Graph-based Test Coverage (A&O Ch. 1, 2.1 – 2.3)

(2nd Ed: 2,3,5,7.1 – 7.3)

Course Software Testing & Verification

2024/25

Wishnu Prasetya & Gabriele Keller

Plan

- The concept of “test coverage”
- Graph-based coverage

Note: graph-based coverage is probably the most well known concept of coverage after line coverage. It is a good foundation towards more advanced concept of coverage.

Are we good now?

- So now you know how to write tests (at least, unit tests).
- Next, you came up with a bunch of tests.
- Question: are these tests “enough” ?

Test Coverage

Doing more tests improves the completeness of our testing, but at some point we have to stop (e.g. we run out of money). Let's try to provide a measure:

Test coverage: a *measure* to compare the relative completeness of our testing (e.g. to say that a "test set" T_1 is relatively more complete than another set T_2).

As a measure, it is *quantitative*.

Simple example: line coverage

```
foo(x,y) {
   while (x>y)
      x=x-y ;
   if (x<y)
      return -1
   else
      return x }
```

- A ***test-case***: another name for a "test".
- A ***test-set*** (Def 1.18/2nd Ed 3.20): is just a set of test-cases.
- We can for example require that the test set of foo "covers" every line in the code of foo. (a test set **covers** a line, if there is one test case whose execution visits the line)

Simple example: line coverage

```
foo(x,y) {
   while (x>y)
      x=x-y ;
   if (x<y)
      return -1
   else
      return x }
```

As a quantitative measure we can talk about:

- Which lines are covered
- How many lines are covered
- Percentage of the lines covered

Is it a good measure?

```
foo(x,y) { while (x>y)  x=x-y ; if (x<y) return -1 else return x }
```

A single test will give full line coverage, but this obviously misses testing some logic of foo().

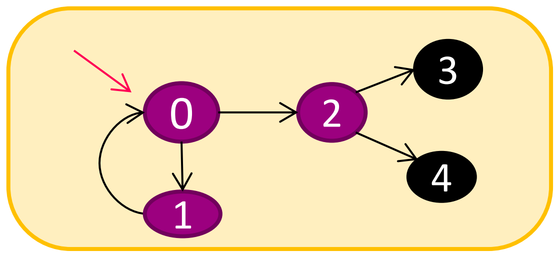
```
foo(x,y) {
   while (x>y)
       x=x-y ;
   if (x<y)
       return -1
   else
       return x }
```

A less trivial example: a test set with full line coverage may miss the scenario where the loop is immediately skipped.

Graph-based test coverage

foo(x,y) { **while** (x>y)   x=x-y ; **if** (x<y) return -1 **else** return x }

0 1 2 3 4



- Abstractly, a program is a set of branches and branching points. An execution flows through these branches, to form a path through the program.
- This can be captured by a so called *Control Flow Graph* (CFG, Legard 1970), such as the one above.
- AO gives you a set of graph-based coverage concepts; most are stronger than line coverage.

Graph terminology

- (Sec 2.1/2nd Ed 7.1) A ***graph*** is described by (N,N_0,N_f,E). E is a subset of $N \times N$.
  – directed, for now: no labels, no multi-edges.

- A ***path*** p is a sequence of nodes [n_0, n_1, ... , n_k]
  – non-empty
  – each consecutive pair is an edge in E

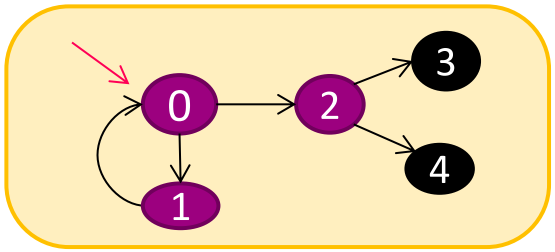
- ***length*** of p = #edges in p = natural length of p – 1

CFG and Test Path

```
foo(x,y) { while (x>y)   x=x-y ;  if (x<y) return -1 else return x }
```

0 1 2 3 4

CFG of foo:



- Control Flow Graph (CFG) is a graph representing a program in terms of how it transitions from one statement to another. We furthermore assume:
  - There is only one initial node.
  - Any execution ends in a terminal node (this implies that if a program can terminates by throwing an exception, a terminal node should be added to model this).
- A <u>*test-path*</u> (Def. 2.31) is a path in the CFG representing the execution of a test case. It should therefore starts at the CFG's initial node, and ends in a terminal node of the CFG.

More terminology

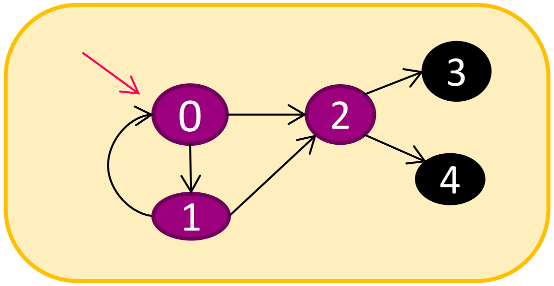
- O&A usually define "coverage" with respect to a "**TR**" = a **set** of *test requirements.*
- For example:

  TR = { "cover line k" | k is a line in P's source }

  Or simply: TR = { k | k is a line in P's source }, and implicitly we mean to "cover" every k.

- Def. 1.21/2nd Ed 5.24. *A coverage criterion = a TR to fulfill/cover.*

CFG-based TR



- A path can be used to express a test-requirement. For example we can specify as our TR:

$$TR = \{ [0,2], [1,2], [3] , [4] \}$$

- But what does “covering” a path e.g. [0,2] mean? (there are several plausible definitions btw)

Paths as test requirements

- Some plausible definitions of "test path *p* covers a required path *q*" :
  1. all nodes in *q* appear in *p.*
  2. *q* is a **subpath** of p (*p* can be written as *prefix* ++ *q ++ suffix*, for some *prefix* and *suffix.*
  3. *q* is a **subsequence** of *p* (there is a way to delete some elements from *p* to obtain *q).*
- Example:
  - [1,2] is a subpath of [0,1,2,3]
  - [2,4] and [1,3] are <u>not</u> a subpath of [0,1,2,3]
  - [1,3] is a subsequence of [0,1,2,3]
- OA use the term "*p* ***tours*** *q*" = *q* is a *subpath* of *p = OA's default definition of "covering q".*

Some basic graph coverage criteria

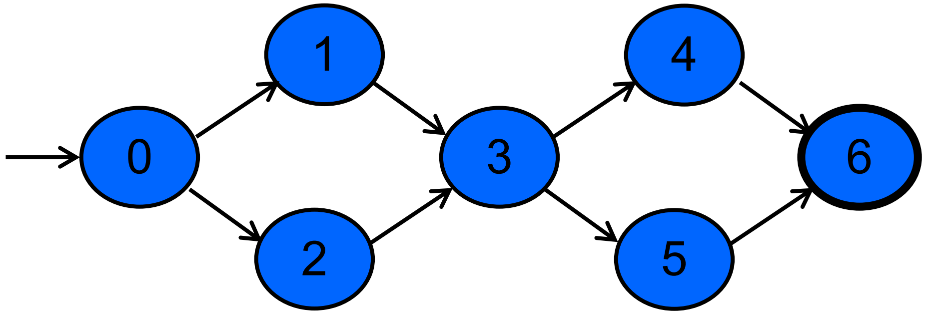
- (C2.1/2nd Ed. C7.7) **Node coverage**: the *TR* is the set of paths of length 0 in G.

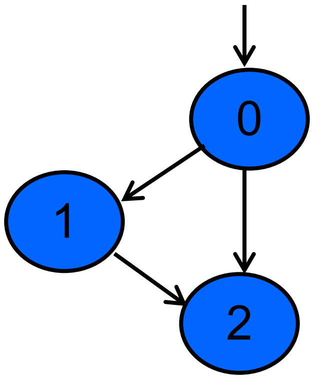
- (C2.2/2nd Ed. C7.8) *TR* is the set of paths of length at most 1 in G.
  - So, what does this criterion tell?
  - Why "at most" ?

- (C2.3/2nd Ed. C7.9) **Edge-pair coverage**: the TR contains all paths of length at most 2.

Examples

- Consider these examples:





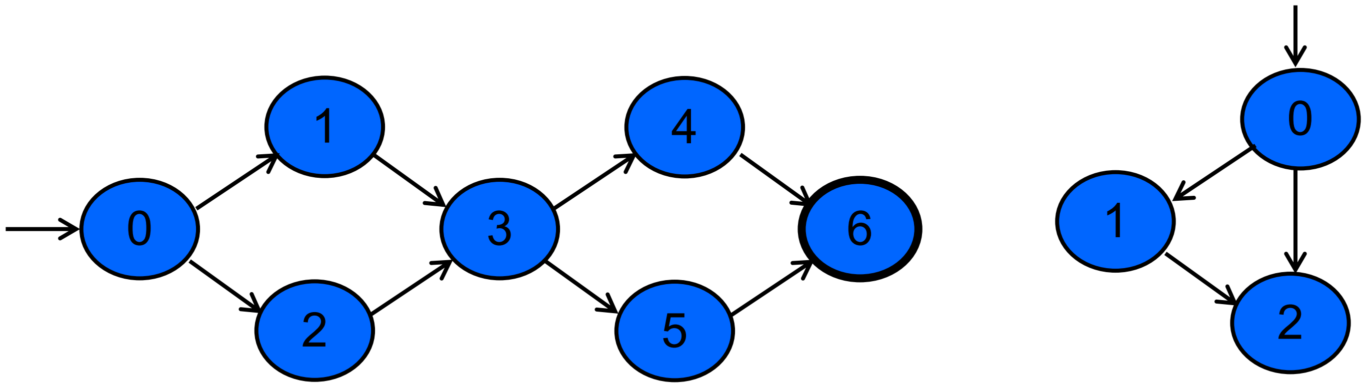
- What do we have to cover in C2.1, C2.2, and C2.3?
- What would be the minimal test set for each?

Subsumption

Def 1.24/2nd Ed. 5.29:

A coverage criterion *C1* ***subsumes*** (stronger than) *C2* iff every test-set that satisfies *C1* also satisfies *C2*.

Examples subsumption

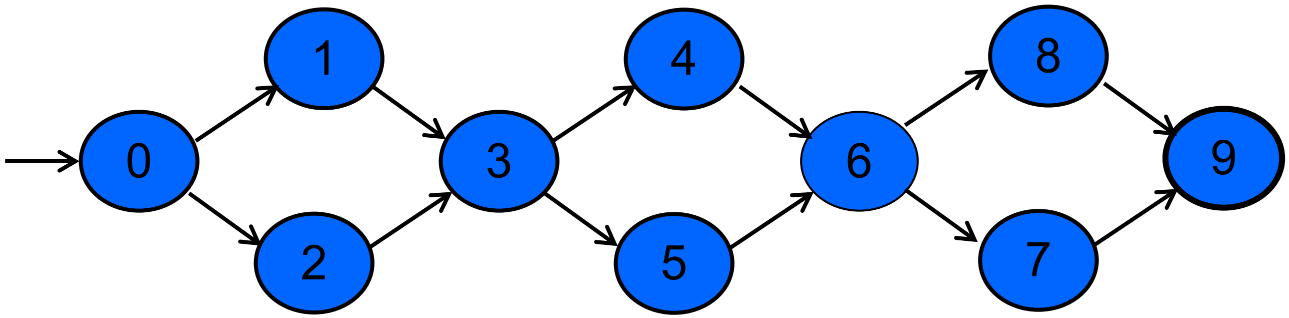


- C2.2 subsumes 2.1 (but not the other way around).
- C2.3 subsumes C2.2 (but not the other way around).

Few notes on TR and subsumption

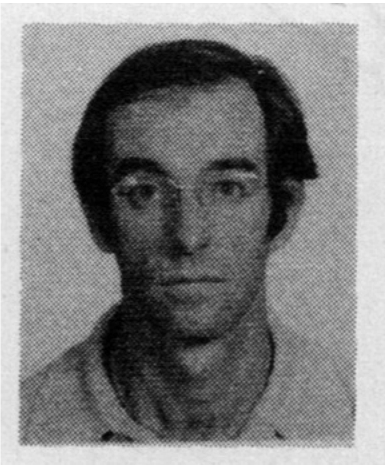
- Consider a TR has N elements, and assume all are feasible:
  - There exists a test set with at most N elements that fully cover this TR.
  - There may be a test set with less that N elements that fully cover the TR.
- Consider two coverage criteria C1 and C2 and C1 does **not** subsume C2.
  - There exists a program P and a test set for P fulfilling C1 but not C2.
  - But keep in mind that there may be a program Q, where any test set for Q that fulfills C1 would also fulfill C2

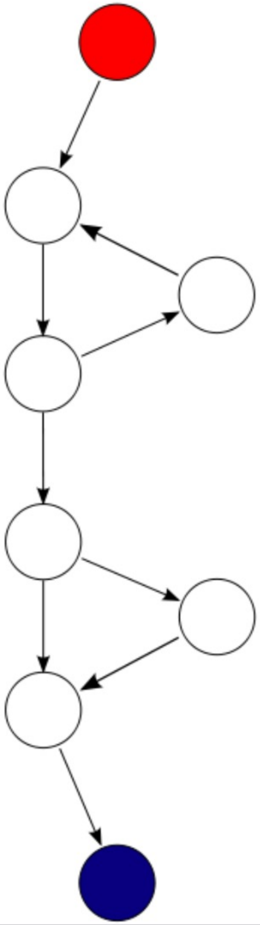
What if we insists on covering ALL paths?



- Path coverage (to cover all paths in the CFG) in the **strongest** graph-based coverage.
- It is challenging: #paths in a CFG may explode.
- Additionally, if the CFG contains a cycle, #paths becomes unbounded.

McCabe Complexity of a Program





The "cyclomatic complexity" of a program is:

$M = E - N + P$

$M = E - N + 2$

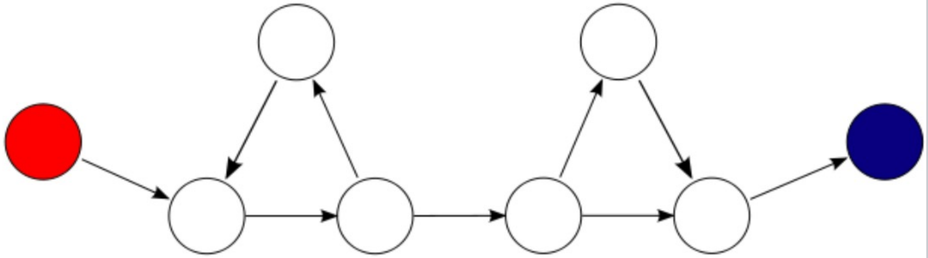
🤔…. but what does this represent??

McCabe's Original Theorem

Definition 1: The cyclomatic number V(G) of a graph G with n vertices, e edges, and p connected components is

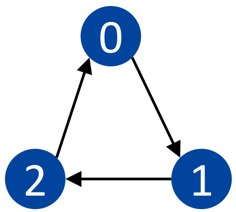
$$v(G) = e - n + p.$$

Theorem 1: In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits.



Note that this CFG does not actually form a strongly connected graph

Linearly Independent Circuits



linerarly independent circuits = 1

Note that in a cycle like this #edge = #node

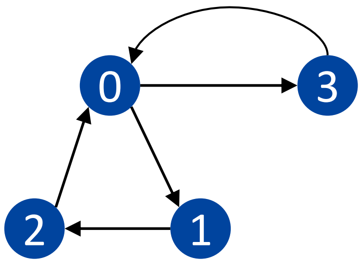
So, #lics = E – N + 1 ?

Circuit: a path that starts and ends in the same node, and never repeats an edge.

Examples: [0,1,2,0] and [1,2,0,1]

A set of circuits **is linearly independent** if each circuit has an edge that others do not have.

Some note: circuit vs simple path

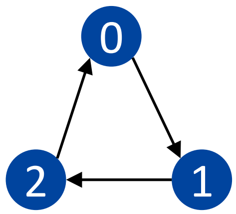


- A circuit does not pass the same edge twice.
- A simple path does not pass the same node twice, except the start and end node.

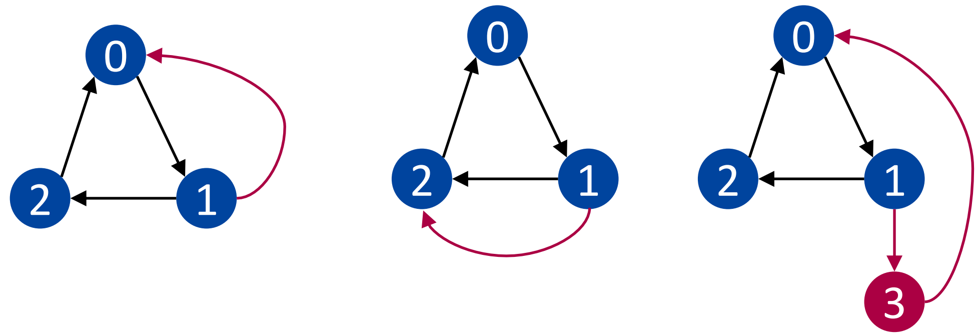
- [0,1,2,0] and [0,3,0] are circuits. They are also (cyclic) simple paths.

- [0,3,0,1,2,0] is also a circuit, **but** not a simple path.

Linearly Independent Circuits

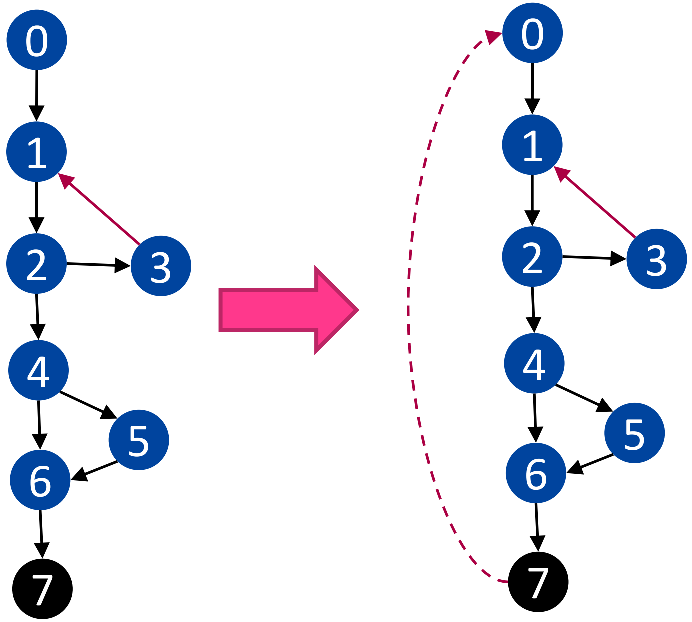


linerarly independent circuits = 1



- Adding new nodes/edges, but we add more edges than nodes → this introduces new cycles.
- In the above examples:
- we add one more cycle
- # linearly independent circuits is now 2

Example

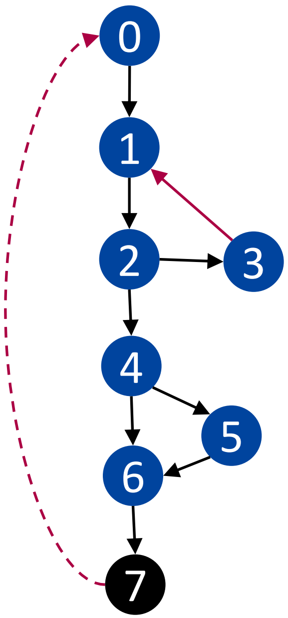


Original CFG is not a strongly connected component.

Add one fake edge to make it strongly connected

- # linerarly independent circuits = 3
- Can you give them?
- #lics = E – N + 1 (McCabe theorem)
- E = number of edges in the extended CFG = original E + 1
- So, #lics = $E_{original}$ – N + 2

Relation with test coverage



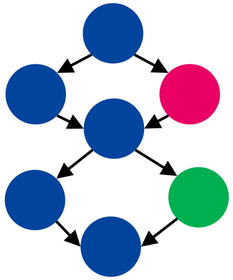
Aim to cover at least all independent “circuits” in your program:

- [0,1,2,4,6,7,~~0~~] , [0,1,2,4,5,6,7,~~0~~], [1,2,3,1]
- You can remove the fake edge from the TR.

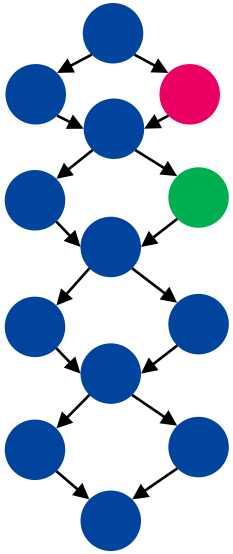
Observations:

- The paths in McCabe TR are linearly independent, so each has something unique.
- McCabe number is not the same as #paths in the program. The later can be exponential or unbounded. #McCabe TR grows much slower.

#McCabe-TR growth

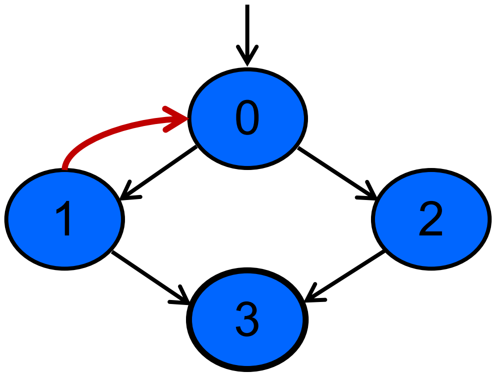


#McCabe-TR = #independent paths = 3
#test cases needed to cover all edges = 2
#test cases needed to cover McCabe = 3



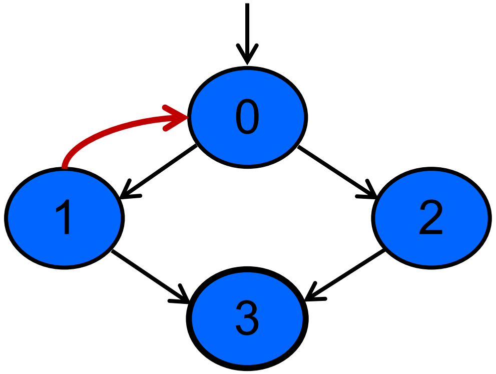
#McCabe-TR = #independent paths = 5
#test cases needed to cover all edges = 2
#test cases needed to cover McCabe = 5

Prime paths coverage



- A ***prime path*** *is a* simple path that is not a subpath of another simple path.
- A ***simple path*** *p* is a path where every node in *p* appears only once, except the first and the last which are allowed to be the same.

Another example

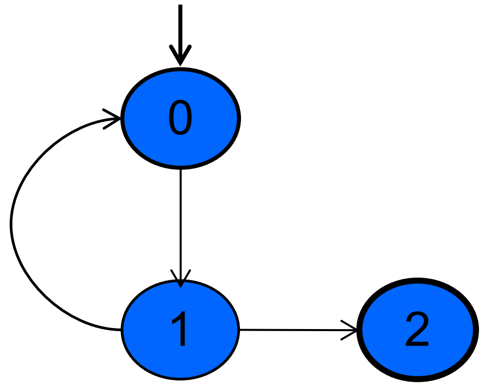


Starting from	prime paths
0	[0,1,0] , [0,1,3]
1	[1,0,1] , [1,0,2,3]
2	-
3	-

- (C2.4/2nd Ed. C7.10) PPC: the *TR* is the set of all prime paths in *G.*
- Strong, but still finite.

Few notes on PPC

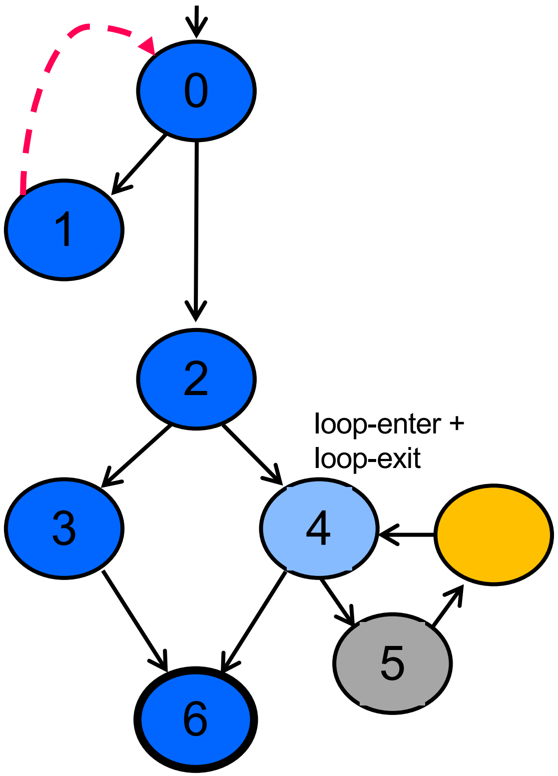
Recall: #TR may not reflect the #test cases you need. Example:



PPC's TR = { 012, 010, 101 }

We can cover this with just one test path: { 01012 }

Identifying Prime Paths



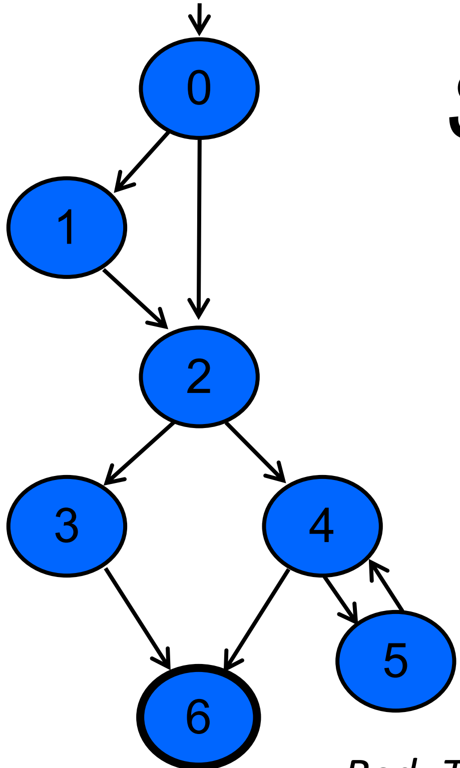
- A cycle with n nodes generate n cyclic pps.
- Not always the case that there is a non-cyclic pp starting from the program entrance.
- A loop may have multiple entry and exit nodes.
- There are non-cyclic pps that end in prev(loop-exit).
- There are non-cyclic pps that start in next(loop-entry).

Identifying the PPCs

AO's Algorithm

- How long can a prime path be?
- It follows that they can be systematically calculated, e.g. :
    1. "invariant": maintain a set *S* of simple but **not** right-maximal paths, and *T* of "right"-maximal simple paths.
    2. Repeat: "right-extend" the paths in *S*; we move them to *T* if they become right-maximals.
    3. (2) will terminate.
    4. Remove members of *T* which are subpaths of other members of *T*.

Example



S :

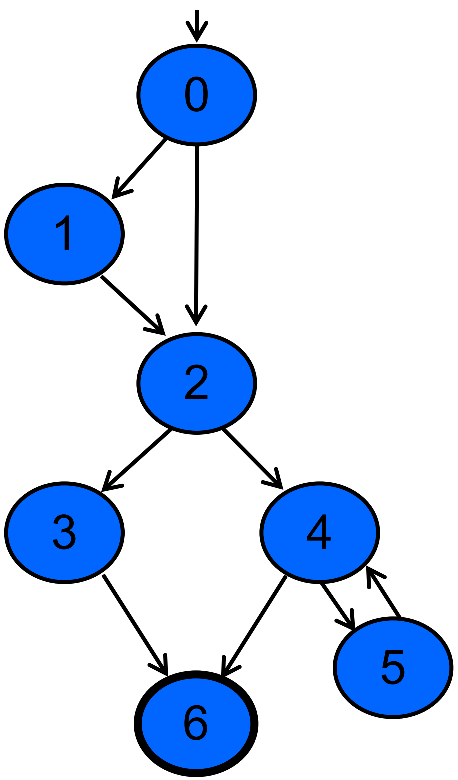
[0]
[1]
[2]
[3]
[4]
[5]
[6] !

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

[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] *

Red: T, consisting of paths which become right-maximal.
We also mark :
** → cycle ; definitely a prime path*
! → right-maximal, non cycle

Example



[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] *

[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] !

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

S is now empty; terminate.

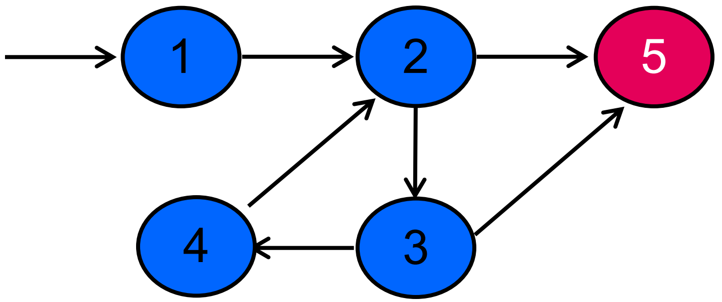
Prime paths =
- all the (*) paths, plus
- non-cyclic right-max simple paths which are not a subpath of another right-max simple path.

Unfeasible test requirement

- A test requirement is **unfeasible** if it turns out that there is no real execution that can cover it
- For example, we cannot test a Kelvin thermometer below 0^o.

Typical unfeasibility in loops

```
i = a.length-1 ;
while (i≥0)  {
    if (a[i]==0)  break ;
    i--
}
```



123425 does not tour 125, but with sidetrips it does.

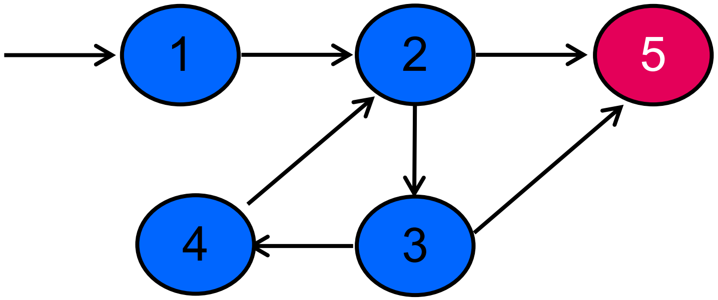
- Some loops ***always*** iterate at least once. E.g. above if *a* is never empty → prime path 125 is unfeasible.
- (Def 2.37/2nd Ed. 7.36) A test-path *p* tours *q* ***with sidetrips*** if all *edges* in *q* appear in *p*, and they appear in the same order.

Generalizing Graph-based Coverage Criterion (CC)

- Obviously, sidetrip is weaker that strict touring.
- Every CC we have so far (C2.1, 2.2, 2.3, 2.4, 2.7) can be thought to be parameterized by the used concept of “tour”. We can opt to use a weaker “tour”, suggesting this strategy:
  - First try to meet your CC with the strongest concept of tour.
  - If you still have some paths uncovered which you suspect to be unfeasible, try again with a weaker touring.

Oh, another scenario...

```
i = a.length-1 ;
while (i≥0)  {
   if (a[i]==0)  break ;
   i--
}
```

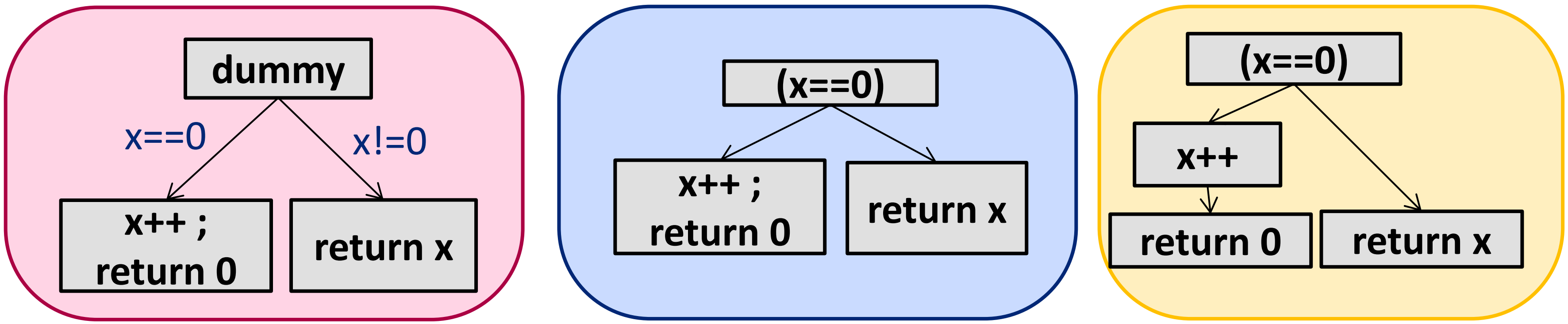


- Often, loops always break in the middle. E.g. above if *a* always contain a 0, then 125 is infeasible.
- Notice: e.g. 1235 does **not** tour 125, not even with sidetrip!
- (Def 2.38/2nd Ed. 7.37) A test-path *p* tours *q* ***with detours*** if all nodes in *q* appear in *p*, and they appear in the same order (*q* is a subsequence of *p*).
- Weaker than sidetrip.

Mapping your program to its CFG

various ways, depending on the purpose

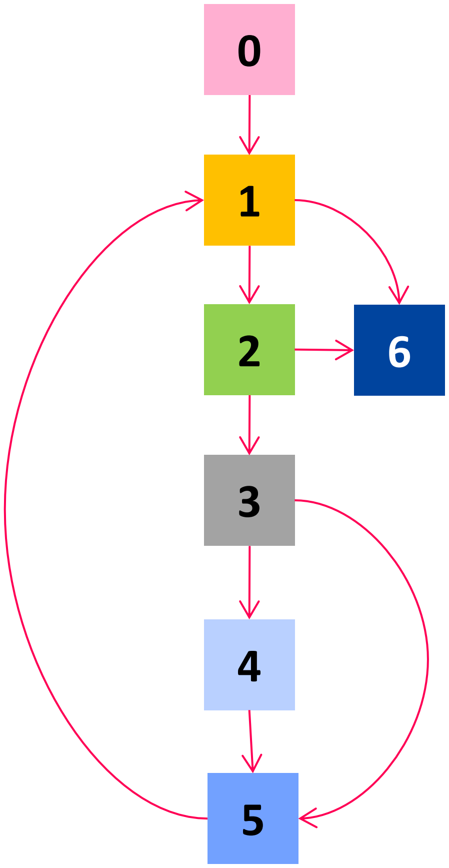
P1(x) { **if** (x==0) { x++ ; return 0 } **else** return x }



- See Sec. 2.3.1. AO (2nd Ed. 7.3.1) use left; I usually use middle.
- (Sec 2.3.1/2nd Ed. 7.3.1) Define a ***block*** a maximum sequence of “instructions” so that if one instruction is executed, all the others are always executed as well. Note that this implies:
  - a block does not contain a jump or branching instruction, except if it is the last one
  - no instruction except the first one, can be the target of a jump/branching instruction in the program.

Mapping your program to its CFG

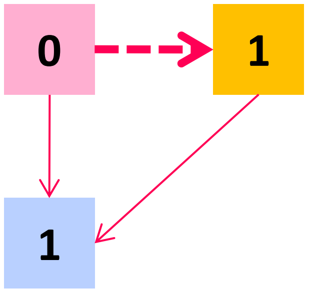
```
P2(a) {
  for (int i=0;  i<a.length ; i++)  {
     if (a[i]==0)  break ;
     if (odd a[i]) continue ;
     b[i] = a[i] ; a[i] = 0
  }
}
```



Add "end-of-program" as a virtual node.

Discussion: exception

```
P3(...) {
    try {  File f = openFile("input")
           x = f.readInt()
           y = (Double) 1/x    }
    catch (Exception e) {  y = -1 }
    return y
}
```



Every instruction can potentially throw an exception. Yet representing each instruction as its own node in our CFG will increase the size of the graph, and thus also the size of the TR. We can decide to abstract over this, as in the CFG above. But you should be aware of the loss of information. E.g. when a test set manages to cover the exception edge 0→1, we are still unsure if we have tested all sources of this exception.

More discussion

- What to do with dynamic binding ?

```
register(Course c, Student s) {
   if (! c.full()) c.add(s) ;
}
```

You don't know ahead which "add" will be called; it depends on the run-time type of c. E.g. it may refuse to add certain type of students. Graph-based coverage is not strong enough to express this aspect.

- What to do with recursion?

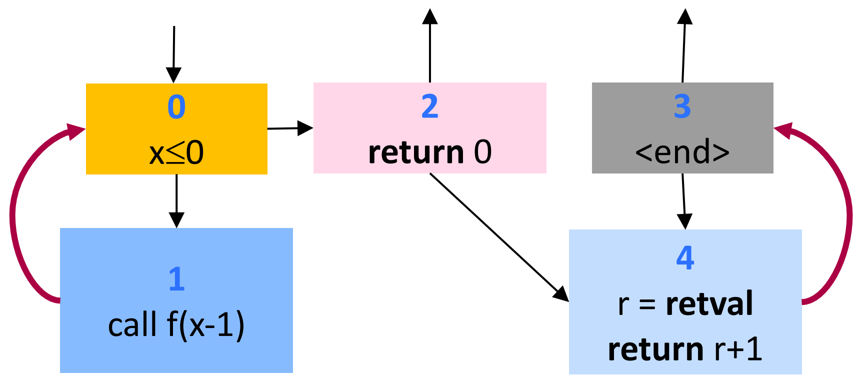
```
f(x) {
  if  (x≤0)  return 0
  else  return f(x-1)
}
```

```
g(x) {
  if  (x≤0)  return 0
  else  { x = g(x-1) ; return x+1 }
}
```

CFG of recursive programs

Constructing the CFG of a recursive program is more complicated.
Let’s look at two simple examples.

```
f(x) {
  if (x≤0) return 0
  else {
    r = f(x-1)
    return r+1
} <end>
```



*Constrain: **both** cycles should be iterated in equal number of times*

One more example

How about this one?

```
h(x) {
  if  (x≤0)  return 0
  if (odd(x))  { x = h(x-1) ; return x+1 }
  else { x = h(x-2) ; return x+2 }
}
```