Lab IV: Control Flow Graph and Data Flow Coverage

Software Testing and QA (COE891)

Week 8 and Week 9

1 Lab Objectives

- Trying to ensure that values are computed and used correctly.
- Checking if the structural flow of the program can be traversed correctly and completely.
- Finding the test paths of a program according to the locations of definitions and uses of variables in the program.
- Finding variables that is used but never defined, variables that is defined but never used, variables that is defined multiple times before its use, and deallocating variables before they are used.

2 Part I: Control Flow and Graph Coverage

2.1 Control Flow Coverage

A control flow graph (CFG) can be used to represent the control flow of a piece of (imperative) source code. Nodes represent basic blocks or sequences of instructions that always execute together in sequence. Edges represent control flow between basic blocks.

The entry node corresponds to a method's entry point. Final nodes correspond to exit points, e.g. in Java: return or throw instructions. Decision nodes represent choices in control flow - e.g. in Java: due to if, switch-case blocks or condition tests for loops. There are some definitions along with their prominent points to be mentioned in the following:

- **Node Coverage:** A test set T satisfies Node Coverage on graph G if and only if for every syntactically reachable node n in N, there is some path pin path(T) such that p visits n.
- **path(T):** the set of paths that are exercised by the execution of T. In other words, the set TR of test requirements for Node Coverage contains each reachable node in G.
- **Edge Coverage:** The TR for Edge Coverage contains each reachable path of length up to 1, inclusive, in a graph G. Note that Edge Coverage subsumes Node Coverage.
- The TR for Edge-Pair Coverage contains each reachable path of length up to 2, inclusive, in a graph G. This definition can be easily extended to paths of any length, although possibly with diminishing returns.
- The TR for Complete Path Coverage contain all paths in a G.
- A path p is simple if it has no repetitions of nodes other than (possibly) the first and last node. So, a simple path p has no internal loops, but may itself be a loop. there are too many simple paths, since many are just sub-paths of longer simple paths.
- If a graph contains a loop, it has an infinite number of paths. Thus Complete Path Coverage (CPC) is not feasible.
- Simple Path Coverage (SPC) is not satisfactory because the results are subjective and vary with the tester.

- **Prime Path:** A path is a prime path if it is a simple path, and it does not appear as a proper sub-path of any other simple path. In other words, a path p is prime if and only if p is a maximal simple path. This cuts down the number of cases to consider. The TR for Prime Path Coverage contains each prime path in a G.

It is noteworthy that when you are creating and plotting the control flow graph for a source code, you should consider some specific architecture and dummy nodes to plot your graph in a suitable way:

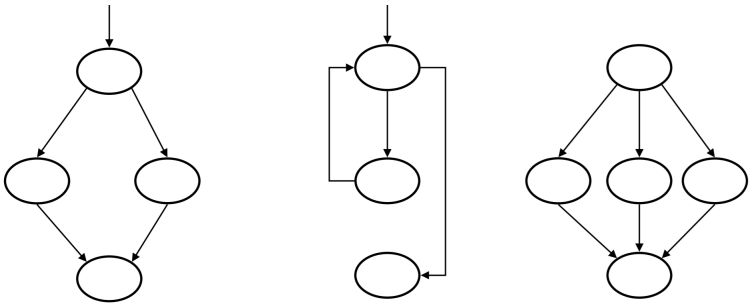

Figure 1

Consider the following control flow graph:

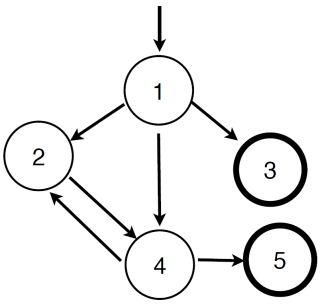

Figure 2

- **Node coverage (NC):** Test requirements: cover every node (all graph paths up of length 0).

$$TR(NC) = set\ of\ nodes\ in\ the\ graph\ = \\ \{[1], [2], [3], [4], [5]\}$$

- **Edge coverage (EC):** cover every edge.
  Test requirements: cover every edge (all paths up to length 1).

$$TR(EC) = set\ of\ edges\ in\ the\ graph\ EC\ subsumes\ NC\ = \\ TR(NC) \cup \{[1,2], [1,3], [1,4], [2,4], [4,2], [4,5]\}$$

- $T_1 = \{[1,3], [1,2,4,5]\}$ satisfies NC, but not EC.
- $T_2 = \{[1,3], [1,2,4,5], [1,4,2,4,5]\}$ satisfies both NC and EC.

As another example, consider the following CFG:

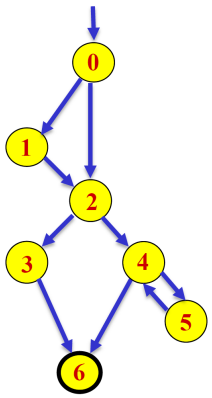

Figure 3

- Node Coverage:

$$TR = \{0, 1, 2, 3, 4, 5, 6\}$$
$$TestPaths : \{[0, 1, 2, 3, 6], [0, 1, 2, 4, 5, 4, 6]\}$$

- Edge Coverage:

$$TR = \{(0, 1), (0, 2), (1, 2), (2, 3), (2, 4), (3, 6), (4, 5), (4, 6), (5, 4)\}$$
$$TestPaths : \{[0, 1, 2, 3, 6], [0, 2, 4, 5, 4, 6]\}$$

- Edge-Pair Coverage:

$$TR = \{[0, 1, 2], [0, 2, 3], [0, 2, 4], [1, 2, 3], [1, 2, 4], [2, 3, 6], [2, 4, 5], [2, 4, 6], [4, 5, 4], [5, 4, 5], [5, 4, 6]\}$$
$$TestPaths : \{[0, 1, 2, 3, 6], [0, 1, 2, 4, 6], [0, 2, 3, 6], [0, 2, 4, 5, 4, 5, 4, 6]\}$$

- Complete Path Coverage:

$$TestPaths : \{[0, 1, 2, 3, 6], [0, 1, 2, 4, 6], [0, 1, 2, 4, 5, 4, 6], [0, 1, 2, 4, 5, 4, 5, 4, 6], [0, 1, 2, 4, 5, 4, 5, 4, 5, 4, 6], ...\}$$

3 Part II: Data Flow Graph and Coverage

A *definition* is a location where a value for a variable is stored into memory, for example, Assignment, input, parameter passing, and so on.
A *use* is a location where a variable's value is accessed. There are two kinds of use:

- **p-use:** a use that occurs in a predicate expression, i.e., an expression used as a condition in a branch statement.
- **c-use:** a use that occurs in an expression that is used to perform certain computation.

A data flow graph (DFG) captures the flow of data in a program. To build a DFG, we first build a CFG and then annotate each node n in the CFG with the following two sets:

- **def(n):** The set of variables defined in node n.
- **use(n):** The set of variables used in node n.

```
1  float x, y, z = 0.0;
2  int count;
3  input (x, y, count);
4  do {
5      if(x <= 0) {
6          if(y >= 0)
7              z = y * z + 1; }
8      else
9          z = 1/x;
10     y = x * y + z;
11     count = count - 1; }
12 while(count > 0)
13 System.out.println(z);
```

Deriving a data flow graph:

Node	Lines
1	1, 2, 3, 4
2	5, 6
3	7
4	8, 9, 10
5	11, 12, 13
6	14, 15, 16
7	17, 18

Figure 4

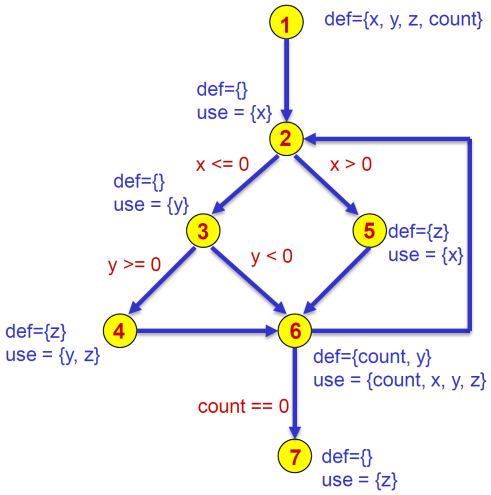

Figure 5

Now, let's consider some other definitions in data flow of a program:

- **PATHS(P):** The set of all CFPs in program P.
- **Definition-use (du)-path:** A definition-use path with respect to a variable v (denoted du-path) is a path in PATHS(P) such that, for some $v \in V$, there are definition and usage nodes $DEF(v, m)$ and $USE(v, n)$ such that m and n are initial and final nodes of the path.
- **Definition-clear (dc)-path:** A definition-clear path with respect to a variable v (denoted dc-path) is a definition-use path in PATH(P) with initial and final nodes $DEF(v, m)$ and $USE(v, n)$ such that no other node in the path is a defining node of v.

3.1 A Simple Example

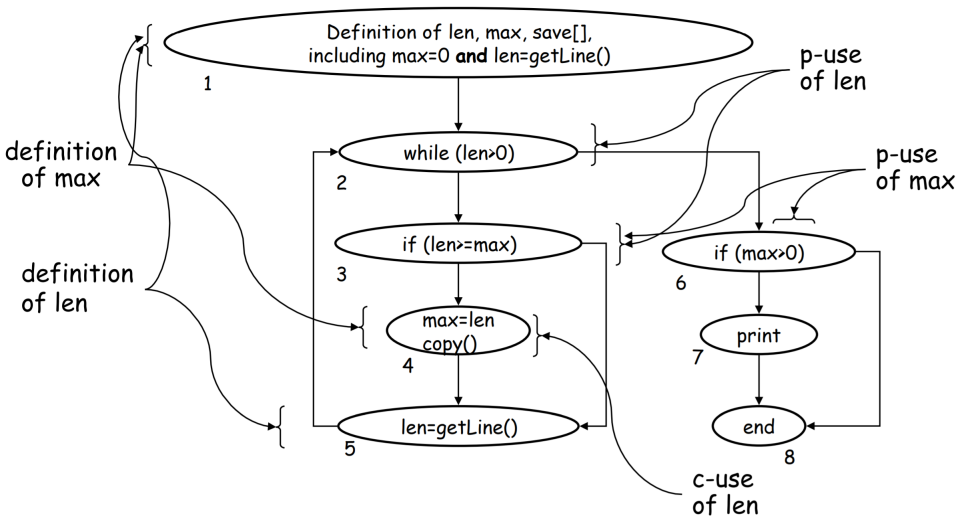

Figure 6

Based on the above graph of a specific program:

- $[1-2], [1-2-3], [1-2-3-4], [5-2], [5-2-3], [1-2-3-4-5-2]$ are du-paths based on the variable *len*.
- They are, except for $[1-2-3-4-5-2]$, definition-clear based on the variable *len*.

So, a data-flow graph of a program, denoted as def-use graph, captures the flow of definitions (denotes as defs) and uses across basic blocks in a program. It is similar to a control flow graph of a program in that the nodes, edges, and all paths in the control flow graph are preserved in the data flow graph.

Attach defs, c-use and p-use to each node in the graph. Label each edge with the condition which when true causes the edge to be taken. We use $d_i(x)$ to refer to the definition of variable x at node i. Similarly, $u_i(x)$ refers to the use of variable x at node i.

Any path starting from a node at which variable x is defined and ending at a node at which x is used, without redefining x anywhere else along the path, is a **def-clear path** for x.

Def of a variable at a line (or node) and its *use* at another line constitute a def-use pair. These two lines (or nodes) can be the same. $dcu(x, i)$ denotes the set of all nodes where the definition of x at node i is live and used. $dpu(x, i)$ denotes the set of all edges (k, l) such that there is a def-clear path from node i to edge (k, l) and x is used at node k. We say that a def-use pair $(d_i(x), u_j(x))$ is covered when a def-clear path that includes nodes i to node j is executed. If $u_j(x)$ is a p-use then all edges of the kind (j, k) must also be taken during some executions for the def-use pair to be covered.

For example, you can see a program below (computing factorial of a number) and its following table of def and use for each variable:

```
1  public int factorial(int n){
2      int i, result = 1;
3      for (i=2; i<=n; i++)
4          result = result * i;
5      return result; }
```

Variable	Definition line	Use line
n	1	3 **(Predicate)**
result	2	4 **(Computation)**
result	2	6
result	4	4
result	4	6
i	3	3
i	3	4

Figure 7

Note that the minimum number of test cases required to test all program paths is equal to the cyclomatic complexity (CC).

$$CC = Number\ of\ edges - Number\ of\ nodes + 2$$

3.2 Data Flow Test Criteria

We try to make sure that every def reaches all possible uses:
All-Uses Coverage (AUC): For each set of du-paths to uses $S = du(n_i, n_j, v)$, TR contains at least one path d in S.
Finally, we cover all the du-paths between defs and uses:
All-du-Paths Coverage (ADUPC): For each set $S = du(n_i, n_j, v)$, TR contains every path d in S.
First, we make sure every def reaches a use:
All-Defs Coverage (ADC): For each set of du-paths $S = du(n, v)$, TR contains at least one path d in S.

4 A Complete Exercise

```
1  public static int occurrences(char[] v, char c) {
2      if (v == null)
3          throw new NullPointerException();
4      int n = 0;
5      for (int i=0; i < v.length; i++) {
6          if (v[i] == c)
7              n++; }
8      return n;
9      }
```

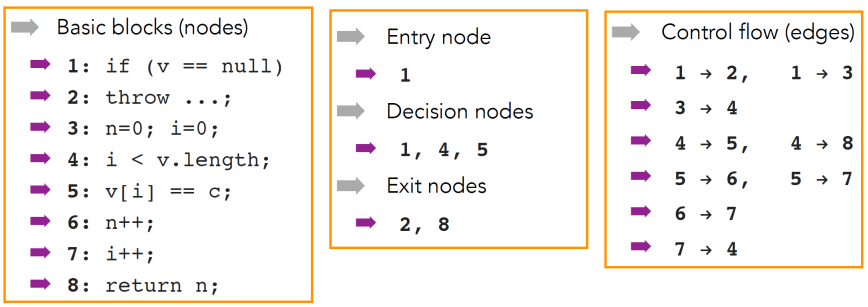

Figure 8

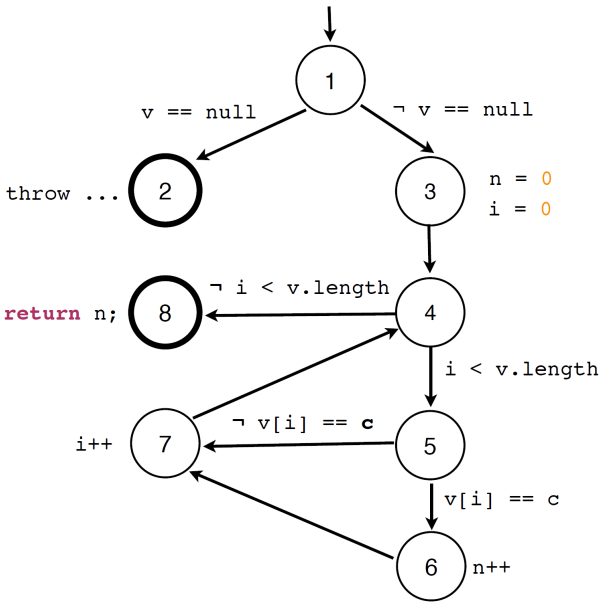

Figure 9

- Basic blocks (nodes)
  - 1: if (v == null)
  - 2: throw ...;
  - 3: n=0; i=0;
  - 4: i < v.length;
  - 5: v[i] == c;
  - 6: n++;
  - 7: i++;
  - 8: return n;
- Control flow (edges)
  - 1 → 2, 1 → 3
  - 3 → 4
  - 4 → 5, 4 → 8
  - 5 → 6, 5 → 7
  - 6 → 7
  - 7 → 4

Figure 10

Node coverage

TR(NC) = { [1], [2],[3],[4], [5],[6],[7],[8] }

NC satisfied by { t1, t2 } or {t1, t3}

Edge coverage

TR(EC) = TR(NC) U {
[1,2],[1,3],[3,4],[4,5],[4,8], [5,6],[5,7][6,7],[7,4]
}

EC satisfied by { t1, t3 } but not by {t1,t2}.

Figure 11

t	test case values (v,c)	exp. values	test path	covered nodes	covered edges
t1	(null, 'a')	NPE.	[1,2]	1 2	[1,2]
t2	({'a'}, 'a')	1	[1,3,4,5,6,7,4,8]	1 3 4 5 6 7 8	[1,3][3,4][4,5][5,6] [6,7][7,4][4,8]
t3	({'x','a'}, 'a')	1	[1,3,4,5,7,4,5,6,7,8]	1 3 4 5 6 7 8	[1,3][3,4][4,5][5,6] [6,7][7,4][5,7][4,8]

Figure 12

Please note that the number of paths may be infinite (e.g., CFGs with loops), CPC (complete path coverage) is not applicable in most cases.

Edge-pair coverage

TR(EPC) = {
[1,2], [1,3,4],
[3,4,5], [3,4,8],
[4,5,6], [4,5,7],
[5,6,7],[5,7,4],
[6,7,4],
[7,4,5], [7,4,8]
}

EC satisfied by {t1, t3}.

EPC satisfied by { t1, t2, t3 } but not by {t1,t3}.
Observe that t3 does not cover [3,4,8].

Figure 13

t	test case values (v,c)	exp. values	test path	covered requirements
t1	(null, 'a')	NPE.	[1,2]	[1,2]
t2	({}, 'a')	0	[1,3,4,8]	[1,3,4][3,4,8]
t3	({'x','a'}, 'a')	1	[1,3,4,5,7,4,5,6,7,4,8]	[1,3,4][3,4,5] [4,5,7][5,7,4] [7,4,5][4,5,6] [5,6,7][7,4.8]

Figure 14

Prime Path Coverage (PPC): A path $p = [n_1, n_2, ..., n_M]$ is a simple path if no node appears more than once, other than possibly the first and last ones. A simple path has no internal loops, but may represent a loop if the first and last nodes are equal.

A prime path is a maximal length simple path, i.e., a simple path that is not a proper sub-path of any other simple path. PPC requires every prime path to be covered by the test set.

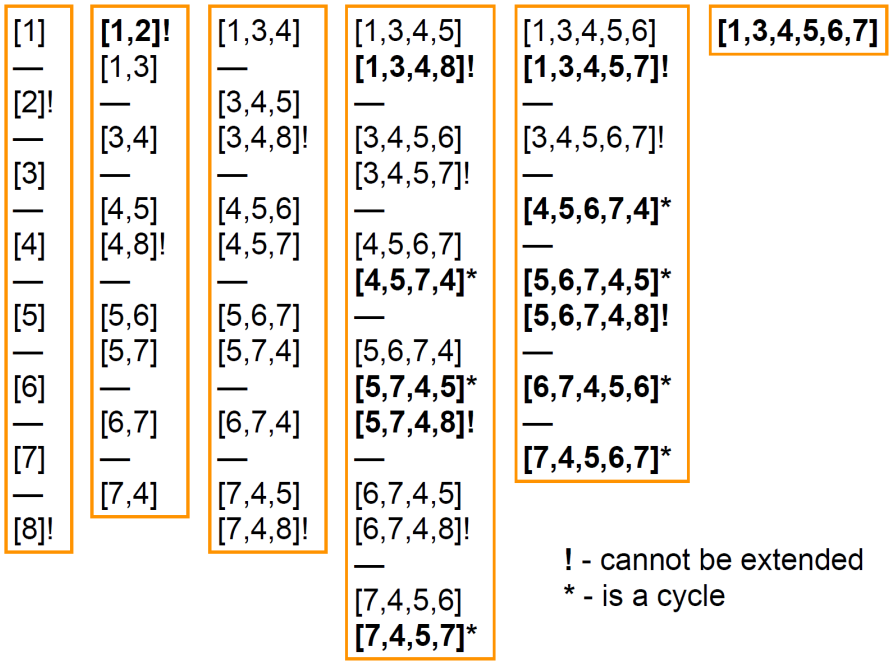

Figure 15

Deriving prime paths: Enumerate all simple paths of length 0, 1, 2, 3, ... until no more simple paths are found. Pick the prime paths among all derived simple paths.

Prime path	Covered by
[1,2]	t1
[1,3,4,8]	t2
[1,3,4,5,6,7]	t4 t5
[1,3,4,5,7]	t3
[4,5,7,4]	t3 t4
[4,5,6,7,4]	t3 t4 t5
[5,7,4,5]	t3
[5,7,4,8]	t4
[5,6,7,4,5]	t4 t5
[5,6,7,4,8]	t3 t5
[6,7,4,5,6]	t5
[7,4,5,7]	t4
[7,4,5,6,7]	t3 t5

Figure 16

t	test case values (v,c)	exp. values	test path
t1	(null, 'a')	NPE.	[1,2]
t2	({}, 'a')	0	[1,3,4,8]
t3	({'x','a'}, 'a')	1	[1,3,4,5,7,4,5,6,7,4,8]
t4	({'a','x'}, 'a')	1	[1,3,4,5,6,7,4,5,7,4,8]
t5	({'a','a'}, 'a')	2	[1,3,4,5,6,7,4,5,6,7,4,8]

Figure 17

PPC satisfied by $\{t1, t2, t3, t4, t5\}$.

4.1 Best-effort Touring

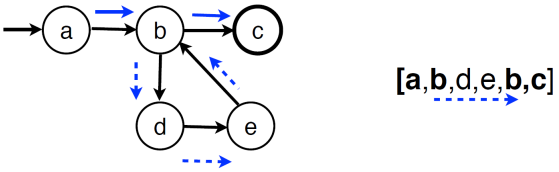

Figure 18

Assume that $p = [a, b, d, e, b, c]$ is a feasible test path in Figure 17, but $q = [a, b, c]$ is not. We say p tours q with a side-trip since every edge in p also appears in q in the same order (the sidetrip is $[b, d, e, b]$). A test set T achieves best effort touring of TR if for every path in TR there is a test in T that tours the path either directly or using side-trips. Best-effort touring is a relevant way to deal with infeasible test requirements.

5 Assignments

Q1. Power function is a program to compute $Z = X^Y$. Provide CFG for the following program:

```
BEGIN
    read (X, Y);
    W = abs(Y);
    Z = 1;
    WHILE (W != 0)
        Z = Z * X;
        W = W - 1;
    END
    IF (Y < 0)
        Z = 1 / Z;
    END
    print (Z);
END
```

Identify:

1. Infeasible paths.
2. Enough test cases for node coverage.
3. Enough test cases for edge coverage.

Q2. For the program shown below:

```
public static boolean isPalindrome(String s) {
    if (s == null)
        throw new NullPointerException();
    int left = 0;
    int right = s.length() - 1;
    boolean result = true;
    while (left < right && result == true) {
        if (s.charAt(left) != s.charAt(right)) {
        result = false; }
        left++;
        right--; }
    return result;
    }
```

1. Draw the CFG for isPalindrome.
2. Identify TR(NC), TR(EC), TR(EPC).
3. If possible define test sets that satisfy:
   - NC but not EC.
   - EC but not EPC.
   - EPC.
4. Identify TR(PPC). Are there infeasible requirements? Define test cases for all the feasible requirements.
5. Write a JUnit test class for the test cases in 4.

Q3. For the program shown below:

```
public static void computeStats (int[] numbers) {
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;
    sum = 0;
    for (int i = 0; i < length; i++)
        sum += numbers [ i ];
```

```
7       med = numbers [length / 2];
8       mean = sum / (double) length;
9       varsum = 0;
10      for (int i = 0; i < length; i++)
11          varsum = varsum + ((numbers[i] - mean) * (numbers[i] - mean));
12      var = varsum / (length - 1.0);
13      sd = Math.sqrt(var);
14      System.out.println("length: " + length);
15      System.out.println("mean: " + mean);
16      System.out.println("median: " + med);
17      System.out.println("variance: " + var);
18      System.out.println("standard deviation: " + sd);
19  }
```

Please provide:

1. control flow and data flow coverage graph.
2. du pairs (node pairs) for each variable.
3. du paths for each du pair for each variable.
4. test cases to cover du paths.
5. some du paths require arrays with length 0. In these cases, what will happen with the program? Why?

Q4. Consider the method printPrimes() reported below.

```
1   /** ****************************************************
2   * Finds and prints n prime integers
3   ****************************************************** */
4   private static void printPrimes (int n) {
5       int curPrime; // Value currently considered for primeness
6       int numPrimes; // Number of primes found so far.
7       boolean isPrime; // Is curPrime prime?
8       int[] primes = new int [100]; // The list of prime numbers.
9       // Initialize 2 into the list of primes.
10      primes[0] = 2;
11      numPrimes = 1;
12      curPrime = 2;
13      while (numPrimes < n) {
14          curPrime++; // next number to consider ...
15          isPrime = true;
16          for (int i = 0; i <= numPrimes-1; i++) { // for each previous prime.
17              if (isDivisible (primes[i], curPrime)) { /* Found a divisor, curPrime
18              is not prime. */
19                  isPrime = false;
20                  break; // out of loop through primes.
21                  }
22          }
23          if (isPrime) { // save it!
24              primes[numPrimes] = curPrime;
25              numPrimes++; }
26      } // End while
27      // Print all the primes out.
28      for (int i = 0; i <= numPrimes-1; i++)
29          System.out.println ("Prime: " + primes[i]);
30  } // end printPrimes
```

1. Draw the control flow graph for the printPrimes() method.
2. For printPrimes(), find a test case such that the corresponding test path visits the edge that connects the beginning of the while statement to the for statement that appears after the while loop, without going through the body of the while loop.
3. Write a (set of) test path(s) that (as a set, if you listed more than one) achieve Edge Coverage but not Prime Path Coverage on the graph. For each of them, identify the input(s) that will allow its execution. Some suggestions:

- Start by identifying the TRs for both criteria;
- More simple test paths will make your job easier when you have to identify the corresponding input than a single complex path.

4. Create effective JUnit test cases (i.e., check for the correct output condition) for the values questions 2 and 3.

6 Submission and Marking Instructions

6.1 Submissions

Once all required tasks are completed, you should submit your lab assignment. Follow the instructions below for a valid submission:

- After checking the accuracy and completeness of your assignment tasks, you should export and submit the FULL Eclipse project/folder as a ZIP file containing packages and source code files (for implementation and coding tasks/questions).

  - Individual files (e.g. java or class files) will NOT be accepted as a valid submission since your submitted package or Java project should be run completely and successfully by itself.

- You MUST submit a single PDF file containing required description or explanation for each of the assignment tasks/questions separately including any required justification, graphs, diagrams, test requirements/cases, calculation, pictures/snapshots from tools or IDE, test results, and so forth.

- The **submission deadline** for this lab (Lab IV) is the corresponding lab session in **Week 10** for both parts (Part I and Part II) altogether.

- The lab demo and questioning-answering will be held during the lab sessions of the corresponding submission weeks.

6.2 Marking Scheme

- This lab (Lab IV) constitutes 5% of your entire grade for this course.

- All assignment tasks/questions in each lab have the same grade.

- The grade for each lab is constituted from 50% for the lab submissions, 10% for the lab attendance, and 40% for demo and questioning-answering during the lab session.

- Note that all the labs constitute 25% of your entire grade for this course.