Testing: Coverage and Structural Coverage

Testing, Quality Assurance, and Maintenance
Winter 2019

Prof. Arie Gurfinkel

based on slides by Prof. Marsha Chechik and Prof. Lin Tan
Introduction to Software Testing
How would you test this program?

floor(x) is the largest integer not greater than x.

def Foo (x, y):
    """requires: x and y are int
    ensures: returns floor(max(x,y)/min(x, y))""
    if x > y:
        return x / y
    else
        return y / x
Testing

Static Testing [at compile time]
  • Static Analysis
  • Review
    – Walk-through [informal]
    – Code inspection [formal]

Dynamic Testing [at run time]
  • Black-box testing
  • White-box testing

Commonly, testing refers to dynamic testing.
Complete Testing?

Poorly defined terms: “complete testing”, “exhaustive testing”, “full coverage”

The number of potential inputs are infinite.

Impossible to completely test a nontrivial system
  • Practical limitations: Complete testing is prohibitive in time and cost [e.g., 30 branches, 50 branches, ...]
  • Theoretical limitations: e.g. Halting problem

Need testing criteria
Test Case

Test Case: [informally]

• What you feed to software; and
• What the software should output in response.

Test Set: A set of test cases

Test Case: input values, expected results, prefix values, and postfix values necessary to evaluate software under test

Expected Results: The result that will be produced when executing the test if and only if the program satisfies its intended behaviour
Test Requirement & Coverage Criterion

**Test Requirement**: A test requirement is a specific element of a software artifact that a test case must satisfy or cover.

- Ice cream cone flavors: vanilla, chocolate, mint
- One test requirement: test one chocolate cone
- **TR** denotes a set of test requirements

A **coverage criterion** is a rule or collection of rules that impose test requirements on a test set.

- Coverage criterion is a recipe for generating TR in a systematic way.
  - Flavor criterion [cover all flavors]
  - **TR** = {flavor=chocolate, flavor=vanilla, flavor=mint}
Adequacy criteria

Adequacy criterion = set of test requirements

A test suite satisfies an adequacy criterion if

• all the tests succeed (pass)
• every test requirement in the criterion is satisfied by at least one of the test cases in the test suite.

Example:

the statement coverage adequacy criterion is satisfied by test suite S for program P if each executable statement in P is executed by at least one test case in S, and the outcome of each test execution was “pass”
Adequacy Criteria as Design Rules

Many design disciplines employ design rules

- e.g.: “traces (on a chip, on a circuit board) must be at least ___ wide and separated by at least ___”
- “Interstate highways must not have a grade greater than 6% without special review and approval”

Design rules do not guarantee good designs

- Good design depends on talented, creative, disciplined designers; design rules help them avoid or spot flaws

Test design is no different
Where do test requirements come from?

Functional (black box, specification-based): from software specifications
  • Example: If spec requires robust recovery from power failure, test requirements should include simulated power failure

Structural (white or glass box): from code
  • Example: Traverse each program loop one or more times.

Model-based: from model of system
  • Models used in specification or design, or derived from code
  • Example: Exercise all transitions in communication protocol model

Fault-based: from hypothesized faults (common bugs)
  • Example: Check for buffer overflow handling (common vulnerability) by testing on very large inputs
Code Coverage

Introduced by Miller and Maloney in 1963
Coverage Criteria

Basic Coverage
- Line coverage
- Statement
- Function/Method coverage
- Branch coverage
- Decision coverage
- Condition coverage
- Condition/decision coverage
- Modified condition/decision coverage
- Path coverage
- Loop coverage
- Mutation adequacy
- ...

Advanced Coverage
Line Coverage

Percentage of source code lines executed by test cases.

• For developer easiest to work with

• Precise percentage depends on layout?
  – int x = 10; if (z++ < x) y = x+z;

• Requires mapping back from binary?

In practice, coverage not based on lines, but on control flow graph
Control Flow Graph (CFG)

Represents the flow of execution in the program

\[ G = (N, E, S, T) \]

- the nodes \( N \) represent executable instructions (statement, statement fragments, or basic blocks);
- the edges \( E \) represent the potential transfer of control;
- \( S \) is a designated start node;
- \( T \) is a designated final node
- \( E = \{ (n_i, n_j) \mid \text{syntactically, the execution of } n_j \text{ follows the execution of } n_i \} \)

Nodes may correspond to single statements, parts of statements, or several statements (i.e., basic blocks)

Execution of a node means that the instructions associated with a node are executed in order from the first instruction to the last.
Example of a Control Flow Graph

total := 0;
count := 1;
max := input();
while (count <= max)
do {
  val := input();
  total := total + val;
  count := count + 1};
print (total)
Control Flow Graph

A CFG is a graph of basic blocks
  - edges represent different control flow

A CFG corresponds to a program syntax
  - where statements are restricted to the form $L_i : S ; \text{goto } L_j$
  - and $S$ is control-free (i.e., assignments and procedure calls)

```
total := 0;
count := 1
max := input()
val := read()
total := total + val
count := count + 1
print(total)
```

$\text{T}$
$\text{F}$
Control Flow Graph as a Goto Program

1: total := 0; count := 1;
   max = input(); goto 2

2: if count <= max
   then goto 3 else goto 4

3: val := read();
   total := total + val;
   count := count + 1; goto 2

4: print(total)
Splitting multiple conditions depends on goal of analysis
Infeasible Paths

Every executable sequence in the represented component corresponds to a path in G

Not all paths correspond to executable sequences

- requires additional semantic information
- “infeasible paths” are not an indication of a fault

CFG usually overestimates the executable behavior
Statement or Node Coverage

Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```c
void foo (int z) {
    int x = 10;
    if (z++ < x) {
        x+= z;
    }
}
```

Coverage:

```
# executed statements
# statements
```
Statement or Node Coverage

Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```java
void foo (int z) {
    int x = 10;
    if (z++ < x) {
        x+= z;
    }
}

@Test
void testFoo() {
    foo(10);
}
```

Coverage:

# executed statements
# statements
Statement or Node Coverage

Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```java
void foo (int z) {
    int x = 10;
    if (z++ < x) {
        x=+ z;
    }
}
```

Coverage:

```
# executed statements
# statements
```
Statement or Node Coverage

Adequacy criterion: each statement (or node in the CFG) must be executed at least once

```java
void foo (int z) {
    int x = 10;
    if (z++ < x) {
        x=+ z;
    }
}
```

Coverage Level:

```
# executed statements
# statements
```

```java
@Test
void testFoo() {
    foo(5);
}
// 100% Statement coverage
```
Control Flow Based Adequacy Criteria

Every block / Statement?

Input: “a”
Trace: b2,b3,b4,b5,b6,b7,b3,b8
Branch / Edge Coverage

Every branch going out of node executed at least once

- Decision-, all-edges-, coverage
- Coverage: percentage of edges hit.

Each branch predicate must be both true and false
Branch Coverage

One longer input:
“a\n\n”

Alternatively:
Block (“a”) and
“\n” and
“\n\n”
Infeasible Test Requirements

Real code from the Linux kernel:

```c
if (false)
    unreachableCall();

while (0)
    {local_irq_disable();}
```

Statement coverage criterion cannot be satisfied for many programs.
Coverage Level

Given a set of test requirements $\text{TR}$ and a test set $T$, the coverage level is the ratio of the number of test requirements satisfied by $T$ to the size of $\text{TR}$.

$$\text{TR} = \{\text{flavor=chocolate, flavor=vanilla, flavor=mint}\}$$
$$\text{Test set 1 } T_1 = \{3 \text{ chocolate cones, 1 vanilla cone}\}$$
$$\text{Coverage Level} = \frac{2}{3} = 66.7\%$$

Coverage levels help evaluate the goodness of a test set, especially in the presence of infeasible test requirements.
Unit Testing

A *unit test* exercises a unit of functionality to test its behavior.

A *unit test framework* provides a standard mechanism for:
- specifying a test (setup, execution, expected result, teardown)
- executing a test
- generating test reports

Python includes a Unit Test framework called *unittest*:
- [https://docs.python.org/2/library/unittest.html](https://docs.python.org/2/library/unittest.html)

It is important to design your code with testing in mind:
- e.g., a code that simply reads and writes to standard input and output is harder to test than code that provides a more structured interaction.
Anatomy of a Unit Test

```python
import unittest

class TestStringMethods(unittest.TestCase):
    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

- **include module**
- A test case is a collection of tests
- A method is a test
- Calls to `assertXXX()` methods indicate test results
- Entry point for the test when ran from command line
Designing for Testing

Factor the program into meaningful units / components
- e.g., parser, command processor, components, data structures, etc.

Each unit should have a well defined specification
- what are legal inputs
- what are legal outputs
- how inputs and outputs are passed around

Avoid monolithic design that reads standard input and writes standard output

Good design requires more work
- additional functionality specifically for testing / debugging purposes
- but ultimately will save time of the overall development
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

Subsumption is a rough guide for comparing criteria, although it’s hard to use in practice.
More powerful coverage criterion helps find more bugs!

```c
int d[2];

N1: if (x >= 0 && x < 2)
    { N2: print (x); }
N3: if (y > 0)
    { N4: print (d[x] + y); }
N5: exit (0);
```

Path [N1, N2, N3, N4, N5]:
  satisfies node coverage but not edge coverage.
  The corresponding test case passes. No bug found.

Path [N1, N3, N4, N5]: buffer overflow bug!
Path Coverage

Adequacy criterion: each path must be executed at least once

Coverage:
\[
\frac{\text{# executed paths}}{\text{# paths}}
\]
Path-based criteria?

All paths?

Which paths?

```java
public static String collapseNewlines(String argStr) {
    char last = argStr.charAt(0);
    StringBuffer argBuf = new StringBuffer();
    for (int cldx = 0; cldx < argStr.length();)
        if (last != 'n')
            if (cldx != 'n')
                argBuf.append(last);
                last = ch;
        cldx++;
    return argBuf.toString();
}
```
Branch vs Path Coverage

if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();

How many test cases to achieve branch coverage?
Branch vs Path Coverage

if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();

How many test cases to achieve branch coverage?

Two, for example:

1. cond1: true, cond2: true
2. cond1: false, cond2: false
Branch vs Path Coverage

if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();

How about path coverage?
Branch vs Path Coverage

```c
if( cond1 )
    f1();
else
    f2();

if( cond2 )
    f3();
else
    f4();
```

How about path coverage?

Four:

1. cond1: true, cond2: true
2. cond1: false, cond2: true
3. cond1: true, cond2: false
4. cond1: false, cond2: false
Branch vs Path Coverage

if( cond1 )
  f1();
else
  f2();
if( cond2 )
  f3();
else
  f4();
if( cond3 )
  f5();
else
  f6();
if( cond4 )
  f7();
else
  f8();
if( cond5 )
  f9();
else
  f10();
if( cond6 )
  f11();
else
  f12();
if( cond7 )
  f13();
else
  f14();

How many test cases for path coverage?

$2^n$ test cases, where $n$ is the number of conditions
A **test path** is a path $p$ [possibly of length 0] that starts at some node in $N_0$ and ends at some node in $N_f$.

Test path examples:

- $[1, 2, 3, 5, 6, 7]$
- $[1, 2, 3, 5, 6, 2, 3, 5, 6, 7]$
Some paths in a control flow graph may not correspond to program semantics.

In path coverage, we generally only talk about the syntax of a graph -- its nodes and edges -- and not its semantics.
Syntactical and Semantic Reachability

A node $n$ is *syntactically* reachable from $m$ if there exists a path from $m$ to $n$.

A node $n$ is *semantically* reachable if one of the paths from $m$ to $n$ can be reached on some input.

Standard graph algorithms when applied to Control Flow Graph can only compute *syntactic reachability*.

*Semantic reachability* is undecidable.
Reachability

Let $\text{reach}_G(X)$ denote the sub-graph of $G$ that is (syntactically) reachable from $X$, where $X$ is either a node, an edge, a set of nodes, or a set of edges.

In this example, $\text{reach}_G(1)$ is the whole graph $G$. 
**Syntactical Reachability**

- \( \text{reach}_{G\#}(2) \) is the subgraph that is syntactically reachable from node 2.

- \( \text{reach}_{G\#}(7) \) is: 7
Connect Test Cases and Test Paths

Connect test cases and test paths with a mapping $path_G$ from test cases to test paths

- e.g., $path_G[t]$ is the set of test paths corresponding to test case $t$.

- Usually just write $path$, as $G$ is obvious from the context.

- Lift the definition of path to test set $T$ by defining $path(T)$

\[
path(T) = \{path(t) | t \in T\}.
\]

- Each test case gives at least one test path. If the software is deterministic, then each test case gives exactly one test path; otherwise, multiple test cases may arise from one test path.
Connecting Test Cases, Test Paths, and CFG

```c
int foo(int x) {
    if (x < 5) {
        x ++;
    } else {
        x --;
    }
    return x;
}
```

- Test case: \( x = 5 \); test path: \([1, 3, 4]\).
- Test case: \( x = 2 \); test path: \([1, 2, 4]\).
Node Coverage

Node coverage: For each node $n \in \text{reachG}[N_0]$, TR contains a requirement to visit node $n$.

Node Coverage [NC]: TR contains each reachable node in $G$.

TR = \{n0, n1, n2, n3, n4, n5, n6\}

a.k.a. statement coverage
Edge Coverage (a.k.a. Branch Coverage)

*Edge Coverage [EC]:* TR contains each *reachable* path of length up to 1, inclusive, in G.

$$TR = \{[1,2], [2,4], [2,3], [3,5], [4,5], [5,6], [6,7], [6,2]\}$$
Edge Pair Coverage [EPC]: TR contains each reachable path of length up to 2, inclusive, in G.

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

A path is **simple** if no node appears more than once in the path, except that the first and last nodes may be the same.

Some properties of simple paths:

- no internal loops;
- can bound their length;
- can create any path by composing simple paths; and
- many simple paths exist [too many!]
Simple Path Examples

Simple path examples:

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

Not simple Path: [1,2,3,5,6,2,4]
Prime Path

Because there are so many simple paths, let’s instead consider prime paths, which are simple paths of maximal length.

A path is prime if it is simple and does not appear as a proper subpath of any other simple path.
Prime Path Examples

Prime path examples:

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

Not a prime path: [3, 5, 6, 7]
Prime Path Coverage

Prime Path Coverage [PPC]: TR contains each prime path in G.

There is a problem with using PPC as a coverage criterion: a prime path may be infeasible but contains feasible simple paths.

• How to address this issue?
More Path Coverage Criteria

Complete Path Coverage [CPC]: TR contains all paths in G.

Specified Path Coverage [SPC]: TR contains a specified set S of paths.
Prime Path Example

Simple paths

<table>
<thead>
<tr>
<th>Len 0</th>
<th>Len 1</th>
<th>Len 2</th>
<th>Len 3</th>
<th>Len 4</th>
<th>Len 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>[1,2]</td>
<td>[2,3]</td>
<td>[3,5]</td>
<td>[4,5]</td>
<td>[5,6]</td>
</tr>
<tr>
<td>[2]</td>
<td>[2,4]</td>
<td>[2,3]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td>[3,5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4]</td>
<td>[4,5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[5]</td>
<td>[4,5]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6]</td>
<td>[5,6]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[7]!</td>
<td>[6,2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

53 Simple Paths
12 Prime Paths

! means path terminates
Prime Path Example

Simple paths

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

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

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

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

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

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

Check paths without a x or *:

12 Prime Paths

53 Simple Paths

! means path terminates.
x means not prime paths.
* denotes path cycles.
Prime Path Example (2)

This graph has 38 simple paths
Only 9 prime paths

Prime Paths

<table>
<thead>
<tr>
<th>Prime Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ 0, 1, 2, 3, 6 ]</td>
</tr>
<tr>
<td>[ 0, 1, 2, 4, 5 ]</td>
</tr>
<tr>
<td>[ 0, 1, 2, 4, 6 ]</td>
</tr>
<tr>
<td>[ 0, 2, 3, 6 ]</td>
</tr>
<tr>
<td>[ 0, 2, 4, 5 ]</td>
</tr>
<tr>
<td>[ 0, 2, 4, 6 ]</td>
</tr>
<tr>
<td>[ 5, 4, 6 ]</td>
</tr>
<tr>
<td>[ 4, 5, 4 ]</td>
</tr>
<tr>
<td>[ 5, 4, 5 ]</td>
</tr>
</tbody>
</table>
Prime Path Example (2)

Simple paths

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

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

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

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

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

! means path terminates

* denotes path cycles

9 Prime Paths
Examples of NC, EC, EPC, CPC

**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, 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 ] ...
Prime Path Coverage vs. Complete Path Coverage

- Prime paths:
  - $\text{path}(t_1) =$
  - $\text{path}(t_2) =$
  - $T_1 = \{t_1, t_2\}$ satisfies both PPC and CPC.
Prime Path Coverage vs. Complete Path Coverage

- Prime paths: $[n0, n1, n3], [n0, n2, n3]$
- $\text{path}(t_1) =$ $[n0, n1, n3]$
- $\text{path}(t_2) =$ $[n0, n2, n3]$
- $T_1 = \{t_1, t_2\}$ satisfies both PPC and CPC.
Prime Path Coverage vs. Complete Path Coverage (2)

- Prime paths:
- \( \text{path}(t_3) = \)  
- \( \text{path}(t_4) = \)  
- \( T_1 = \{t_3, t_4\} \) satisfies PPC but not CPC.
Prime Path Coverage vs. Complete Path Coverage (2)

- Prime paths:
  - \([q_0, q_1, q_2]\), \([q_0, q_1, q_3, q_4]\), \([q_3, q_4, q_1, q_2]\), \([q_1, q_3, q_4, q_1]\), \([q_3, q_4, q_1, q_3]\), \([q_4, q_1, q_3, q_4]\)  
  - \([q_0, q_1, q_2]\)

- \(\text{path}(t_3) = \) 
- \(\text{path}(t_4) = \)  
  - \([q_0, q_1, q_3, q_4, q_1, q_3, q_4, q_1, q_2]\)  
  - \(T_1 = \{t_3, t_4\}\) satisfies PPC but not CPC.
Graph Coverage Criteria Subsumption

Complete Path Coverage
- CPC

Prime Path Coverage
- PPC

Edge-Pair Coverage
- EPC

Edge Coverage
- EC

Node Coverage
- NC
How do we measure coverage?

First:
1. Parse the source code to build an Abstract Syntax Tree (AST)
2. Analyze the AST to build a Control Flow Graph (CFG)
3. Count points of interest
   - (total # of statements, branches, etc.)
4. Instrument the AST using the CFG
   - add tracing statements in the code
How do we measure coverage?

*Then:*

1. Transform AST back to instrumented code
2. Recompile and run the test suite on the recompiled code
3. Collect tracing data
   - (line 1 executed, line 3 executed, etc.)
4. Calculate coverage:
   - # traced points / total # points
Coverage May Affect Test Outcomes

*Heisenberg effect*

- *the act of observing a system inevitably alters its state.*

Coverage analysis changes the code by adding tracing statements

Instrumentation can change program behaviour
### Enabled In-code Assertions Mess Up Branch Coverage Reporting

**assert** P

Turned into:

```c
if assertion-enabled then
    if P then skip()
    else abort()
else skip()
```

Thus 4 branches!

Reported as such

**Assertions shouldn’t fail**

Resulting branch coverage reports:

- Not useful with assertion checking enabled
- Without it, they miss invariants
Coverage: Useful or Harmful?

Measuring coverage (% of satisfied test obligations) can be a useful indicator ... 
• Of progress toward a thorough test suite, of trouble spots requiring more attention

... or a dangerous seduction
• Coverage is only a proxy for thoroughness or adequacy
• It’s easy to improve coverage without improving a test suite (much easier than designing good test cases)

The only measure that really matters is **effectiveness**
Exercise 1: Bridge Coverage

Bridge Coverage (BC): If removing an edge adds unreachable nodes to the graph, then this edge is a bridge. The set of test requirements for BC contains all bridges. Assume that a graph contains at least two nodes, and all nodes in a graph are reachable from the initial nodes.

(a) Does BC subsume Node Coverage (NC). If yes, justify your answer. If no, give a counterexample.

(b) Does NC subsume BC? If yes, justify your answer. If no, give a counterexample
Bridge Coverage: Part (a)

Bridge Coverage does not subsume Node Coverage.

\[ TR_{BC} = \{[1,2], [2,3], [3,5]\} \]
\[ TR_{NC} = \{[1, 2, 3,4, 5}\] 
Test path [1,2,3,5] satisfies BC, but not NC because node 4 is not visited.
Bridge Coverage: Part (b)

NC subsumes BC

Key points for the proof:

• For any bridge \([a, b]\), any test case that visits \(b\) must also visit the edge \([a, b]\) (can be proved by contradiction).

• Any test set that satisfies NC must visit node \(b\) (TR of NC contains all nodes, including node \(b\)). Therefore, for any bridge \([a, b]\), the test set will visit it. Therefore, NC subsumes BC.
Exercise 2 (1/2)

Answer questions [a]-[g] for the graph defined by the following sets:

- $N = \{1, 2, 3, 4, 5, 6, 7\}$
- $N_0 = \{1\}$
- $N_f = \{7\}$
- $E = \{[1, 2], [1, 7], [2, 3], [2, 4], [3, 2], [4, 5], [4, 6], [5, 6], [6, 1]\}$

Also consider the following test paths:

- $t_0 = [1, 2, 4, 5, 6, 1, 7]$
- $t_1 = [1, 2, 3, 2, 4, 6, 1, 7]$
Exercise 2 (2/2)

[a] Draw the graph.

[b] List the test requirements for EPC. [Hint: You should get 12 requirements of length 2].

[c] Does the given set of test paths satisfy EPC? If not, identify what is missing.

[d] List the test requirements for NC, EC and PPC on the graph.

[e] List a test path that achieve NC but not EC on the graph.

[f] List a test path that achieve EC but not PPC on the graph.
Exercise 2: Partial Solutions (1/2)

[a] Draw the graph.

[b] List the test requirements for EPC. [Hint: You should get 12 requirements of length 2].

- The edge pairs are: \{[1, 2, 3], [1, 2, 4], [2, 3, 2], [2, 4, 5], [2, 4, 6], [3, 2, 3], [3, 2, 4], [4, 5, 6], [4, 6, 1], [5, 6, 1], [6, 1, 2], [6, 1, 7] \}

[c] Does the given set of test paths satisfy EPC? If not, identify what is missing.

- No. Neither t0 nor t1 visits the following edge-pairs: \{[3, 2, 3], [6, 1, 2] \}
Exercise 2: Partial Solutions (2/2)

[d] TR for NC, EC, and PPC.

• \( TR_{NC} = \)

• \( TR_{EC} = \)

• \( TR_{PPC} = \{[3, 2, 4, 6, 1, 7], [3, 2, 4, 5, 6, 1, 7], [4, 6, 1, 2, 3], [4, 5, 6, 1, 2, 3], [3, 2, 3], [2, 3, 2], [1, 2, 4, 5, 6, 1], [1, 2, 4, 6, 1], [2, 4, 6, 1, 2], [2, 4, 5, 6, 1, 2], [4, 6, 1, 2, 4], [4, 5, 6, 1, 2, 4], [5, 6, 1, 2, 4, 5], [6, 1, 2, 4, 6], [6, 1, 2, 4, 5, 6]\} \)

[e] A test path that achieve NC but not EC.

• \([1, 2, 3, 2, 4, 5, 6, 1, 7]\) does not cover edge \([4, 6]\).  

[f] A test path that achieve EC but not PPC.

• \([1, 2, 3, 2, 4, 5, 6, 1, 2, 4, 6, 1, 7]\) does not cover prime paths such as \([3,2,3]\).