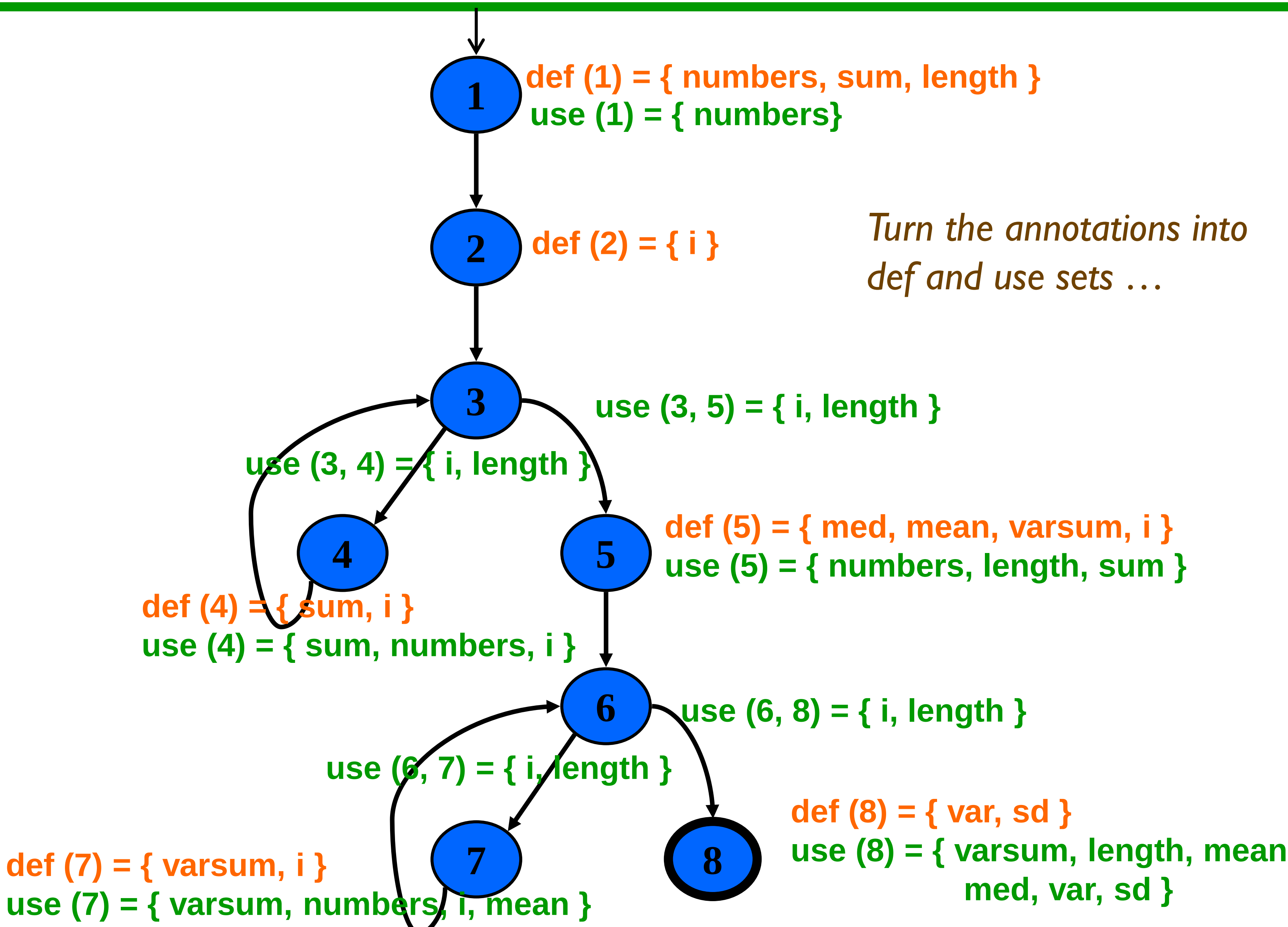
```
public static void computeStats (int [ ] numbers)
{
     int length = numbers.length;
     double med, var, sd, mean, sum, varsum;

     sum = 0.0;
     for (int i = 0; i < length; i++)
     {
          sum += numbers [ i ];
     }
     med   = numbers [ length / 2 ];
     mean = sum / (double) length;

     varsum = 0.0;
     for (int i = 0; i < length; i++)
     {
          varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
     }
     var = varsum / ( length - 1 );
     sd  = Math.sqrt ( var );

     System.out.println ("length:                    " + length);
     System.out.println ("mean:                      " + mean);
     System.out.println ("median:                    " + med);
     System.out.println ("variance:                  " + var);
     System.out.println ("standard deviation: " + sd);
}
```

Control Flow Graph for Stats



Annotate with the statements …

CFG for Stats – With Defs & Uses



Turn the annotations into def and use sets …

Defs and Uses Tables for Stats

Node	Def	Use
1	{ numbers, sum, length }	{ numbers }
2	{ i }	
3		
4	{ sum, i }	{ numbers, i, sum }
5	{ med, mean, varsum, i }	{ numbers, length, sum }
6		
7	{ varsum, i }	{ varsum, numbers, i, mean }
8	{ var, sd }	{ varsum, length, var, mean, med, var, sd }

Edge	Use
(1, 2)	
(2, 3)	
(3, 4)	{ i, length }
(4, 3)	
(3, 5)	{ i, length }
(5, 6)	
(6, 7)	{ i, length }
(7, 6)	
(6, 8)	{ i, length }

DU Pairs for Stats

variable	DU Pairs
numbers	(1, 4) (1, 5) (1, 7)
length	(1, 5) (1, 8) (1, (3,4)) (1, (3,5)) (1, (6,7)) (1, (6,8))
med	(5, 8)
var	(8, 8)
sd	(8, 8)
mean	(5, 7) (5, 8)
sum	(1, 4) (1, 5) (4, 4) (4, 5)
varsum	(5, 7) (5, 8) (7, 7) (7, 8)
i	(2, 4) (2, (3,4)) (2, (3,5)) (2, 7) (2, (6,7)) (2, (6,8)) (4, 4) (4, (3,4)) (4, (3,5)) (4, 7) (4, (6,7)) (4, (6,8)) (5, 7) (5, (6,7)) (5, (6,8)) (7, 7) (7, (6,7)) (7, (6,8))

DU Pairs for Stats

variable	DU Pairs
numbers	(1, 4) (1, 5) (1, 7)
length	(1, 5) (1, 8) (1, (3,4)) (1, (3,5)) (1, (6,7)) (1, (6,8))
med	(5, 8)
var	(8, 8)
sd	(8, 8)
mean	(5, 7) (5, 8)
sum	(1, 4) (1, 5) (4, 4) (4, 5)
varsum	(5, 7) (5, 8) (7, 7) (7, 8)
i	(2, 4) (2, (3,4)) (2, (3,5)) ~~(2, 7) (2, (6,7)) (2, (6,8))~~ (4, 4) (4, (3,4)) (4, (3,5)) ~~(4, 7) (4, (6,7)) (4, (6,8))~~ (5, 7) (5, (6,7)) (5, (6,8)) (7, 7) (7, (6,7)) (7, (6,8))

defs come before uses, do not count as DU pairs

defs after use in loop, these are valid DU pairs

No def-clear path … different scope for i

No path through graph from nodes 5 and 7 to 4 or 3

DU Paths for Stats

variable	DU Pairs	DU Paths
numbers	(1, 4) (1, 5) (1, 7)	[1, 2, 3, 4] [1, 2, 3, 5] [1, 2, 3, 5, 6, 7]
length	(1, 5) (1, 8) (1, (3,4)) (1, (3,5)) (1, (6,7)) (1, (6,8))	[1, 2, 3, 5] [1, 2, 3, 5, 6, 8] [1, 2, 3, 4] [1, 2, 3, 5] [1, 2, 3, 5, 6, 7] [1, 2, 3, 5, 6, 8]
med	(5, 8)	[5, 6, 8]
var	(8, 8)	*No path needed*
sd	(8, 8)	*No path needed*
sum	(1, 4) (1, 5) (4, 4) (4, 5)	[1, 2, 3, 4] [1, 2, 3, 5] [4, 3, 4] [4, 3, 5]

variable	DU Pairs	DU Paths
mean	(5, 7) (5, 8)	[5, 6, 7] [5, 6, 8]
varsum	(5, 7) (5, 8) (7, 7) (7, 8)	[5, 6, 7] [5, 6, 8] [7, 6, 7] [7, 6, 8]
i	(2, 4) (2, (3,4)) (2, (3,5)) (4, 4) (4, (3,4)) (4, (3,5)) (5, 7) (5, (6,7)) (5, (6,8)) (7, 7) (7, (6,7)) (7, (6,8))	[2, 3, 4] [2, 3, 4] [2, 3, 5] [4, 3, 4] [4, 3, 4] [4, 3, 5] [5, 6, 7] [5, 6, 7] [5, 6, 8] [7, 6, 7] [7, 6, 7] [7, 6, 8]

DU Paths for Stats—No Duplicates

There are 38 DU paths for Stats, but only 12 unique

[1, 2, 3, 4]	[4, 3, 4]
[1, 2, 3, 5]	[4, 3, 5]
[1, 2, 3, 5, 6, 7]	[5, 6, 7]
[1, 2, 3, 5, 6, 8]	[5, 6, 8]
[2, 3, 4]	[7, 6, 7]
[2, 3, 5]	[7, 6, 8]

4 expect a loop not to be "entered"

6 require at least one iteration of a loop

2 require at least <u>two</u> iterations of a loop

Test Cases and Test Paths

Test Case: numbers = *[44]*; length = 1

Test Path : [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]

<u>Additional DU Paths covered (no sidetrips):</u>

{[1, 2, 3, 4],[2, 3, 4],[4, 3, 5],[5, 6, 7],[7, 6, 8]}

The five stars that require at least one iteration of a loop

Test Case: numbers = *[2, 10, 15]*; length = 3

Test Path: [1, 2, 3, 4, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 7, 6, 8]

<u>DU Paths covered (no sidetrips):</u>

{[4, 3, 4], [7, 6, 7]}

The two stars that require at least two iterations of a loop.

Other DU paths require arrays with length 0 to skip loops

But the method fails with index out of bounds exception…

med = numbers [length / 2];

A fault was found

Fault in Stats

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0.0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2 ]; // faulty line
    mean = sum / (double) length;

    varsum = 0.0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                    " + length);
    System.out.println ("mean:                      " + mean);
    System.out.println ("median:                    " + med);
    System.out.println ("variance:                  " + var);
    System.out.println ("standard deviation: " + sd);
}
```

Tools Computing Graph Coverage

- LCov (C/C++)
  - LCOV is an extension of GCOV, a GNU tool which provides information about what parts of a program are actually executed (i.e. "covered") while running a particular test case.
  - https://github.com/linux-test-project/lcov
- OpenClover (Java and Groovy)
  - https://openclover.org/
- Jcov (Java)
  - https://github.com/openjdk/jcov
- *vstest.console.exe (.NET)*
  - https://learn.microsoft.com/en-us/visualstudio/test/vstest-console-options?view=vs-2022

Program instrumentation

- **Instrumentation** refers to the measure of a product's performance, in order to diagnose errors and to write trace information.
  - Kind of program profiling
  - Two types: **source** instrumentation and **binary** instrumentation.
- Instrumentation is limited by **execution coverage**.
  - If the program never reaches a particular point of execution, then instrumentation at that point collects no data.
- Source code insertion (SCI) technology uses instrumentation techniques that automatically add specific code to the source files under analysis.

Source code instrumentation

```
#include <stdio.h>

#include <iostream>


int main()

{

    int num1, num2, i, gcd;

    std::cout << "Enter two integers: ";

    std::cin >> num1 >> num2;

    for(i=1; i <= num1 && i <= num2; ++i)

    {

        // Checks if i is factor of both integers

        if(num1%i==0 && num2%i==0)

            gcd = i;

    }

    std::cout << "G.C.D is " << gcd << std::endl;

    return 0;

}
```

GCD program

```
#include <stdio.h>
#include <iostream>
#include <fstream>
std::ofstream logFile("log_file.txt");
int main()
{
  logFile << "p1" << std::endl;
  int num1, num2, i, gcd;
  std::cout << "Enter two integers: ";
  std::cin >> num1 >> num2;
  for(i=1; i <= num1 && i <= num2; ++i)
  {
    logFile << "p2" << std::endl;
    // Checks if i is factor of both integers
    if(num1%i==0 && num2%i==0) {
      logFile << "p3" << std::endl;
      gcd=i;
    }
}
  std::cout << "G.C.D is " << gcd << std::endl;
  //return 0;
  logFile << "p4" << std::endl;
}
```

GCD program after instrumenting

Source code instrumentation

```
def enterStatement(self, ctx: CPP14Parser.StatementContext):
        if isinstance(ctx.parentCtx, (CPP14Parser.SelectionstatementContext,
                      CPP14Parser.IterationstatementContext)):
            # if there is a compound statement after the branchning condition:
            if isinstance(ctx.children[0], CPP14Parser.CompoundstatementContext):
                self.branch_number += 1
                new_code = '\n logFile << "p' + str(self.branch_number) + '" << endl; \n'
                self.token_stream_rewriter.insertAfter(ctx.start.tokenIndex, new_code)
      # if there is only one statement after the branchning condition then create a block.
            elif not isinstance(ctx.children[0],
                                (CPP14Parser.SelectionstatementContext,
CPP14Parser.IterationstatementContext)):
                self.branch_number += 1
                new_code = '{'
                new_code += '\n logFile << "p' + str(self.branch_number) + '" << endl; \n'
                new_code += ctx.getText()
                new_code += '\n}'
                self.token_stream_rewriter.replaceRange(ctx.start.tokenIndex,
ctx.stop.tokenIndex, new_code)
```

Source code instrumentation

- An example of executing the GCD program after instrumenting.

```
D:\AnacondaProjects\iust_start\input_source\basic>gcd
p(1)
Enter two integers: 24
18
p(2)
p(3)
p(2)
p(3)
p(2)
p(3)
p(2)
p(2)
p(2)
p(3)
p(2)
p(2)
p(2)
p(2)
p(2)
p(2)
p(2)
p(2)
p(2)
p(2)
p(2)
p(2)
G.C.D of 24 and 18 is 6
```

Source code instrumentation

- C++ source code instrumentation with ANTLR
  - https://m-zakeri.github.io/program-dynamic-analysis-with-antlr.html#program-dynamic-analysis-with-antlr

- Implement source code instrumentation for JAVA to measure node, edge, and prime path coverage
  - https://github.com/m-zakeri/CodA/
  - https://github.com/m-zakeri/DomainCoverage

Simple paths → **Prime paths** → **Test paths** → **Minimum cost test paths (MCTC)**

Instruments program → **Produce test data** → **Execute SUT with test data** → **Measure coverage**

Summary

- Applying the graph test criteria to control flow graphs is relatively straightforward
  - Most of the developmental research work was done with CFGs
- A few subtle decisions must be made to translate control structures into the graph.
- Some tools will assign each **statement** to a unique node
  - These slides and the book uses **basic blocks**.
  - Coverage is the same, although the bookkeeping will differ.