Graph Coverage for Software Testing

UPULEE KANEWALA

COMPUTER SCIENCE DEPARTMENT, COLORADO STATE UNIVERSITY, USA

Today's lecture…

- Black-box and white-box testing
- Control flow graphs
- Node coverage
- Edge coverage
- Coverage tool demonstration

How would you test this program?

Specification: *computeAverage* method finds the average of a list of numbers

Input: integer array

Output: double value

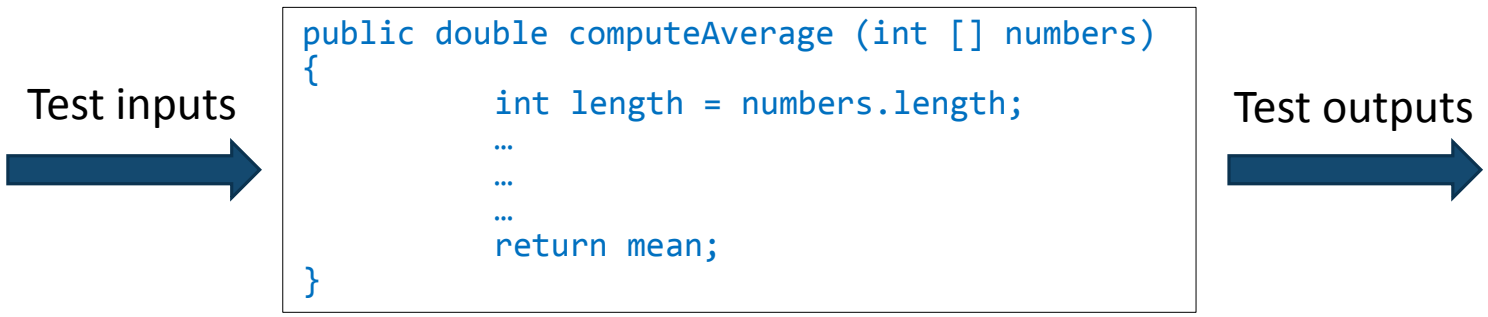
Black-box testing

- Treat the program as a black box
- Test inputs are created based on specification of the program
- Correctness of the outputs are checked using the specification

- Problem: **no guarantee that at least all the statements were executed**
  - Black-box tests usually cover around 65% - 85% of the code
  - 95% of errors are in 5% of the code

White-box testing

- Goal is to exercise different programming structures and data structures in the program
- Create test inputs to “cover” different structures in the program
- Correctness of the outputs are checked using the specification
- Need access to the source code



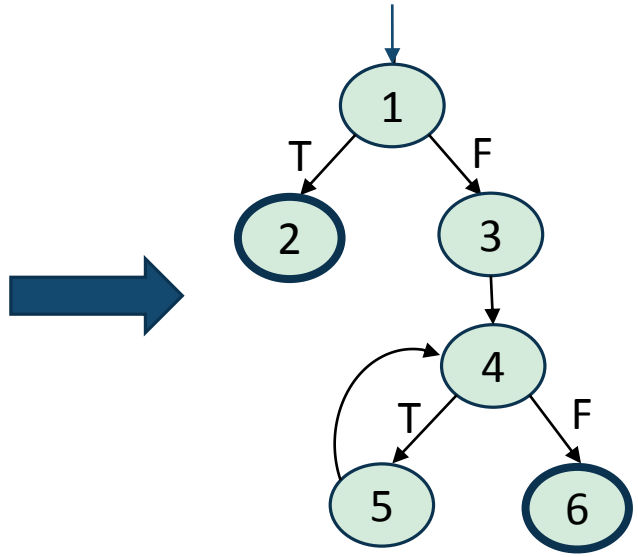
Applying white-box testing

- Common approach:
  - Represent the program as a graph
  - Create test cases to visit elements in the graph such as nodes or edges
- Most commonly used graph representation is called a ***control flow graph***

Control flow graph (CFG)

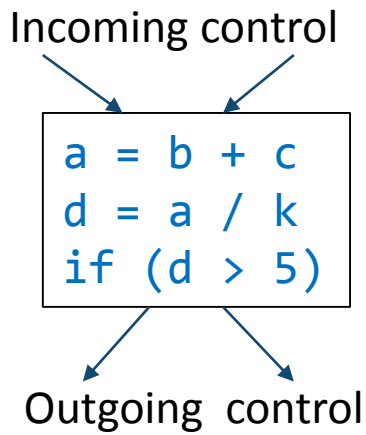
- A graph that represents all executions of a method by describing control structures
- Nodes – sequence of statements (a basic block)
  - Start node: denoted with an incoming arrow
  - Exit node: denoted with a thick border
- Edges – transfer of control

```
public double computeAverage (int [] numbers)
{
            int length = numbers.length;
            double mean=0;
            double sum = 0;
            if (length <= 0)
            {
                  return Double.NaN;
            }
            for (int i = 0; i < length; i++)
            {
                   sum += numbers [i];
            }
            mean = sum / (double) length;
            return mean;
}
```



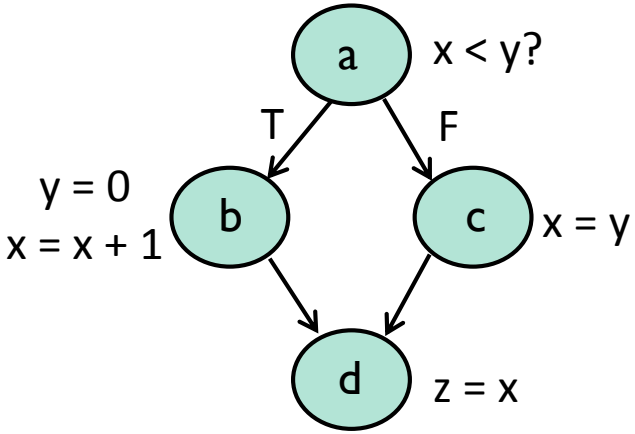
Basic blocks

- Sequence of consecutive statements such that:
  - Control enters only at the beginning of the sequence
  - Control leaves only at the end of the sequence
- No branching in or out from the middle of the basic blocks

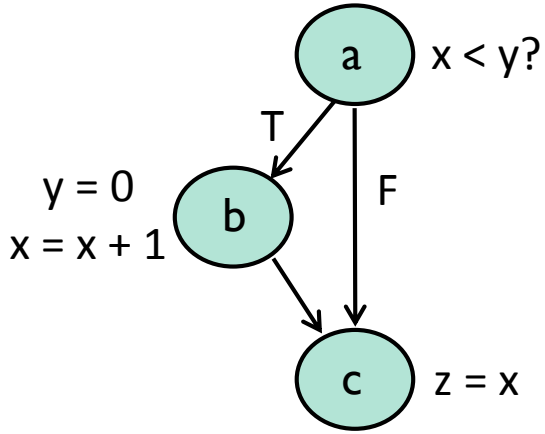


Constructing CFGs: *if* statements

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

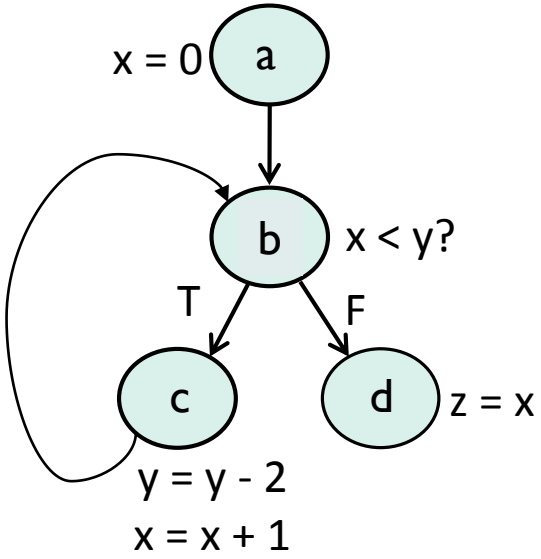



```
if (x < y)
{
   y = 0;
   x = x + 1;
}
z = x
```



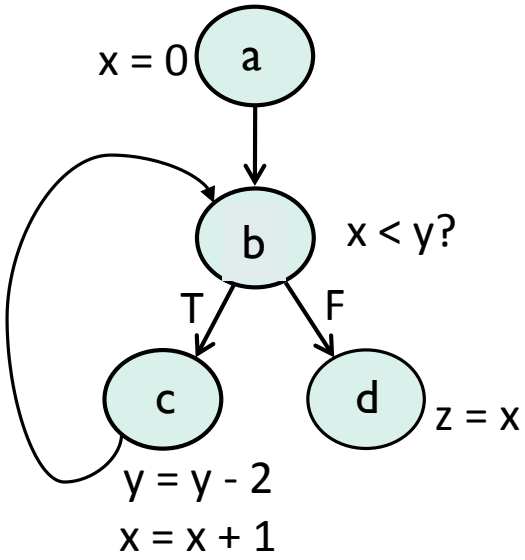
Constructing CFGs: *while* loops

```
x = 0;
while (x < y)
{
   y = y - 2;
   x = x + 1;
}
z = x
```



Constructing CFGs: *for* loops

```
for (x = 0; x < y; x++)
{
   y = y - 2;
}
z = x
```

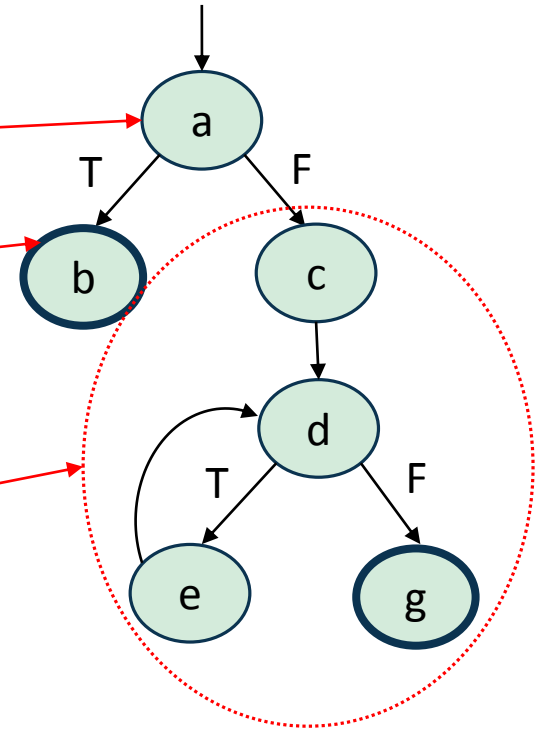


Applying white-box testing to *computeAverage…*

```
public double computeAverage (int [] numbers)
{
        int length = numbers.length;
        double mean = 0;
        double sum = 0;
        if (length <= 0)
        {
                return Double.NaN;
        }
        for (int i = 0; i < length; i++)
        {
                sum += numbers [i];
        }
        mean = sum / (double) length;
        return mean;
}
```

In-class exercise

```
public double computeAverage (int [] numbers)
{
    int length = numbers.length;
    double mean = 0;
    double sum = 0;
    if (length <= 0)
    {
        return Double.NaN;
    }
    for (int i = 0; i < length; i++)
    {
        sum += numbers [i];
    }
    mean = sum / (double) length;
    return mean;
}
```



Let’s move on to using CFGs for testing….

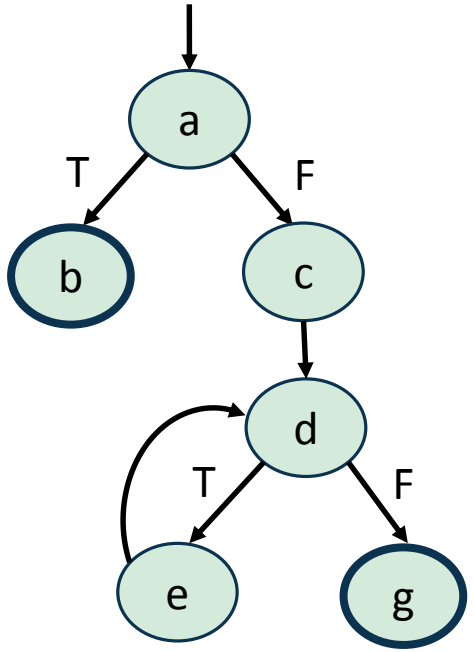
Using CFGs in testing

- Execution of a test case corresponds to traveling a path in the CFG
- This is called the ***test path***
- Test path should start from the start node and end from an exit node

Test path

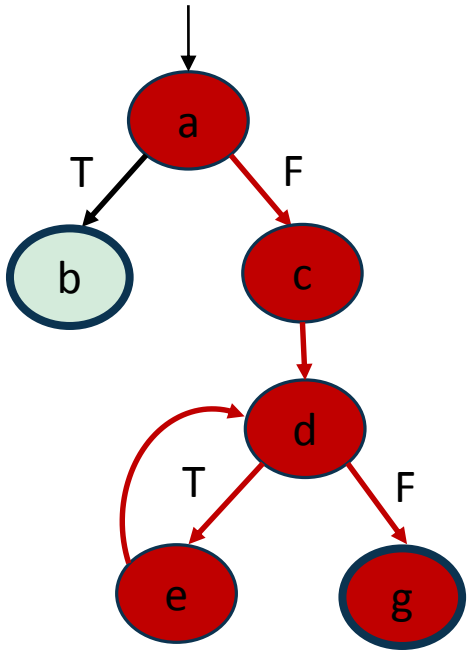
```
public double computeAverage (int [] numbers)
{
        int length = numbers.length;
        double mean = 0;
        double sum = 0;
        if (length <= 0)
        {
                return Double.NaN;
        }
        for (int i = 0; i < length; i++)
        {
                sum += numbers [i];
        }
        mean = sum / (double) length;
        return mean;
}
```

Test input: [2]



Test path: [a,c,d,e,d,g]

Test path: visits



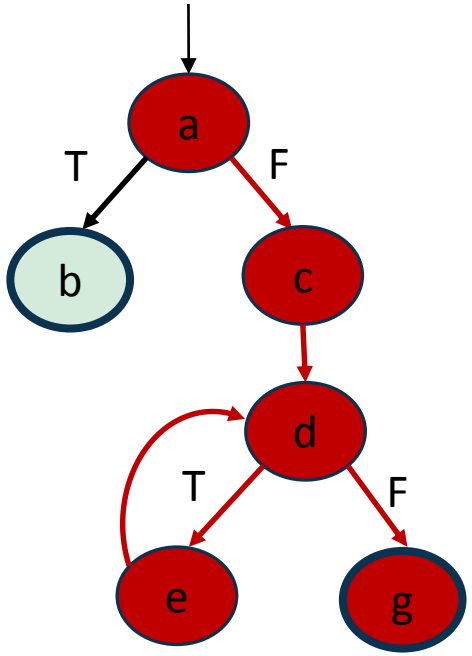
Test path: [a,c,d,e,d,g]

- Visits nodes: a,c,d,e,g
- Visits edges: (a,c), (c,d), (d,e), (e,d), (d,g)
- Note:
  - Does not visit all the edges
  - Does not visit all the nodes

Need to define our test criteria using the CFG

Test criteria: node coverage

- Satisfied if every reachable node in the CFG is visited by least one test path
- Corresponds to statement coverage in the program



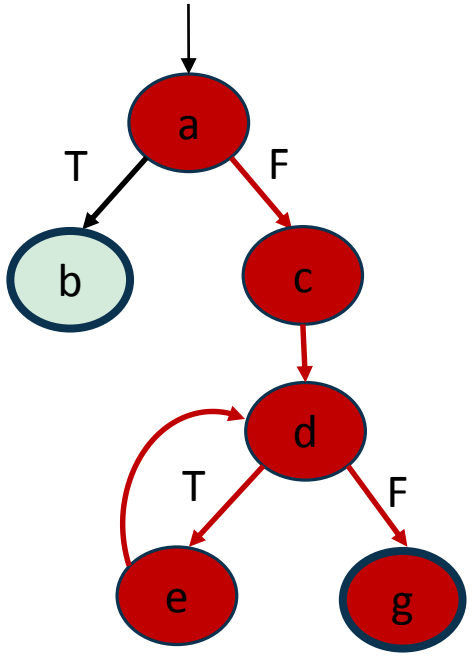
Test path: [a,c,d,e,d,g]

1. Does this test set satisfy the node coverage criteria?

2. What do we need to add?
   A test path that visits node b

Test criteria: edge coverage

- Satisfied if every reachable edge in the CFG is visited by least one test path
- Corresponds to branch coverage in the program



Test path: [a,c,d,e,d,g]

1. Does this test set satisfy the edge coverage criteria?

2. What do we need to add?

A test path that visits node edge (a,b)

Stronger criteria

- Set of test paths that satisfy edge coverage should satisfy node coverage
- Edge coverage criteria is stronger than node coverage criteria
- What does this mean in practice?

Coverage tool demonstration

- Coverage tool - EclEmma

Problems with edge coverage

- Consider the test cases required to satisfy edge coverage in *computeAverage*
- Some faults revealed when executing combinations of branches – paths
  - *Path coverage criteria*
  - Requires all possible paths in CFG to be executed during testing
  - Stronger than edge coverage

Problems with edge coverage

- Decisions having multiple conditions

```
if (x >= 0 && x <= 10)
{
   return true;
}
return false;
```

Fault should be 100

Test inputs: x=2 and x=-1

Solution: *decision/condition coverage*