50.003 Elements of Software Construction

Code-based testing - Path testing

Learning Outcomes

By the end of this lesson you should be able to

1. Construct a control flow graph from a structured program
2. Explain different types of test coverage metrics based on program graph
3. Apply path testing techniques to generate test case to attain path coverage and MCDC coverage.

Recap

- Specification based testing
  - Test cases are derived from the specification
  - The goal is to ________________________
- Code-based testing
  - Test cases are defined by making use of the knowledge of the internal structure and algorithm used in the test subject.
  - The goal is to ________________________

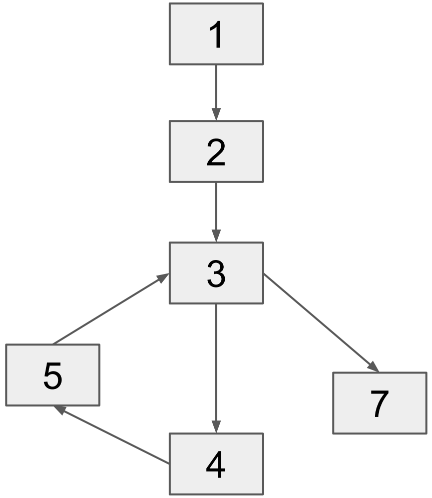
Code based testing

To find faults, we must exhaustively execute all possible sequence of statements in the code.

Characterizing the statement sequence

To find the set of statement sequence, we could view the source code in a graph form by following its control flow (order of execution).

```
// example A
function sum_all(arr) {
  let sum = 0;
  let i = 0;
  while (i < arr.length) {
    sum = arr[i] + sum;
    i++;
  }
  return sum;
}
```



Constructing Control Flow graph

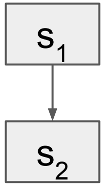
For simplicity, we consider only structured program

- Functions and loops are single entry and single exit
- If-then is rewritten to if-then-else with an empty alternative
- For-loop is rewritten into while loop

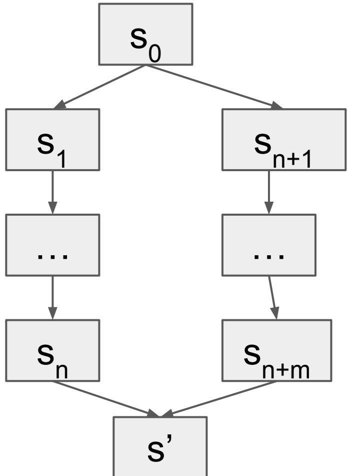
Constructing Control Flow graph

Walk through the code from top to bottom

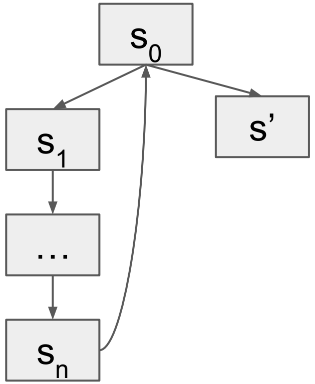
```
stmt1;
stmt2;
```



```
if cond {
   stmt1;...;stmtn;
} else {
   stmtn+1;...;stmtn+m;
}
// the following stmt'
```



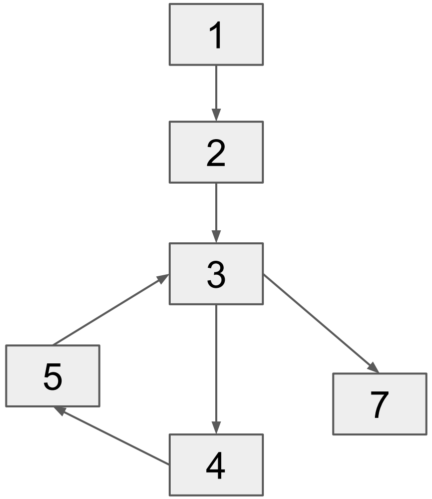
```
while cond {
    stmt1;...;stmtn;
}
// the following stmt'
```



Path

A path is a sequence of nodes connected by edges.

- 1
- 1,2,3
- 1,2,3,7
- 1,2,3,4,5,3,7
- 2,3
- ...

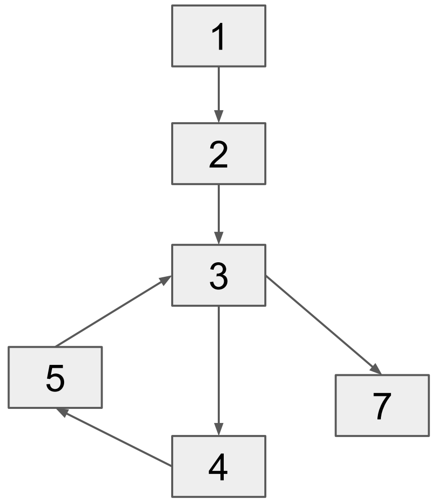


Test Coverage Metrics

- Node coverage
  - if all nodes in the control flow graph are visited when the tests are executed.
- Edge coverage
  - if all the edges in the control flow graph are visited when the tests are executed.
- Condition coverage
  - ~~edge coverage plus~~ all conditional predicates are evaluated to both true and false.
  - Conditional predicates refer to those atomic predicate without conjunction && , disjunction || and negation operations !,
- Path coverage
  - if all the paths in the control flow graph are visited when the tests are executed.
- MCDC coverage
  - TBD

Node coverage

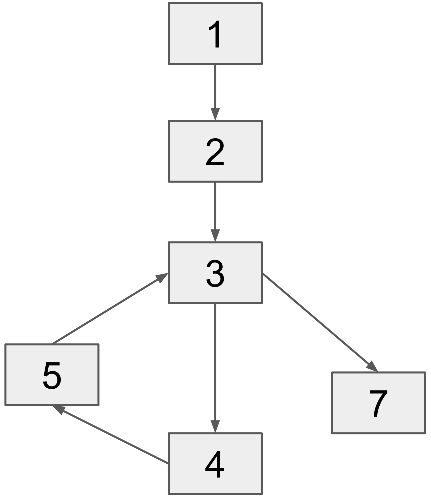
id	arr	expected
1	[1,2]	3


$$\begin{tikzcd} & 1 \arrow[d] & \\ & 2 \arrow[d] & \\ & 3 \arrow[d] \arrow[dr] & \\ 5 \arrow[ur] & 4 \arrow[l] & 7 \end{tikzcd}$$

```
// example A
function sum_all(arr) {
  let sum = 0;
  let i = 0;
  while (i < arr.length) {
    sum = arr[i] + sum;
    i++;
  }
  return sum;
}
```

Edge coverage

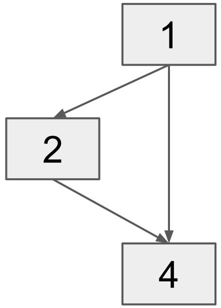
id	arr	expected
1	[1,2]	3



$$\begin{tikzcd}
 & 1 \arrow[d] & \\
 & 2 \arrow[d] & \\
 & 3 \arrow[d] \arrow[dr] & \\
5 \arrow[ur] & & 7 \\
 & 4 \arrow[ul] &
\end{tikzcd}$$

```
// example A
function sum_all(arr) {
  let sum = 0;
  let i = 0;
  while (i < arr.length) {
    sum = arr[i] + sum;
    i++;
  }
  return sum;
}
```

Node Coverage vs Edge coverage

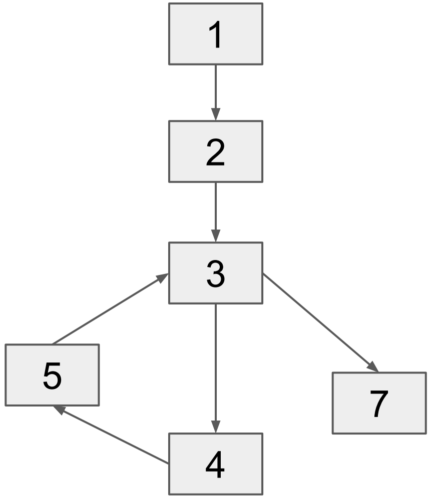


```
if (x > 0) {
   y = 1;
} else { }
return y;
```

When x=1, we cover all the nodes, but not all the edges, ie. the edges for the else-branch are not visited.

Condition Coverage

id	arr	expected
1	[1,2]	3


$$\begin{tikzcd}
 & 1 \arrow[d] & \\
 & 2 \arrow[d] & \\
 & 3 \arrow[d] \arrow[dr] & \\
5 \arrow[ur] & 4 \arrow[l] & 7
\end{tikzcd}$$

```
// example A
function sum_all(arr) {
  let sum = 0;
  let i = 0;
  while (i < arr.length) {
    sum = arr[i] + sum;
    i++;
  }
  return sum;
}
```

Condition Coverage vs Edge Coverage

Consider

```
function div_by_zero(x,y) {
  if ((x==0) || (y>0)) {
    y = y/x;
  } else {
    x = y++;
  }
  return x + y;
}
```

id	x	y	expected
1	5	-5	-9
2	7	5	7.714

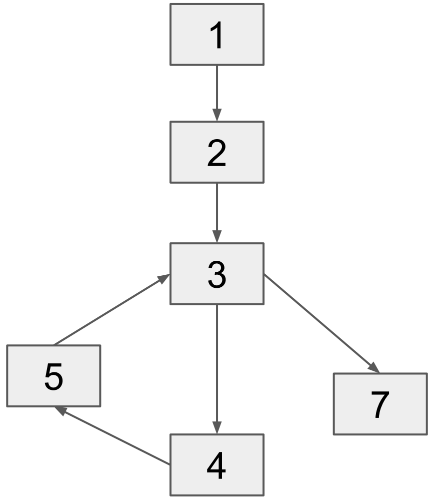
We have Edge coverage, but a bug is still lurking.

id	x	y	expected
1	5	-5	-9
2	7	5	7.714
3	0	1	error

Path Coverage

- With the presence of loops, there could be infinitely many paths in a program graph
- Traverse the loop once and merge the loop body into a node.
-

id	arr	expected
1	[1,2]	3

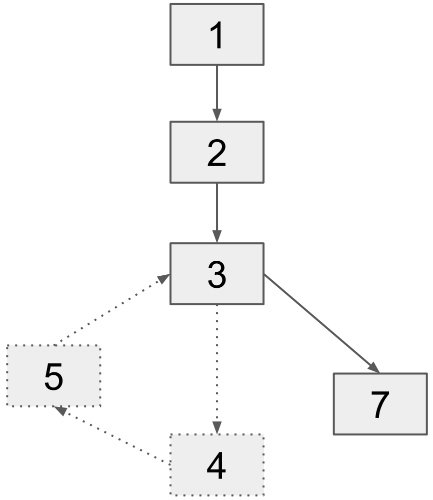


```
// example A
function sum_all(arr) {
  let sum = 0;
  let i = 0;
  while (i < arr.length) {
    sum = arr[i] + sum;
    i++;
  }
  return sum;
}
```

Path Coverage

- With the presence of loops, there could be infinitely many paths in a program graph
- Traverse the loop once and merge the loop body into a node.

id	arr	expected
1	[1,2]	3
2	[]	0



```
// example A
function sum_all(arr) {
  let sum = 0;
  let i = 0;
  while (i < arr.length) {
    sum = arr[i] + sum;
    i++;
  }
  return sum;
}
```

2 paths from entry to exit

- 1,2,3,7
- 1,2,3,4,5,3,7

Condition Coverage vs Path Coverage

Is path coverage always better?

```
if ((x>0) || (x <-1)) {
  res = 1;
} else {
  res = 0;
}
```

id	x	res
1	1	1
2	0	0

We have edge and path coverage, but not condition coverage.

But not all paths are accessible

We can’t access the then-branch

```
if ((x>0) && (x <-1)) {
  res = 1;
} else {
  res = 0;
}
```

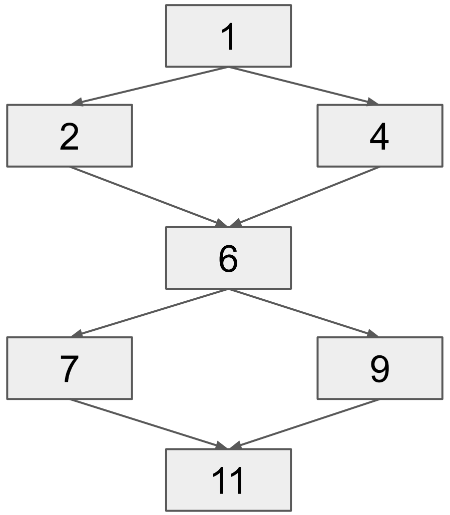
id	x	res
1	1	0
2	-2	0

We have condition coverage, we don’t have edge coverage, thus no path coverage

Condition Coverage vs Path Coverage

Condition coverage + edge coverage => path coverage?

```
function f(x) {
  if (x > 1) {
    y = 1;
  } else {
    y = 0;
  }
  if (x > 0) {
    z = 1;
  } else {
    z = 0;
  }
  return y * z;
}
```



id	x	expected
1	2	1
2	-1	0

We have condition and edge coverage, but not path coverage, e.g. the path 1,2,6,9,11 is not covered.

MCDC Coverage

- It stands for Modified Condition Decision Coverage.
- It requires
  a. Each statement must be executed at least once. (Node coverage)
  b. Every program entry point and exit point must be invoked at least once. (Node coverage)
  c. All possible outcomes of every control statement are taken at least once. (Edge Coverage)
  d. Every nonconstant Boolean expression has been evaluated to both true and false outcomes. A boolean expression is constant iff it is a tautology, e.g. a==a, a!=a and p && !p (Condition Coverage)
  e. Every nonconstant condition in a Boolean expression has been evaluated to both true and false outcomes. (Top-level only, Path coverage)
     - Otherwise, some branch must be dead.
  f. Every nonconstant condition in a Boolean expression has been shown to ***independently affect the outcomes*** (of the expression).
     - Otherwise, some branch must be redundant

MCDC Coverage

- Construct the decision table
- Toggle each sub condition while fixing the rest of the sub conditions

```
if (x && y) {
  res = 1;
} else {
  res = 0;
}
```

id	x	y	res
1	true	true	1
2	false	true	0
3	true	false	0

cond	**1**	**2**	**3**	**4**
x	T	T	F	F
y	T	F	T	F
x && y	T	F	F	F
actions				
res = 1	X			
res = 0		X	X	X

MCDC Coverage

- Construct the decision table
- Toggle each sub condition while fixing the rest of the sub conditions

```
if ((x>0) && (x <-1)) {
  res = 1;
} else {
  res = 0;
}
```

MCDC impossible.

id	x	res
1	1	0
2	-2	0

cond	1	2	3
x>0	T	F	F
x<-1	F	T	F
x >0 && x<-1	F	F	F
actions			
res = 1			
res = 0	X	X	X

Path testing

To conduct path testing, a general approach is

- to generate enough test cases cover all paths with condensation (if possible).
- in case of complex condition expression (esp with dependent condition expressions), we should construct a decision table to check impossibilities and generate additional test cases to ensure condition coverage.

How many unique paths

Cyclomatic complexity

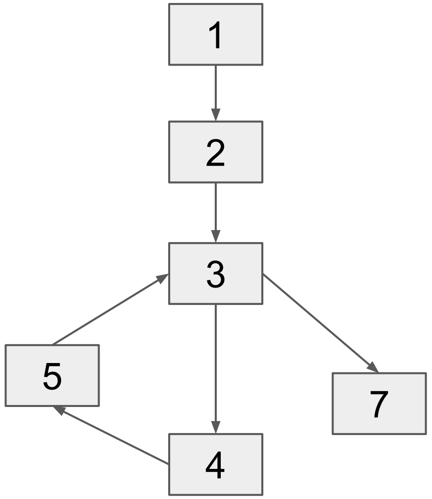
V(G) = e - n + 2p

where

- e is the number of edges,
- n is the number of nodes, and
- p is the number of ~~strongly~~ connected components, which is usually 1, given all the statements are linked in a function.

id	arr	expected
1	[1,2]	3
2	[]	0

```
function sum_all(arr) {
  let sum = 0;
  let i = 0;
  while (i < arr.length) {
    sum = arr[i] + sum;
    i++;
  }
  return sum;
}
```

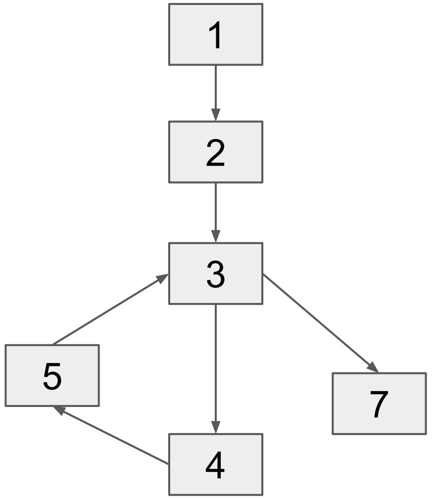


6- 6 + 2*1 = 2 unique linearly paths from entry to exit.

Checking out the condition

- i < arr.length is a single predicate.
- It's not a constant.
- It's not dependent.
- There is no impossible path.

```
function sum_all(arr) {
  let sum = 0;
  let i = 0;
  while (i < arr.length) {
    sum = arr[i] + sum;
    i++;
  }
  return sum;
}
```



cond	1	2
i < arr.length	T	F
actions		
Enter loop	X	
Exit loop		X

Jest Coverage Report

To enable, edit package.json

```
"jest": {
  "collectCoverage": true
  ,"coverageReporters": ["text", "html"]
},
```

Given

```
const sum_all = require('../src/sum_all');
  describe("test suite for sum_all", () => {
    test ("test 1 for sum_all", () => {
    const expected = 55;
    expect(sum_all([1,2,3,4,5,6,7,8,9,10])).toBe(expected);
  })
}
```

Run npm run test sum_all.test.js

```
PASS test/sum_all.test.js
test suite for sum_all
✓ test 1 for sum_all (3 ms)

------------|---------|----------|---------|---------|-------------------
File | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
------------|---------|----------|---------|---------|-------------------
All files | 100 | 100 | 100 | 100 |
sum_all.js | 100 | 100 | 100 | 100 |
------------|---------|----------|---------|---------|-------------------
```

Note that Jest is only checking node and edge coverages.

Cohort Exercise (Graded)

1. construct CFG of the given program.
2. find the cyclomatic complexity of the CFG.
3. generate the test cases to cover all paths.
4. generate extra test cases (if needed) to attain MCDC coverage.
5. verify your test cases's coverage using jest (though it is only up to edge coverage)

```
function gcd(x,y) {
  let r = null;
  if ((x < 1) || (y < 1)) {
    r = null;
  } else {
    while (x != y) {
      if (x > y) {
        let t = x - y;
        x = y;
        y = t
      } else {
        let t = y - x;
        y = x;
        x = t;
      }
    }
    r = x;
  }
  return r;
}
```