Faculty of Engineering & Architectural Science

Department of Biomedical, Computer, and Electrical Engineering

Course Number	COE891
Course Title	Software Testing and Quality Assurance
Semester/Year	Winter2024
Instructor	Dr. Reza Samavi
Section No.	01
Group No.	N/A
Submission Date	Mar 18, 2024 10:00 AM
Due Date	Mar 18, 2024

Lab/Tut Assignment NO.	4

Assignment Title	Control Flow Graph and Data Flow Coverage

Name	**Student ID**	**Signature***
Astha Patel	501040209	

By signing above you attest that you have contributed to this submission and confirm that all work you have contributed to this submission is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a "0" on the work, an "F" in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: www.ryerson.ca/senate/current/pol60.pdf

Q1

```
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
```

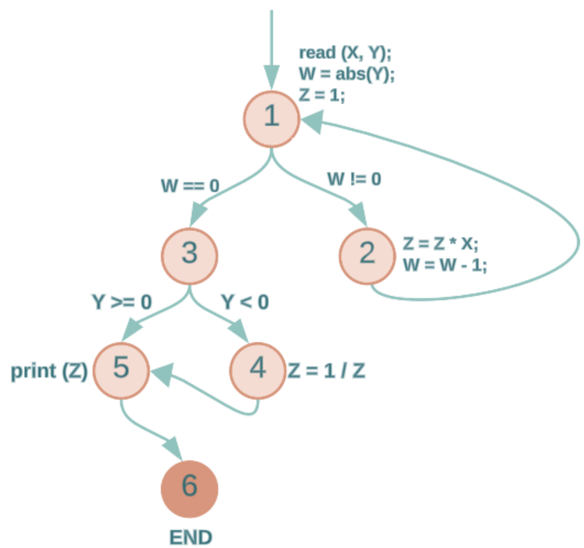

Figure 1: CFG of the above code in Q1

1. **Infeasible paths:**

   The program does not have infeasible paths; all parts of the program are feasible where every node is reachable by traversing through different edges.

2. **Test paths for node coverage:**

   ```
   TR = {[1], [2], [3], [4], [5], [6]},
   Test paths = {t1, t2},
   t1 = {1, 3, 4, 5, 6},
   t2 = {1, 2}
   ```

3. **Test paths for edge coverage (edges of at least length 1 or more):**

   ```
   TR = {(1,2), (2,1), (1,3), (3,4), (3,5), (4,5), (5,6)},
   Test paths = NC ∪ {t1, t2, t3}
   t1 = {1, 3, 4, 5, 6},
   t2 = {1, 2, 1},
   t3 = {1, 3, 5, 6}
   ```

Q2

```java
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. **CFG for the isPalindrome() method**

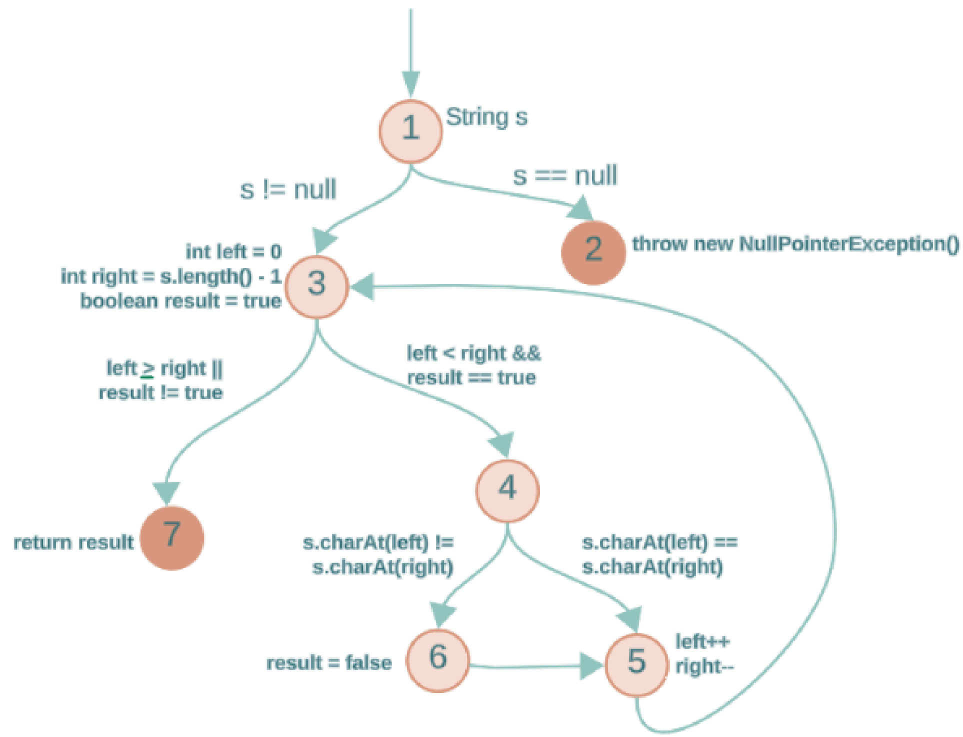

Figure 2: CFG for isPalindrome method in Q2.

2. **TR(NC), TR(EC), TR(EPC):**
   a. **TR(NC)**
   `TR = {[1], [2], [3], [4], [5], [6],[7]},`

   b. **TR(EC)**
   `TR = {(1,2), (1,3), (3,4), (3,7), (4,5), (4,6), (6,5), (5,3)}`

   c. **TR(EPC)**

TR = {(1, 3, 7),(1, 3, 4),(3, 4, 5),(3, 4, 6), (4, 6, 5), (4, 5, 3), (5, 3, 7), (5, 3, 4)}

3. **Paths to satisfy the following**

   d. **NC but not EC**

   TR = NC but not EC

   - Test paths for NC but not EC do not exist since EC subsumes NC. At least one edge is required to reach another node, this would satisfy EC since an edge of length of at least one is required.

   e. **EC but not EPC**

   TR = EC but not EPC

   - Test paths for the isPalindrome() method can satisfy EC but not EPC when the method throws a nullPointerException. In this case, the path traversed is only of length 1: from node 1 to node 2 before the code exits.

   f. **EC but not EPC**

   Test paths: NC ∪ {t1, t2, t3, t4, t5, t6, t7, t8, t9}

   t1 = {1, 3, 4, 6, 5, 3},

   t2 = {1, 3, 7},

   t3 = {1, 3, 4},

   t4 = {3, 4, 5},

   t5 = {3, 4, 6},

   t6 = {4, 5, 3},

   t7 = {4, 6, 5},

   t8 = {5, 3, 7},

   t9 = {5, 4, 3}

4. **TR(PPC)**

   TR = PPC for the CFG in Figure 1.

```
Test paths = {t1,t2,t3,t4}
t1 = {1,2},
t2 = {1,3,7},
t3 = {1,3,4,6,5,3,7},
t4 = {1,3,4,5,3,7},
```

- The isPalindrome() method does not have any infeasible requirements. Every branch and statement in the method is reachable through different input conditions. The following JUnit test in part 5 demonstrates with different test cases that every branch and statement in the program is feasible.

5. **JUnit tests**

The following code snippet showcases the JUnit tests for the test paths. EAch case is tailored to reach its respective test path. The test paths are tested in the order as follows in part 4: t1, t2, t3, t4.

```java
package assignmentTests;
import static org.testng.Assert.assertEquals;
import static org.testng.Assert.assertThrows;
import org.junit.Test;
import assignments.Q2Part4;
public class Q2Part4Test {

	@Test
	public void test1() throws Exception{
		System.out.println("Test path 1");
		assertThrows(NullPointerException.class, () -> Q2Part4.isPalindrome(null));
	}
	@Test
	public void test2() {
		System.out.println("Test path 2");
		boolean expected = true;
		boolean actual = Q2Part4.isPalindrome("A");
```

```
            assertEquals(expected, actual);
        }

        @Test
        public void test3() {
            System.out.println("Test path 3");
            boolean expected = false;
            boolean actual = Q2Part4.isPalindrome("AC");
            assertEquals(expected, actual);
        }

        @Test
        public void test4() {
            System.out.println("Test path 4");
            boolean expected = true;
            boolean actual = Q2Part4.isPalindrome("AA");
            assertEquals(expected, actual);
        }
}
```

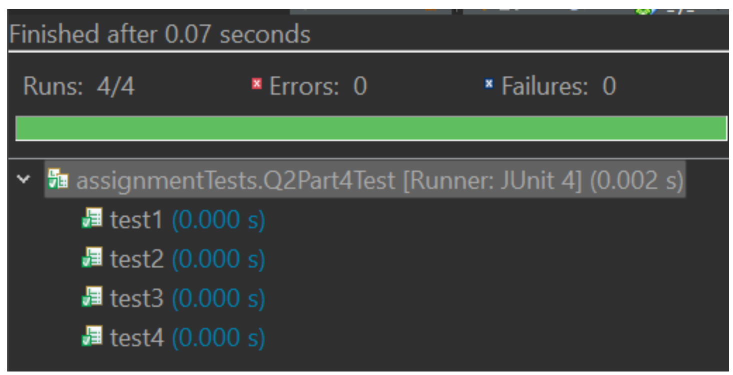

Figure 1: The test results for the JUnit test class in part 5.

Q3

```java
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 ];

	med = numbers [length / 2];
	mean = sum / (double) length;
	varsum = 0;
	for (int i = 0; i < length; i++)
		varsum = varsum + ((numbers[i] - mean) * (numbers[i] - mean));

	var = varsum / (length - 1.0);
	sd = Math.sqrt(var);
	System.out.println("length: " + length);
	System.out.println("mean: " + mean);
	System.out.println("median: " + med);
	System.out.println("variance: " + var);
	System.out.println("standard deviation: " + sd);
}
```

1. **CFG and DFG of the above code for the computeStats(int[] numbers) method**

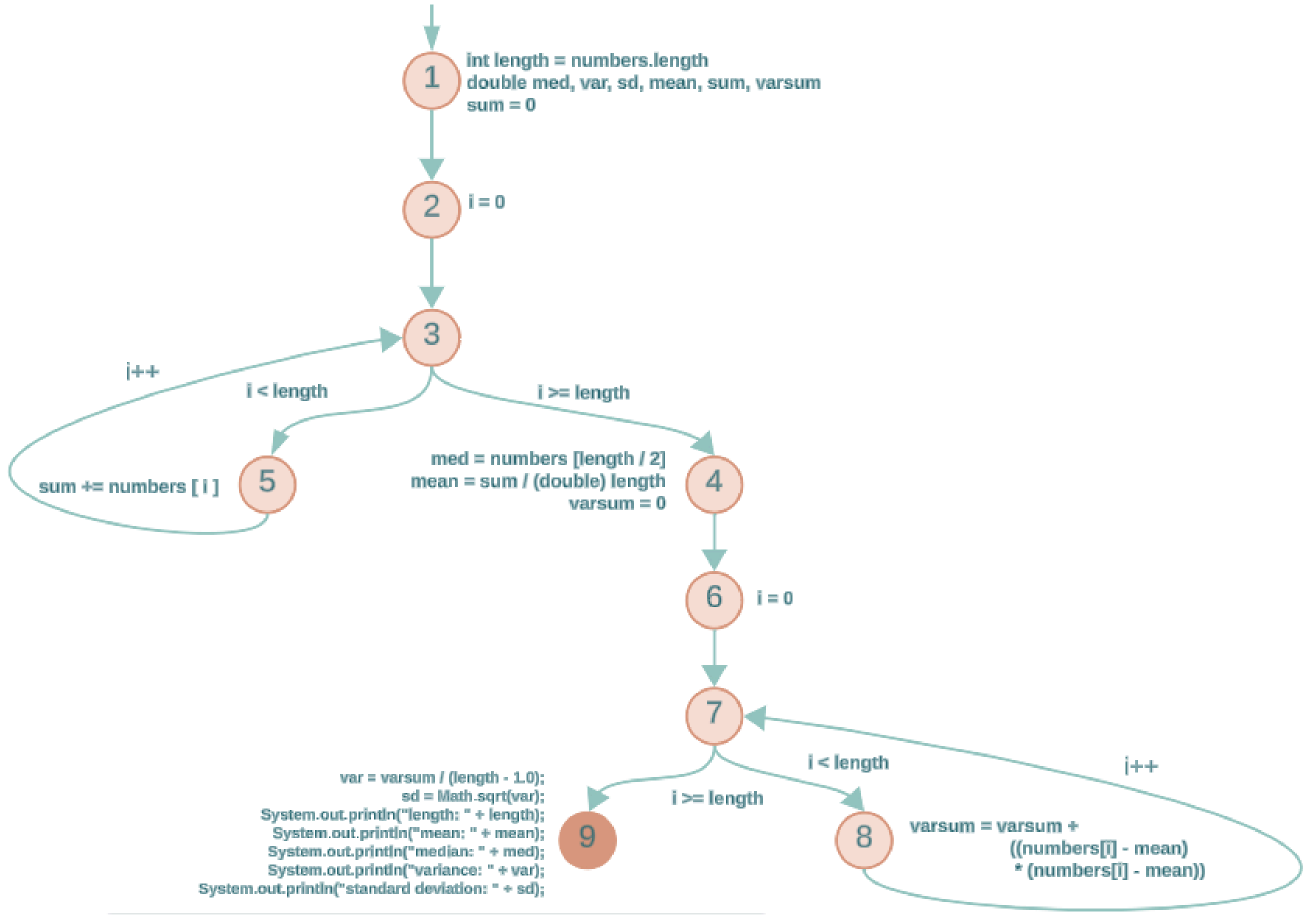

Figure 3: CFG for the code above in Q3

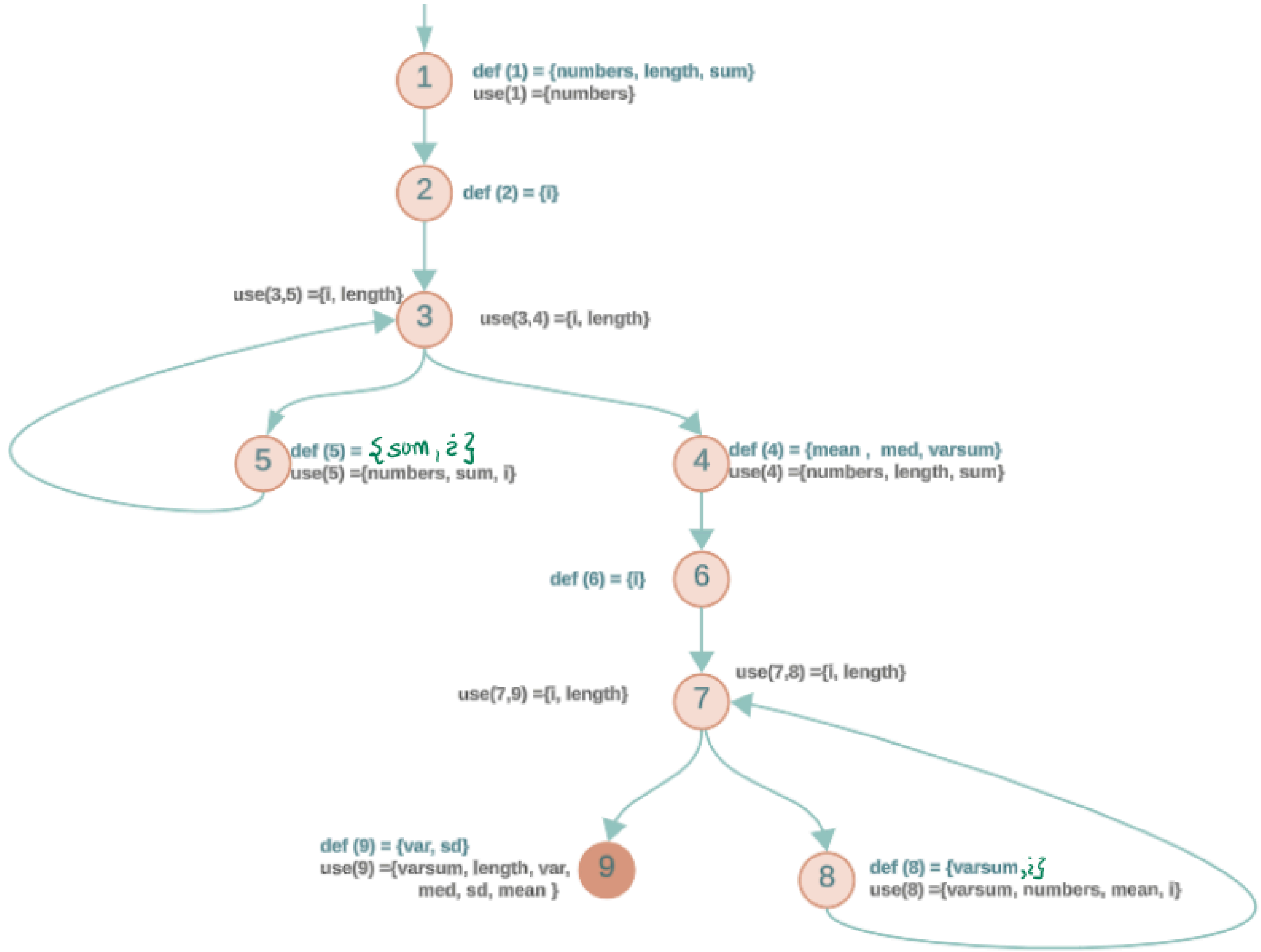

Figure 4: Data Flow Graph for the CFG in figure 3.

2. **DU pairs (nodes) for each variables**

Variable	DU Pairs
numbers	(1,5),(1,4),(1,8)
length	(1,4),(1,9),(1,(3,5)),(1,(3,4)),(1,(7,9)), (1,(7,8))
sum	(1,4),(5,5),(1,5),(5,4)
mean	(4,9),(4,8)
med	(4,9)
varsum	(4,9),(4,8),(8,8),(8,9)
var	(9,9)
sd	(9,9)
i	(5,5),(2,5),(2,(3,5)), (2,(3,4)), (6,(7,9)), (6,(7,8)), (6,8), (5,(3,4)), (8,8), (8,(7,9)),(8,(7,8)), (4,(7,8)), (4,(7,9)), (4,8)

3. **DU paths for each DU pair for each variable**

Variable	DU Pair	DU path
numbers(3)	(1,4) (1,5) (1,8)	{1,2,3,4} {1,2,3,5} {1,2,3,4,6,7,8}
length(6)	(1,4) (1,9)	{1,2,3,4} {1,2,3,4,6,7,9}

	(1,(3,5)) (1,(3,4)) (1,(7,9)) (1,(7,8))	{1,2,3,5} {1,2,3,4} {1,2,3,4,6,7,9} {1,2,3,4,6,7,8}
sum(4)	(1,4) (5,5) (1,5) (5,4)	{1,2,3,4} {5,3,5} {1,2,3,5} {5,3,4}
mean(2)	(4,9) (4,8)	{4,6,7,9} {4,6,7,8}
med(1)	(4,9)	{4,6,7,9}
varsum(4)	(4,9) (4,8) (8,8) (8,9)	{4,6,7,9} {4,6,7,8} {8,7,8} {8,7,9}
var(0)	(9,9)	{no path needed}
sd(0)	(9,9)	{no path needed}
i(14)	(5,5) (2,5) (2,(3,5)) (2,(3,4)) (6,(7,9)) (6,(7,8)) (6,8) (5,(3,4)) (8,8) (8,(7,9))	{5,3,5} {2,3,5} {2,3,5} {2,3,4} {6,7,9} {6,7,8} {6,7,8} {5,3,4} {8,7,8} {8,7,9}

	(8,(7,8)) (4,(7,8)) (4,(7,9)) (4,8)	{8,7,8} {4,6,7,8} {4,6,7,9} {4,6,7,8}

4. **Test paths to cover DU paths**

Test Cases	Test Path
Numbers = (44) Length = (1)	{1,2,3,5,3,4,6,7,8,7,9}
Numbers = (2, 10, 15) Length = 3	{1,2,3,5,3,5,3,5,3,4,6,7,8,7,8,7,8,7,9}

5. When arrays are required to have lengths 0 for some DU paths, the program will skip loops but the method will fail with an ArrayOutOfIndexException being thrown because there are no elements in the array.

<u>Q4</u>

```
/* ******************************************************
* Finds and prints n prime integers
******************************************************** */
private static void printPrimes (int n) {
      int curPrime; // Value currently considered for primeness
      int numPrimes; // Number of primes found so far.
```

```
      boolean isPrime; // Is curPrime prime?
      int[] primes = new int [100]; // The list of prime numbers.
      // Initialize 2 into the list of primes.
      primes[0] = 2;
      numPrimes = 1;
      curPrime = 2;
      while (numPrimes < n) {
            curPrime++; // next number to consider ...
            isPrime = true;
            for (int i = 0; i <= numPrimes-1; i++) { // for each previous
                           prime.
                  if (isDivisible(primes[i], curPrime)) { /* Found a
                        divisor, curPrime is not prime. */
                        isPrime = false;
                  break; // out of loop through primes.
                  }
            }
            if (isPrime) { // save it!
                  primes[numPrimes] = curPrime;
                  numPrimes++; }
      } // End while
 // Print all the primes out.
for (int i = 0; i <= numPrimes-1; i++)
      System.out.println ("Prime: " + primes[i]);
} // end printPrimes
```

1. **CFG graph**

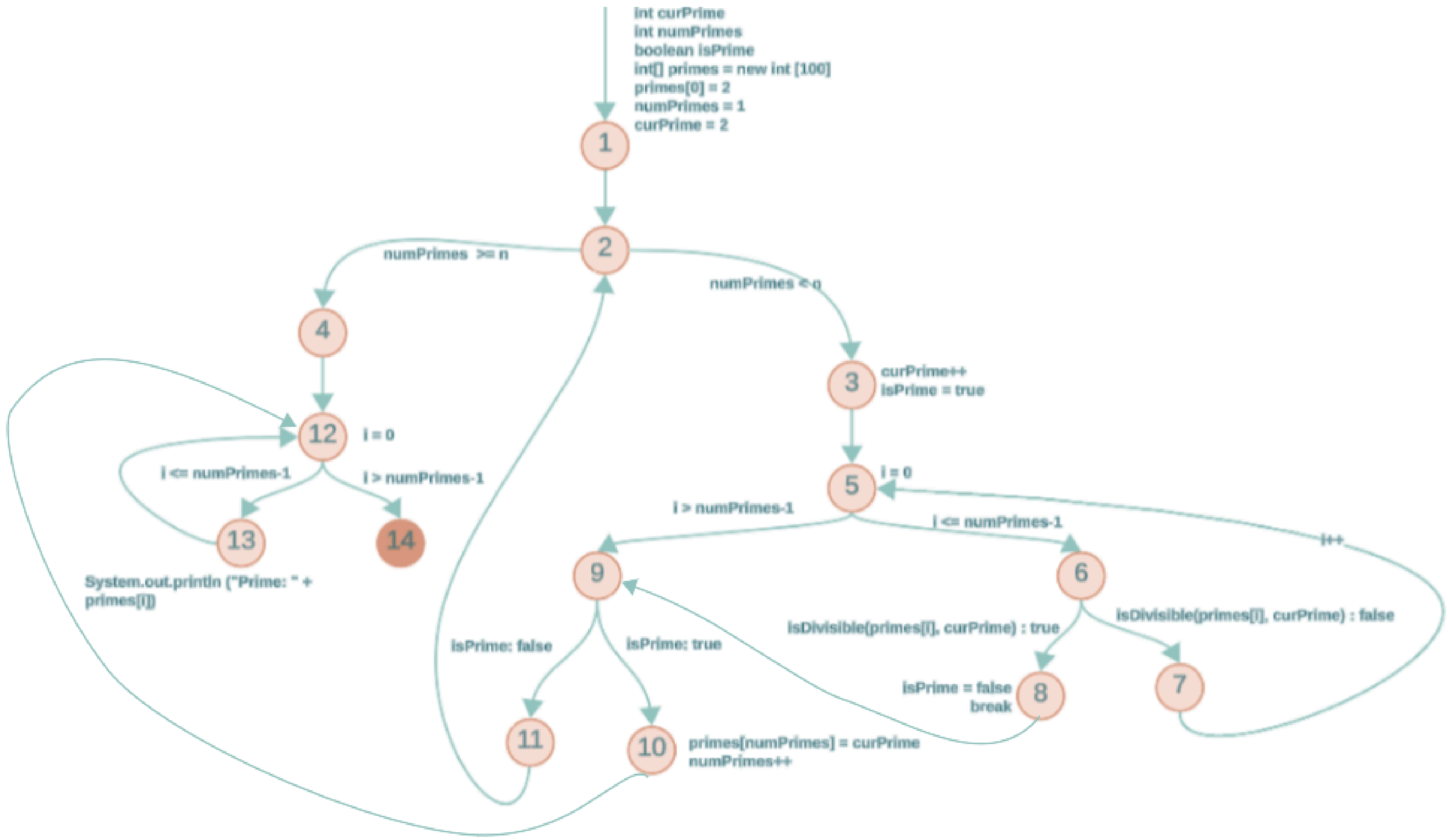

Figure 5: CFG of the above code in Q4.

2. **printPrimes(0) :** This input to the method will not execute the while loop since the while loop condition states that numPrime < n, where numPrimes is set to 1. Hence, the while loop condition fails since numPrimes is not less than the input value ‘n’.

3. **TR(EC)** =
{(1,2),(2,3),(2,4),(4,12),(12,13),(12,14),(3,5),(5,6),(6,7),(6,8),(5,9),(9,10),(9,11),(3,5,6,7),(3,5,6,8),(5,9,10),(5,9,11),(4,12,14),(1,2,4,12,14),...}
TR(PPC) = {[1,2,4,12,13,12,14],[1,2,4,12,14],[1,2,3,5,6,7,5,6,8,9,10,12,14],[]1,2,3,5,6,7,5,6,8,9,10,12,13,12,14],[1,2,3,5,6,7,8,9,11,2,3,5,6,7,5,6,8,9,10,12,14],[1,2,3,5,6,7,5,6,8,9,11,2,3,5,6,7,5,6,8,9,10,12,14],[1,2,3,5,6,7,5,9,10,12,14],[1,2,3,5,9,11,2,4,12,13,12,14],...}

printPrimes(1):

The test paths for this input is as follows : {1,2,4,12,14}

printPRimes(2):

The test paths for this input is as follows: {1,2,3,5,6,8,9,11,2,4,12,13,12,14}

printPrimes(3):

The test paths for this input is as follows: {1,2,3,5,6,7,5,6,7,5,8,9,11,4,12,13,12,14}

With inputs n =1 , n = 2, and n = 3, EC is achieved by covering every edge on the CGF and PPC is not achieved by not covering every PP on the CFG. Only a few prime paths are covered in the test cases.

4. **JUnit tests**

```
package assignmentTests;
import org.junit.Test;
import assignments.Q4;
import static org.junit.Assert.*;
public class Q4Test {
   @Test
   public void testPrintPrimesWithZero() {
       try {
           Q4.printPrimes(0);
       } catch (Exception e) {
           fail("Unexpected exception: " + e);
```

```
        }
    }
    @Test
    public void testPrintPrimesWithOne() {
        try {
            Q4.printPrimes(1);
        } catch (Exception e) {
            fail("Unexpected exception: " + e);
        }
    }
    @Test
    public void testPrintPrimesWithTwo() {
        try {
            Q4.printPrimes(2);
        } catch (Exception e) {
            fail("Unexpected exception: " + e);
        }
    }
    @Test
    public void testPrintPrimesWithThree() {
        try {
            Q4.printPrimes(3);
        } catch (Exception e) {
            fail("Unexpected exception: " + e);
        }
    }
}
```