Software Testing
Lecture 4
More about Coverage

Justin Pearson

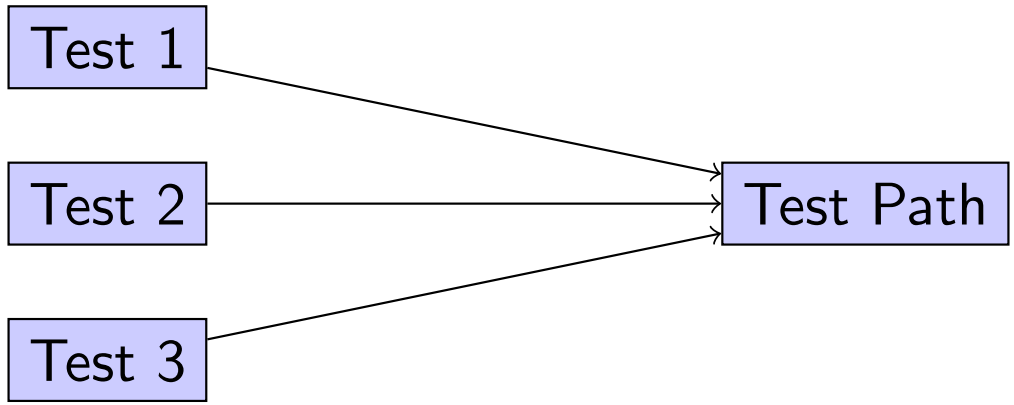
2019

Summary so far

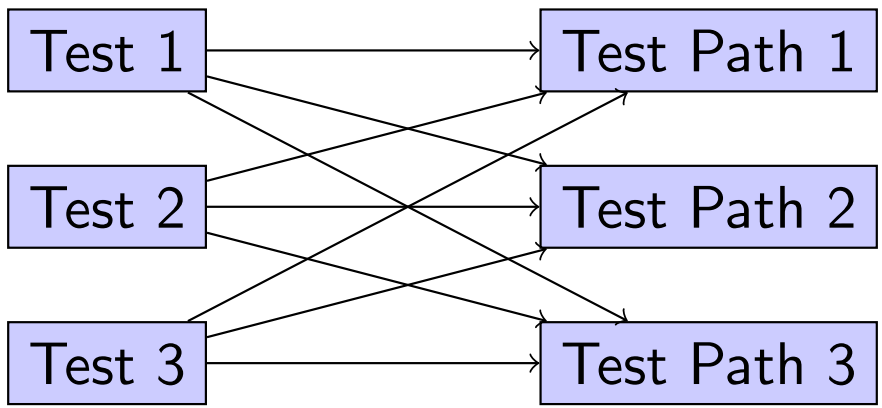
- Turing's halting theorem tells us in a strong sense that testing for correctness is impossible.
- Formalising programs as control flow graphs gives us a way to talk about testing.
- Node coverage corresponds to statement coverage, edge coverage corresponds to something like branch coverage. Covering all execution paths is impossible with loops, so there are various approximations.

Don't forget the distinction between syntactic and semantic reachability.

Test Cases and Test Paths



Many to one. Deterministic software, each test path has identical execution.



Many to many, non-deterministic software (you'll meet it all the time) a test can execute many test paths.

Thinking about testing

Important to separate:

- What coverage criteria are we trying to test?
  - Branch, statement, function points ...
- How do we test this property?
  - What paths do we cover in the control flow graph?
- What test cases (inputs and expected outputs) will make the paths execute?

More material from the last lecture

- Test path: represents the execution of a test case. It is a purely syntactic characterisation. A test path might not be semantically possible.
- Test case design: separate out *Test Requirements* describe the theoretic properties of test paths, based on the graph; while *Test Criterion* is what we want the test requirement to do.
- The *Satisfaction* problem: given some test requirement how do I find or does my set of test paths satisfy the requirements.

General idea in testing. Define your test requirements separately from the tests cases. Reformulate your requirements into test criteria and then try to find test paths that satisfy your test criteria.

Approaches to testing

- Black Box Testing: Test without looking at the code/hardware
- White Box Testing (clear box testing): Test the internal structure of the software

There is also grey box testing where you look find test cases that cover the specification and some aspect of the code. It is a grey area.

It is all about coverage

- Black box testing: test by covering the specification
- White box testing: test by covering the source code
  - Execution paths
  - Statements
  - Decision coverage
  - ...

Short version:

- Complete coverage is hard to define or impossible;
- So we have to find some approximation.

Testing and Coverage of Control Flow Graphs

- *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 $\mathrm{path}(T)$ that meets the test requirement.

General idea in testing. Define your test requirements separately from the tests cases. Reformulate your requirements into test criteria and then try to find test paths that satisfy your test criteria.

Node Coverage — Statement Coverage

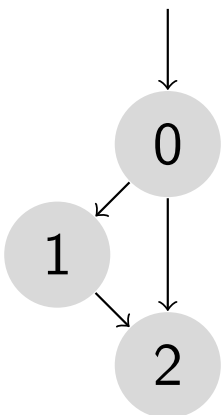
- Node Coverage (NC) : Test set T satisfies node coverage on graph G iff for every syntactically reachable node n in N, there is some path p in path(T) such that p visits n.

Edge Coverage — Branch Coverage

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

Is there any difference between node and edge coverage?

Difference between node and edge coverage



- Node coverage
  - Test requirement (TR) = $\{0, 1, 2\}$.
  - Test path = $[0, 1, 2]$.
- Edge Coverage
  - Test requirement (TR) = $\{(0, 1), (0, 2), (1, 2)\}$.
  - Test paths = $[0, 1, 2], [0, 2]$.

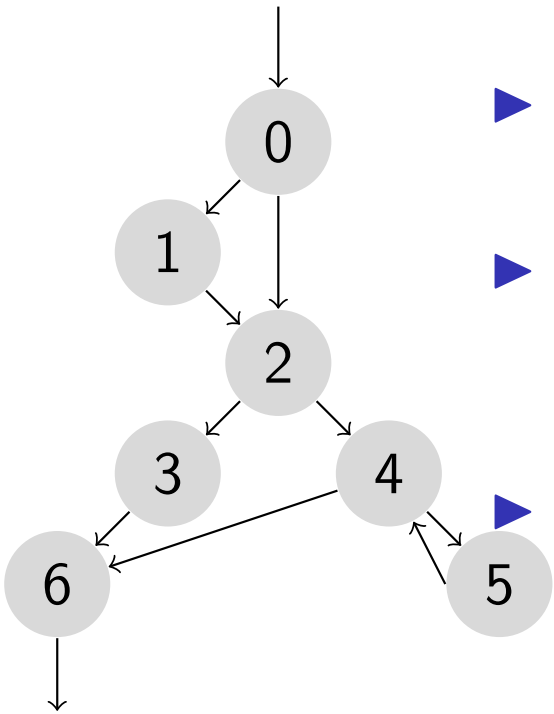
Complete Path Coverage

- Require that all paths are covered.

Often, there are too many paths. So you have to make an approximation, which is a comment theme in software testing.

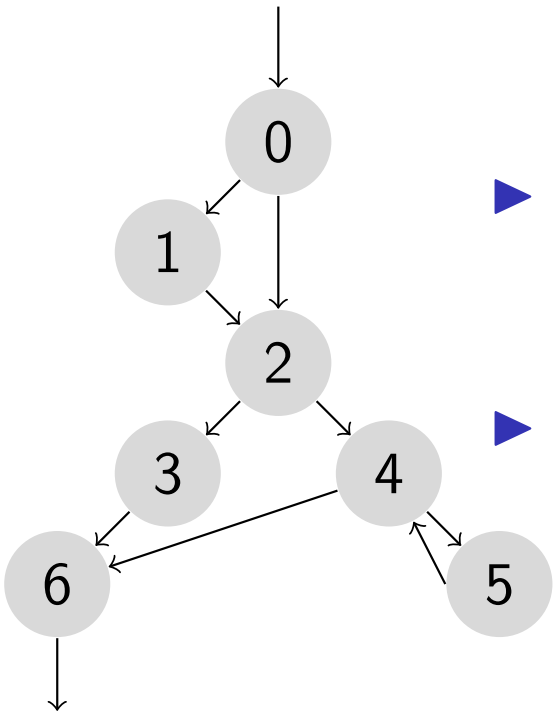
- Require that all paths up to length k are covered.
  - $k = 0$, node coverage.
  - $k = 1$, edge coverage.
  - $k = 2$, edge-pair coverage.

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: $[0, 1, 2, 3, 6]$, $[0, 2, 4, 5, 4, 6]$.
- Complete Path Coverage. Test paths: $[0, 1, 2, 3, 6]$,$[0, 1, 2, 4, 6]$,$[0, 1, 2, 4, 5, 4, 6]$, $[0, 1, 2, 4, 5, 4, 5, 4, 6]$, etc.

Structural Coverage Example



- 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
  - $[0,1,2,3,6],[0,1,2,4,6],\ [0,2,3,6]$
  - $[0,2,4,5,4,5,4,6]$.

Loops

There is a lot of theory, most of it is unsatisfactory.

- Don't be content with branch coverage
- Look at your loops.
  - Try to get them to execute zero times, once and many times.

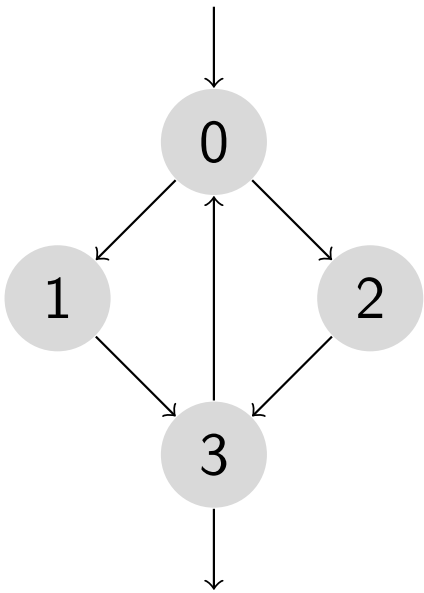
Loops

- If a graph contains a loop then it has an infinite number of paths.
- Thus you can not ask for complete path coverage.
- Attempts to deal with loops:
  - 1970s : Execute cycles once ([4, 5, 4] in previous example, informal)
  - 1980s : Execute each loop, exactly once (formalised)
  - 1990s : Execute loops 0 times, once, more than once (informal description)
  - 2000s : Prime paths

Simple and Prime Paths

- A *path* is simple if no node appears more than once except possible that the first and last node can be the same. Note that this gives us unique edges.
- A *prime path* of a graph is a simple path that is not a sub-path of any other simple path.

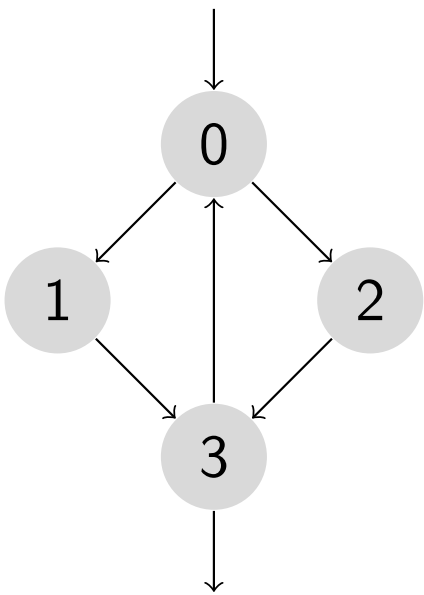
Simple Paths



- [0],[1],[2],[3]
- [0, 1],[0, 2],[1, 3],[2, 3],[3, 0]
- [0, 1, 3], [0, 2, 3], [1, 3, 0], [2, 3, 0],[3, 0, 1]
- [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1].

Prime Paths

Remove all simple paths that can be extended (either direction) to a longer simple path.

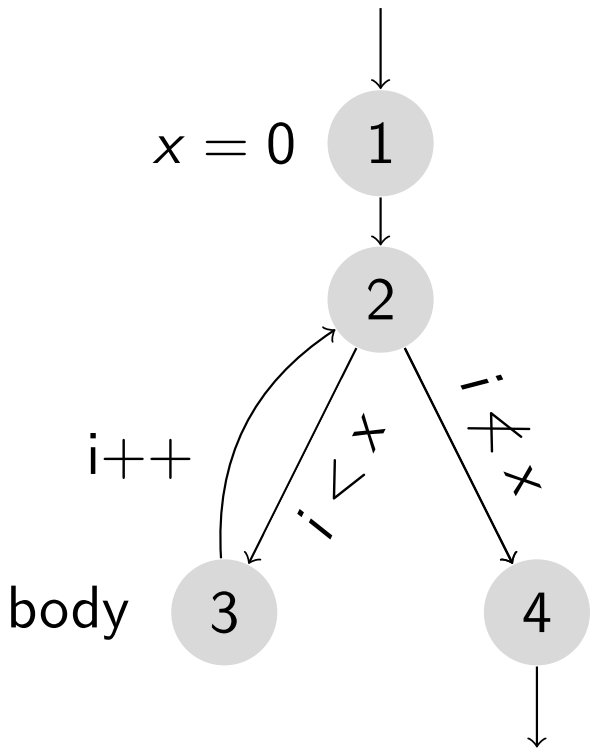


- ~~[0],[1],[2],[3]~~
- ~~[0, 1],[0, 2],[1, 3],[2, 3],[3, 0]~~
- ~~[0, 1, 3]~~, ~~[0, 2, 3]~~, ~~[1, 3, 0]~~, ~~[2, 3, 0]~~,~~[3, 0, 1]~~
- $[0, 1, 3, 0]$, $[0, 2, 3, 0]$, $[1, 3, 0, 1]$, $[2, 3, 0, 2]$, $[3, 0, 1, 3]$, $[3, 0, 2, 3]$, $[1, 3, 0, 2]$, $[2, 3, 0, 1]$.

In this case the prime paths are all the longest simple paths. Not always the case.

Prime Paths

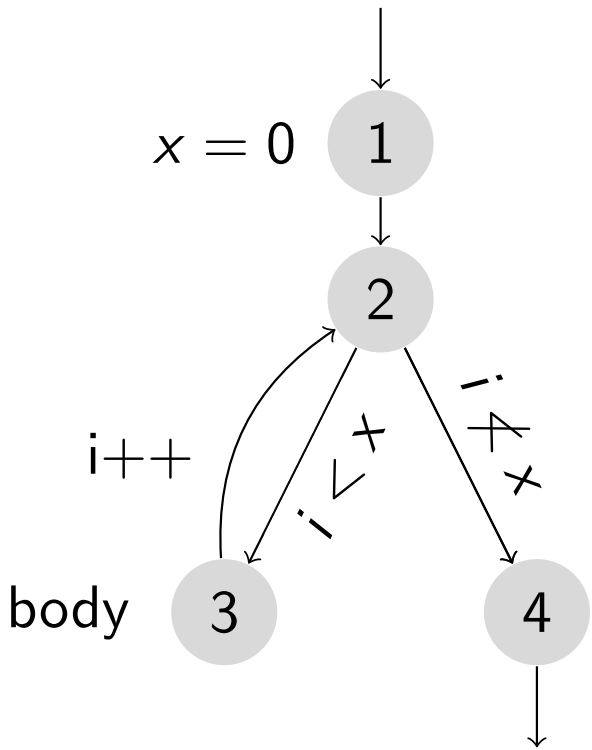
Enumerate all simple paths of length, 1,2,3, . . . then remove simple paths that can be extended. You will be left with the prime paths.



- [1],[2],[3],[4]
- $[1, 2]$, $[2, 3]$, $[2, 4]$, $[3, 2]$
- $[1, 2, 3]$, $[1, 2, 4]$, $[2, 3, 2]$, $[3, 2, 3]$,$[3, 2, 4]$
- We have to be careful about the paths of length 4.
  - $[1, 2, 3, 2]$ is not a simple path. Repeats 2 which is not at the beginning or the end.
- In fact there are no simple paths of length 4 in this graph.

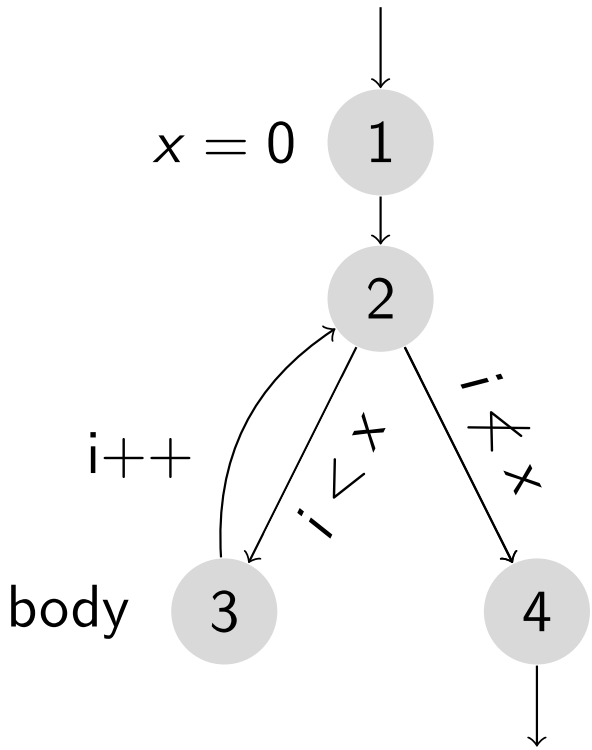
Prime Paths

Enumerate all simple paths of length, 1,2,3, . . . then remove simple paths that can be extended. You will be left with the prime paths.



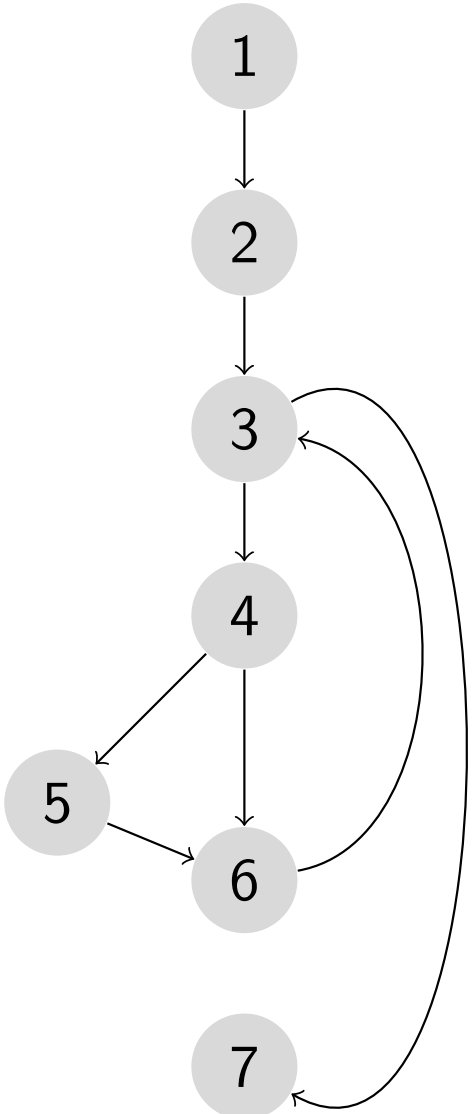
- ~~[1]~~,~~[2]~~,~~[3]~~,~~[4]~~
- ~~[1, 2]~~, ~~[2, 3]~~, ~~[2, 4]~~, ~~[3, 2]~~
- $[1, 2, 3]$, $[1, 2, 4]$, $[2, 3, 2]$, $[3, 2, 3]$

Prime Paths to Test Paths



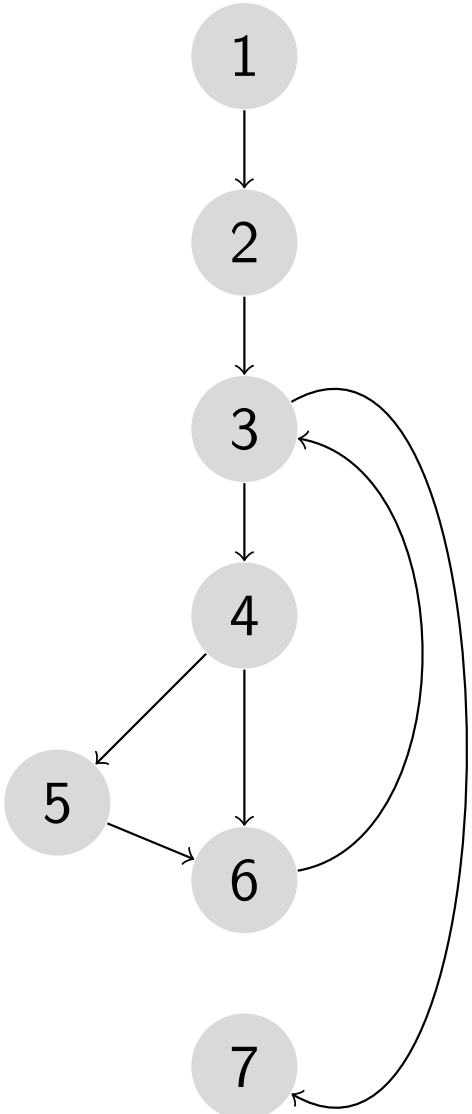
- $[1, 2, 3] \rightarrow [1, 2, 3, 2, 4]$ or $[2, 3, 2] \rightarrow [1, 2, 3, 2, 4]$
  - Execute loop once.
- $[1, 2, 4] \rightarrow [1, 2, 4]$
  - Execute loop zero times.
- $[3, 2, 3] \rightarrow [1, 2, 3, 2, 3, 2, 4]$
  - Execute loop more than once.

Simple Paths



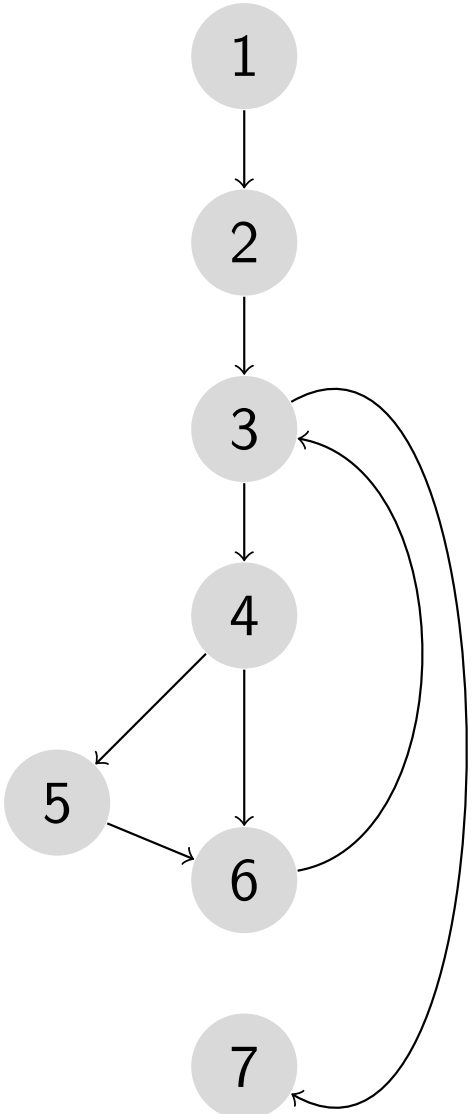
- $[1]$, $[2]$, $[3]$,$[4]$,$[5]$,$[6]$,$[7]$!
- $[1,2]$,$[2,3]$,$[3,4]$,$[3,7]$,
  $[4,5]$,$[4,6]$,$[5,6]$, $[6,3]$,$[6,3]$
- $[1,2,3]$, $[2,3,4]$, $[2,3,7]$!, $[3,4,5]$,
  $[3,4,6]$, $[4,5,6]$, $[4,6,3]$, $[4,6,3]$,
  $[5,6,3]$, $[5,6,3]$, $[6,3,4]$
- $[1,2,3,4]$, $[1,2,3,7]$!, $[2,3,4,5]$,
  $[2,3,4,6]$, $[3,4,5,6]$, $[3,4,6,3]$,
  $[3,4,6,3]$, $[4,5,6,3]$,$[6,3,4,5]$,
  $[4,5,6,3]$, $[4,6,3,4]$, $[5,6,3,4]$
- $[1,2,3,4,5]$, $[1,2,3,4,6]$,
  $[2,3,4,5,6]$,$[2,3,4,6,3]$, $[3,4,5,6,3]$,
  $[3,4,5,6,3]$,
- $[1,2,3,4,5,6]$.

Prime Paths



- ~~[1]~~, ~~[2]~~, ~~[3]~~,~~[4]~~,~~[5]~~,~~[6]~~,~~[7]~~!
- ~~[1, 2]~~,~~[2, 3]~~,~~[3, 4]~~,~~[3, 7]~~,
  ~~[4, 5]~~,~~[4, 6]~~,~~[5, 6]~~, ~~[6, 3]~~,~~[6, 3]~~
- ~~[1, 2, 3]~~, ~~[2, 3, 4]~~, ~~[2, 3, 7]~~!, ~~[3, 4, 5]~~,
  ~~[3, 4, 6]~~, ~~[4, 5, 6]~~, ~~[4, 6, 3]~~, ~~[4, 6, 3]~~,
  ~~[5, 6, 3]~~, ~~[5, 6, 3]~~, ~~[6, 3, 4]~~
- ~~[1, 2, 3, 4]~~, [1, 2, 3, 7]!, ~~[2, 3, 4, 5]~~,
  ~~[2, 3, 4, 6]~~, ~~[3, 4, 5, 6]~~, [3, 4, 6, 3] ,
  ~~[4, 5, 6, 3]~~ ,[6, 3, 4, 5], [4, 6, 3, 4],
  [5, 6, 3, 4]
- ~~[1, 2, 3, 4, 5]~~, ~~[1, 2, 3, 4, 6]~~, ~~[2, 3, 4, 5, 6]~~,
  [3, 4, 5, 6, 3],
- [1, 2, 3, 4, 5, 6].

Prime Paths



- $[1, 2, 3, 7]! \rightarrow [1, 2, 3, 7]$
  - Do the loop zero times.
- $[3, 4, 6, 3] \rightarrow [1, 2, 3, 4, 6, 3, 7]$
  - Do the loop once and do not do the `if`
- $[6, 3, 4, 5] \rightarrow [1, 2, 3, 4, 6, 3, 4, 5, 6, 3, 7]$,
  - Do the loop twice, once with the `if` and once without.
- $[4, 6, 3, 4] \rightarrow [1, 2, 3, 4, 6, 3, 4, 6, 3, 7]$
  - Do the loop twice, both times without taking the `if`.
- $[5, 6, 3, 4] \rightarrow [1, 2, 3, 4, 5, 6, 3, 4, 6, 3, 7]$
  - Do the loop twice, take the `if` once and once without, other way round from the previous case.
- $[3, 4, 5, 6, 3] \rightarrow [1, 2, 3, 4, 5, 6, 3, 7]$, loop once with one `if`.

Prime Paths: Summary

- Prime paths give you a good way of deriving a set of test cases that cover various combinations of loops and branches.
- There is no formal guarantee about *completeness*. As in all testing it just formalises a good compromise.

Model, define, and approximate

- Model what you want to test.
- Define coverage criteria.
- If coverage criteria is undecidable or require too many test cases then approximate.

Separate test requirements and test cases

- Have a reason for a test.
- Test requirements are the reasons for tests.
- You need to find satisfying test cases.

Example

```
int count_spaces(char* str) {
 int length, i,count;
 count = 0;
 length = strlen(str);
 for(i=1; i<length; i++) {
   if(str[i] == ' ') { count++; }
 }

}
```

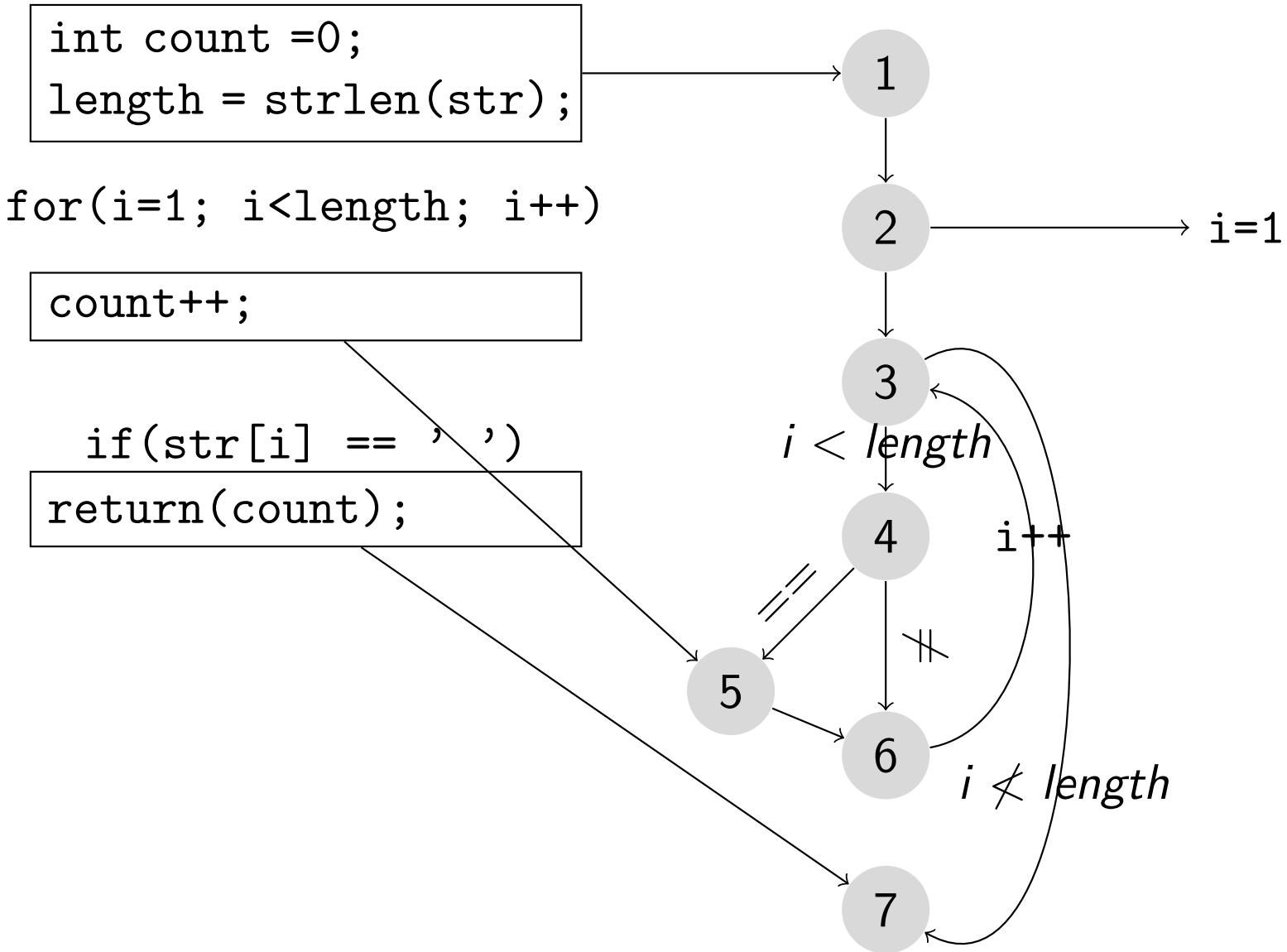
First Divide into Basic Blocks

```
int count =0;
length = strlen(str);
```

```
    for(i=1; i<length; i++)
       if(str[i] == ' ')
count++;
```

```
return(count);
```

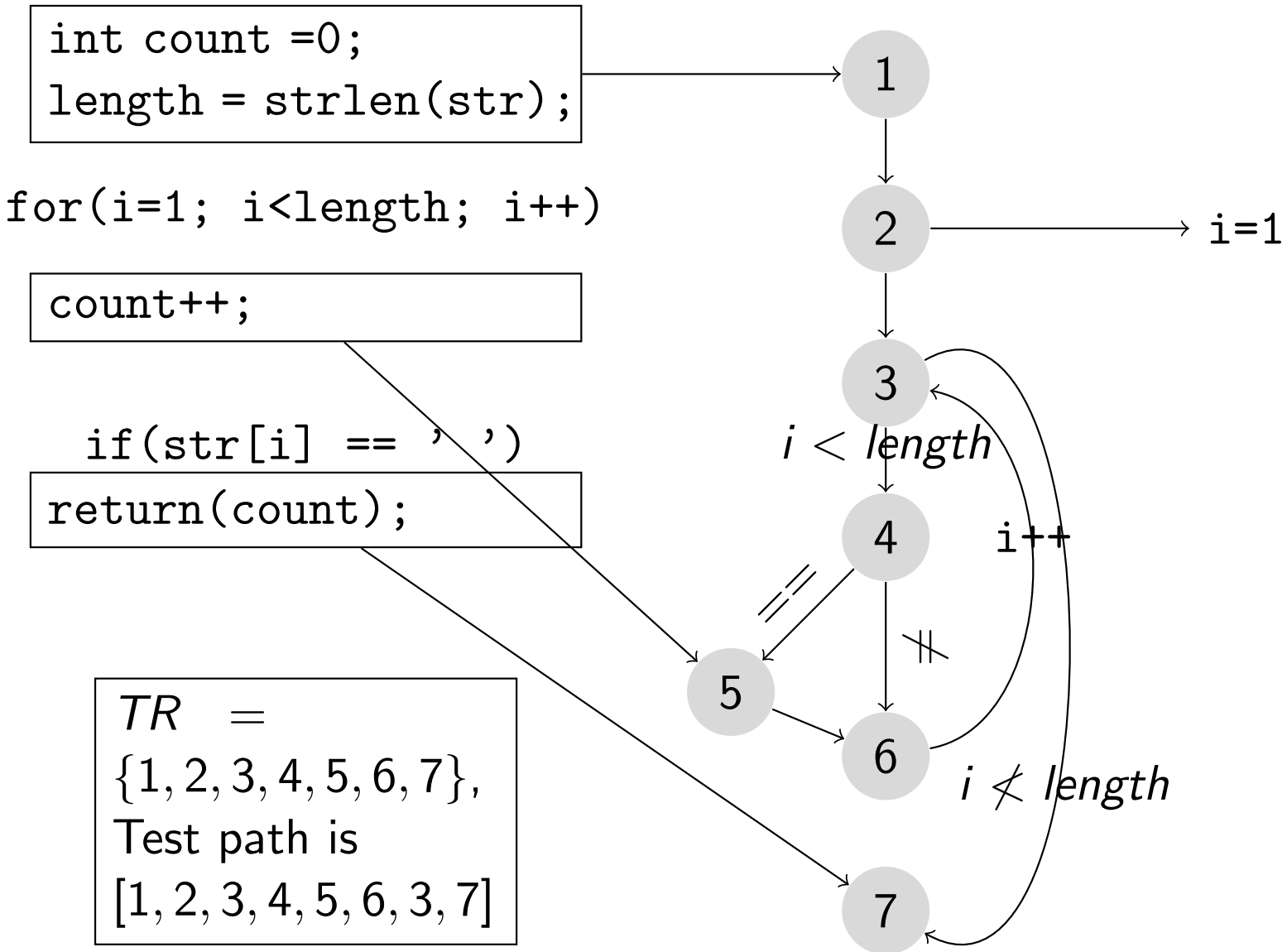
CFG


int count =0;
length = strlen(str);
for(i=1; i<length; i++)
count++;
if(str[i] == ' ')
return(count);
1
2
i=1
3
i < length
4
i++
5
6
i ≮ length
7

Test Path

- Remember a *test path* is a path that starts at an entry node and leaves at an exit node.

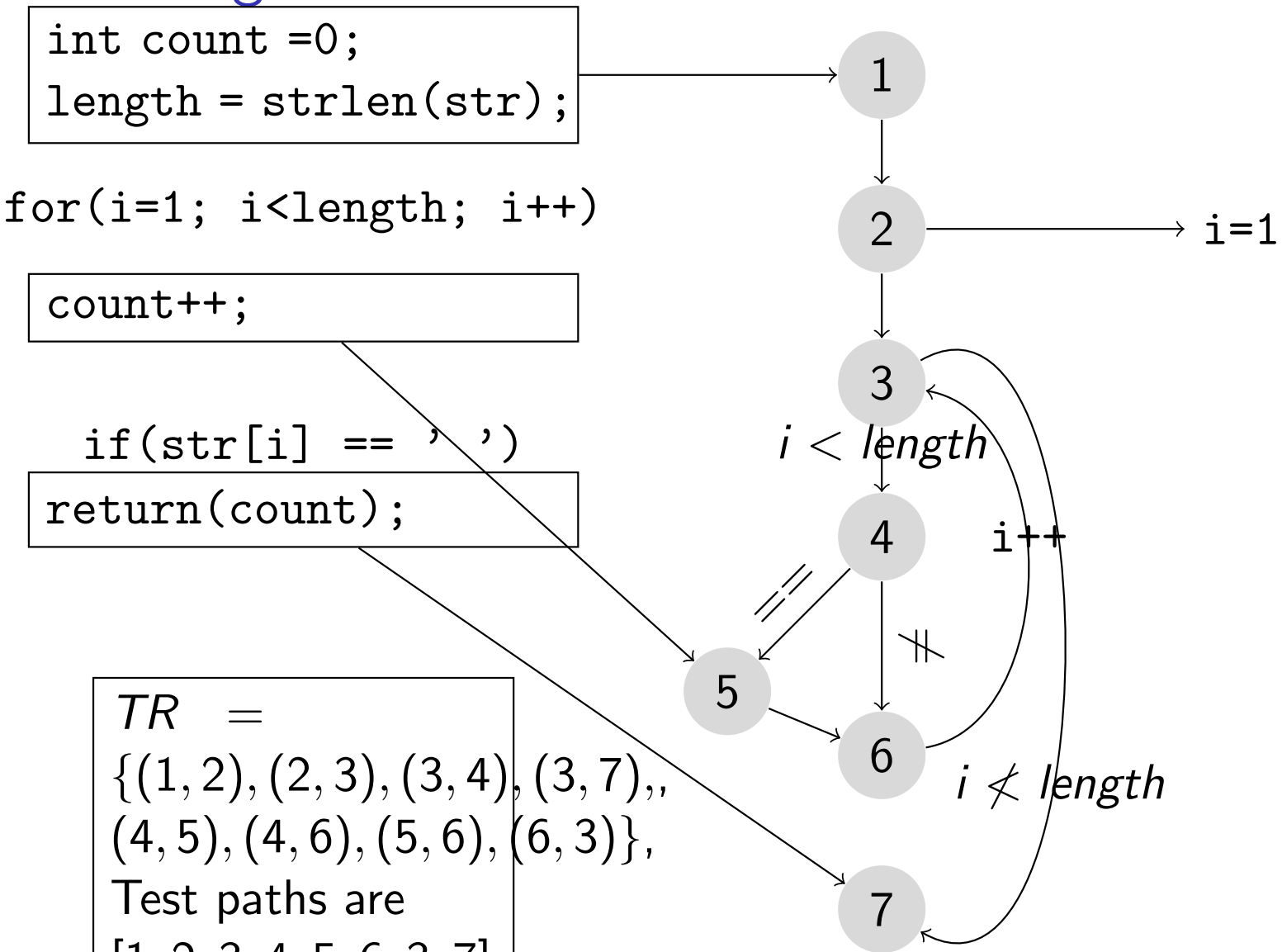
Node Coverage



Grey box testing

- Our test path $[1, 2, 3, 4, 5, 6, 3, 7]$ requires the loop to execute exactly once and to detect one space. So we might try the test case `(" ",1)` but this won't work. Don't forget that `i=1` in the loop body.
- Instead we have to use the test case `("H ",1)`
- Thinking about what the code should do, and trying to construct a test case corresponding to a path, we have uncovered a fault.

Edge Coverage



TR =
$\{(1,2),(2,3),(3,4),(3,7),,$
$(4,5),(4,6),(5,6),(6,3)\},$
Test paths are
$[1,2,3,4,5,6,3,7]$,
$[1,2,3,4,6,3,7]$,
$[1,2,3,7]$

Test Cases

- $[1, 2, 3, 4, 5, 6, 3, 7]$ (" ",1)
- $[1, 2, 3, 4, 6, 7]$ ("H",0)
- $[1, 2, 3, 7]$ ("",0)

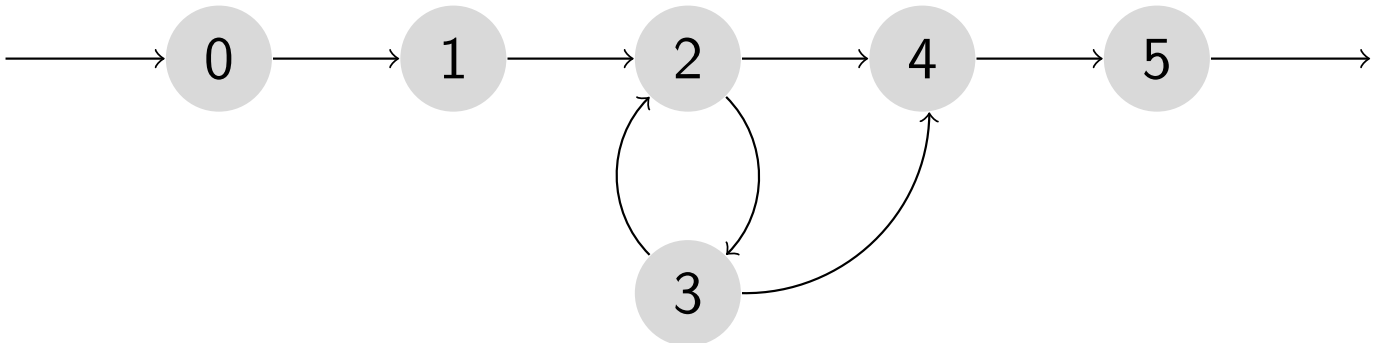
Relaxing test cases

- As we have seen, sometimes we have infeasible test cases.
  - This might be because there is a fault.
  - Or, that we have to do other things to get to the code. There might be a bit of setup code that we have to call first that is not in our path.
- A path p tours the path s if s is a sub-sequence of p.
  - $[1, 2, 3, 4, 6, 3, 4, 6, 3, 7]$ tours the test path $[1, 2, 3, 4, 6, 3]$ it also tours many other paths including $[4, 6, 3, 7]$.
  - Don't forget the difference between a *test path* and a *path*.

Relaxing Test Cases

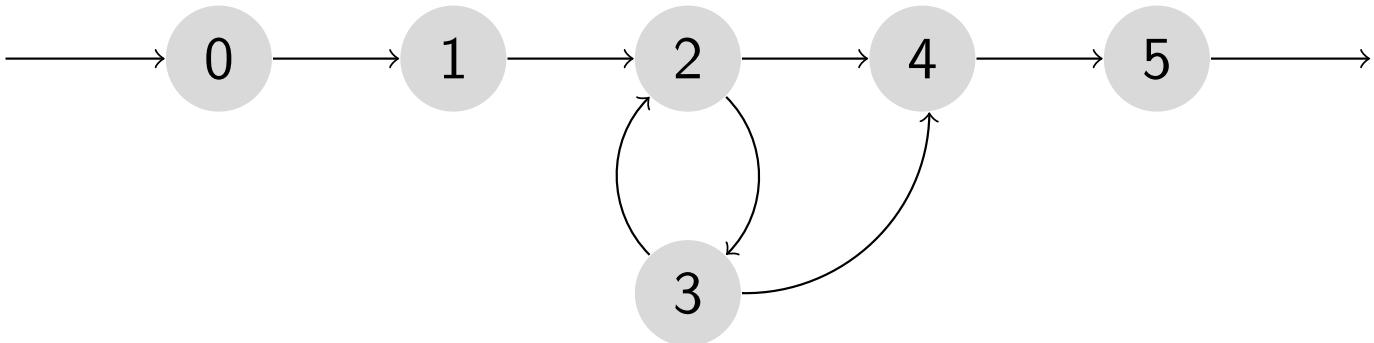
- A *test path* p is set to *tour* sub-path q with *side-trips* if every edge that is in q is also in p in the same order.
- A *test path* p is set to *tour* sub-path q with *detours* if every node that is in q is also in p in the same order.

Sidetrips



The path $[0, 1, 2, 3, 2, 4, 5]$ tours the path $[0, 1, 2, 4, 5]$ a side trip.

Detours



The path $[0, 1, 2, 3, 4, 5]$ tours the path $[0, 1, 2, 4, 5]$ with a detour.

Infeasible test requirements

- An infeasible test requirement *cannot be satisfied*
  - Unreachable statement (dead code)
  - Can only be executed if a contradiction occurs $X > 0 \wedge X < 0$.
  - Always check against the specification, it could be a fault.

Infeasible test requirements

- Most test criteria have some infeasible test requirements.
- It is usually undecidable if all test requirements are feasible (halting problem again).
- Allowing side trips might weaken the test cases, but allows more feasible test cases.
- Practical recommendation, best effort touring. Allow as many as possible without side-trips; only allow side-trips on infeasible test paths.

Summary

- Prime paths provide some intuition for test cases for loops.
- With nested loops and complicated branching, prime-paths give us test cases that you might not look for.
- There are various ways of relaxing test paths when they are infeasible.

Do I expect you to draw control flow graphs?

- For the exam yes.
- But, why bother? For large pieces of code you will never bother. For troublesome small bits of code, *prime paths* would be a useful tool to look for test cases you might not consider.
- Formalisation allows you to think about what coverage means for a piece of code.