IT 314 - Software Engineering
Lab-09
Mutation Testing

Student ID:-202201176

1. The code below is part of a method in the ConvexHull class in the VMAP system. The following is a small fragment of a method in the ConvexHull class. For the purposes of this exercise, you do not need to know the intended function of the method. The parameter p is a Vector of Point objects, p.size() is the size of the vector p, (p.get(i)).x is the x component of the ith point appearing in p similarly for (p.get(i)).y. This exercise is concerned with structural testing of code, so the focus is on creating test sets that satisfy some particular coverage criteria.

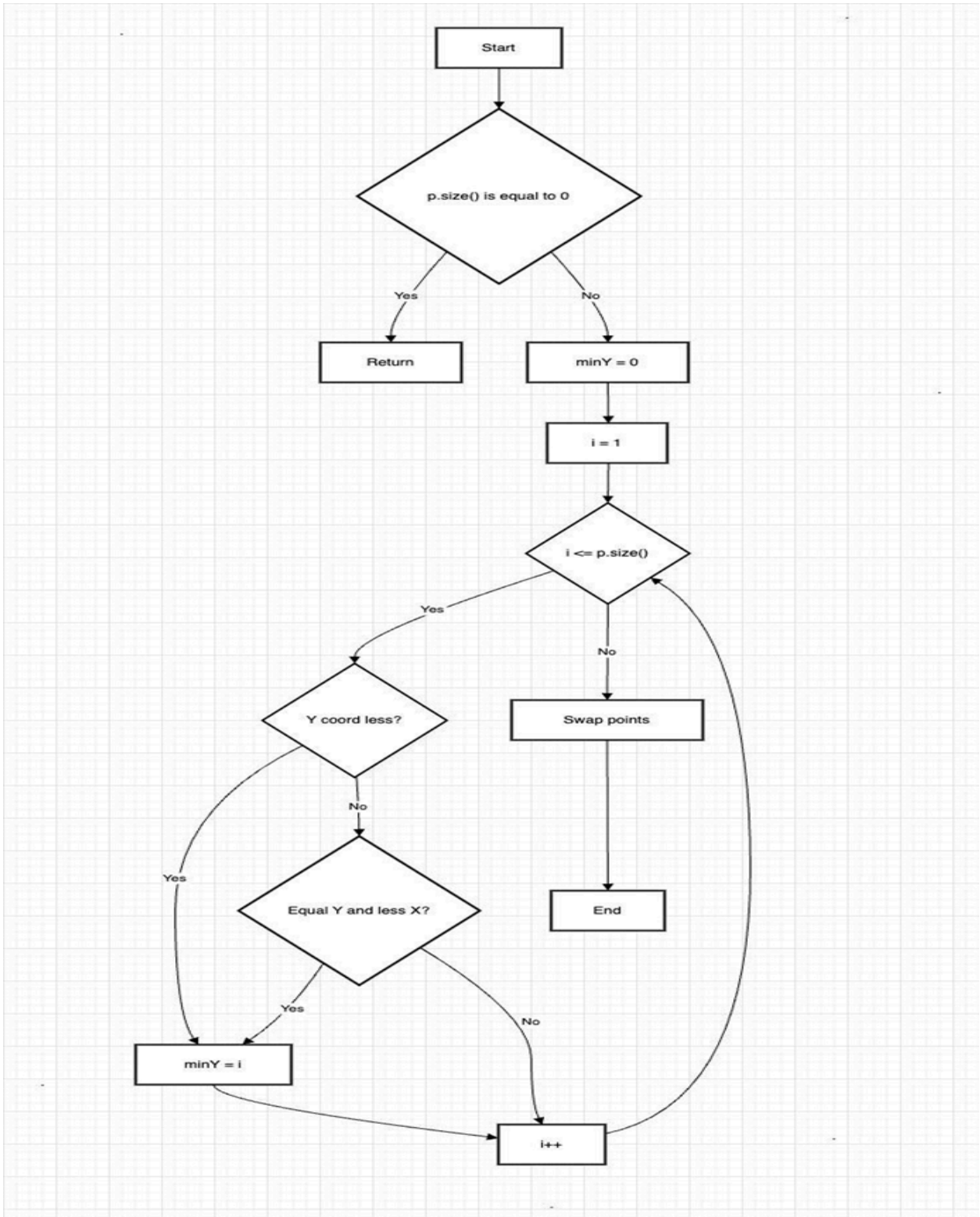
```
Vector doGraham(Vector p) {
        int i,j,min,M;

        Point t;
        min = 0;

        // search for minimum:
        for(i=1; i < p.size(); ++i) {
            if( ((Point) p.get(i)).y <
                        ((Point) p.get(min)).y )
            {
                min = i;
            }
        }

        // continue along the values with same y component
        for(i=0; i < p.size(); ++i) {
            if(( ((Point) p.get(i)).y ==
                        ((Point) p.get(min)).y ) &&
                 (((Point) p.get(i)).x >
                        ((Point) p.get(min)).x ))
            {
                 min = i;
            }
        }
```

For the given code fragment, you should carry out the following activities.

1. Convert the code comprising the beginning of the doGraham method into a control flow graph (CFG).

```
public class Point {
   double x;
   double y;

   public Point(double x, double y) {
      this.x = x;
      this.y = y;
   }
}

public class ConvexHull {
   public void doGraham(Vector<Point> p) {
      if (p.size() == 0) {
         return;
      }

      int minY = 0;
      for (int i = 1; i < p.size(); i++) {
         if (p.get(i).y < p.get(minY).y ||
            (p.get(i).y == p.get(minY).y && p.get(i).x < p.get(minY).x)) {
            minY = i;
         }
      }

      Point temp = p.get(0);
      p.set(0, p.get(minY));
      p.set(minY, temp);
   }
}
```

Control Flow graph of above code


Start
p.size() is equal to 0
Yes
No
Return
minY = 0
i = 1
i <= p.size()
Yes
No
Y coord less?
Swap points
No
Yes
Equal Y and less X?
End
Yes
No
minY = i
i++

2. Construct test sets for your flow graph that are adequate for the following criteria:
a. Statement Coverage.

To ensure that every line of code is executed at least once and achieve statement coverage, we propose the following test cases:

Test Cases

- **Test Case 1:** Check if the input vector is empty by evaluating `p.size() == 0`.

  Expected Outcome: The function should terminate immediately, performing no further actions.

- **Test Case 2:** Verify that the input vector p contains at least one `Point` element.

  Expected Outcome: The function should proceed to determine which point has the minimum y-coordinate.

b. Branch Coverage.

To achieve branch coverage, we must verify that each decision point in the code is evaluated as both true and false at least once.

Test Cases

- **Test Case 1:** Test with an empty vector p (i.e., `p.size() == 0`).

  Expected Outcome: When `p.size() == 0`, the branch evaluates to true, and the function exits immediately.

- **Test Case 2:** Test with a vector p containing a single point (e.g., `Point(0, 0)`).

  Expected Outcome: Since `p.size() == 0` evaluates to false, the function continues to the for loop, which does not execute as there is only one point present.

- **Test Case 3:** Test with a vector p containing multiple points where no point has a y-coordinate smaller than `p[0]`.

  Example: `p = [Point(0, 0), Point(1, 1), Point(2, 2)]`

Expected Outcome: The condition within the for loop remains false for all iterations, resulting in `minY` retaining its initial value of 0.

- **Test Case 4:** Test with a vector p containing multiple points where at least one point has a y-coordinate smaller than that of `p[0]`.

  Example: `p = [Point(2, 2), Point(1, 0), Point(0, 3)]`

  Expected Outcome: The condition inside the for loop evaluates to true at least once, leading to an update of `minY`.

c. Basic Condition Coverage.

To achieve basic condition coverage, it's essential to independently test each condition within the program's branches. This includes validating both the true and false outcomes of each condition separately. Below are the test cases designed for this purpose, along with their expected results:

Test Cases

- **Test Case 1:** Empty vector p (i.e., `p.size() == 0`).

  Objective: Verify that the condition `if (p.size() == 0)` returns true when the vector is empty.

  Expected Result: Since the condition `p.size() == 0` evaluates to true, the corresponding branch of the code should execute.

- **Test Case 2:** Non-empty vector p (i.e., `p.size() > 0`).

  Objective: Ensure that the condition `if (p.size() == 0)` evaluates to false when the vector contains elements.

  Expected Result: As the condition `p.size() == 0` is false, the alternative branch of the code should be executed.

- **Test Case 3:** Multiple points where the condition `p.get(i).y < p.get(minY).y` is true.

Example: `p = [Point(1, 1), Point(0, 0), Point(2, 2)]`.

Objective: Assess the condition `p.get(i).y < p.get(minY).y` to confirm it accurately identifies the minimum y-coordinate and updates `minY` accordingly.

Expected Result: The condition `p.get(i).y < p.get(minY).y` is true for the second point, `Point(0, 0)`, leading to an update of `minY` to index 1.

- **Test Case 4:** Multiple points where both conditions `p.get(i).y == p.get(minY).y` and `p.get(i).x < p.get(minY).x` are true.

  Example: `p = [Point(1, 1), Point(0, 1), Point(2, 2)]`.

  Objective: Ensure that when multiple points have the same y-value, the code correctly compares x-values to identify the smallest x-coordinate.

  Expected Result: Both conditions `p.get(i).y == p.get(minY).y` and `p.get(i).x < p.get(minY).x` are true for the second point, `Point(0, 1)`, so `minY` is updated to index 1 (since `0 < 1`).

3. For the test set you have just checked can you find a mutation of the code (i.e. the deletion, change or insertion of some code) that will result in failure but is not detected by your test set. You have to use the mutation testing tool.

1). Deletion Mutation

Remove the assignment of minY to 0 at the beginning of the method

```
public class ConvexHull {
  public void doGraham(Vector<Point> p) {
     if (p.size() == 0) {
        return;
     }
```

```
        int minY = 0;
        for (int i = 1; i < p.size(); i++) {
            if (p.get(i).y < p.get(minY).y ||
                (p.get(i).y == p.get(minY).y && p.get(i).x < p.get(minY).x)) {
                minY = i;
            }
        }

        Point temp = p.get(0);
        p.set(0, p.get(minY));
        p.set(minY, temp);
    }
}
```

Impact:This modification results in `minY` potentially being accessed without initialization, leading to unpredictable behavior. Your test cases currently do not verify that `minY` is properly initialized, which may allow faults to go unnoticed.

2) Insertion Mutation: Insert a line that overrides minY incorrectly based on a condition that should not occur.

```
public class ConvexHull {
    public void doGraham(Vector<Point> p) {
        if (p.size() == 0) {
            return;
        }

        int minY = 0;
        if (p.size() > 1) {
            minY = 1;
        }

        for (int i = 1; i < p.size(); i++) {
            if (p.get(i).y < p.get(minY).y ||
                (p.get(i).y == p.get(minY).y && p.get(i).x < p.get(minY).x)) {
                minY = i;
            }
        }

        Point temp = p.get(0);
```

```
        p.set(0, p.get(minY));
        p.set(minY, temp);
    }
}
```

3) Modification Mutation: Change the logical operator from | to && in the conditional statement

```
public class ConvexHull {
    public void doGraham(Vector<Point> p) {
        if (p.size() == 0) {
            return;
        }

        int minY = 0;
        for (int i = 1; i < p.size(); i++) {
            if (p.get(i).y < p.get(minY).y ||
                (p.get(i).y == p.get(minY).y && p.get(i).x < p.get(minY).x)) {
                minY = i;
            }
        }

        Point temp = p.get(0);
        p.set(0, p.get(minY));
        p.set(minY, temp);
    }
}
```

Analyzing Detection by Test Cases:

Statement Coverage:

- Potential Issue: Failing to initialize minY properly might not be detected in testing, as it may not cause an immediate error or exception. If the surrounding logic does not explicitly depend on minY being initialized, the lack of initialization might not be caught by the test cases, allowing this issue to pass unnoticed.

Branch Coverage:

- Potential Issue: Changing the code to force minY to a specific value (such as 1) could yield incorrect outcomes. However, if there are no targeted tests that check the positions

of points after execution, this issue might not be detected. The test suite may not thoroughly verify if the value of minY is accurate following changes in logic.

Basic Condition Coverage:

- Potential Issue: Modifying the logical operator from || to && may not produce an error or failure directly. However, the test cases may not confirm if minY is updated correctly under these modified conditions. Consequently, the impact of this change might go unnoticed if the tests do not specifically assess whether the altered condition affects the correctness of the minY value.

4. Create a test set that satisfies the path coverage criterion where every loop is explored at least zero, one or two times.

Test Case 1: No Loop Execution
Input: An empty vector p.
Test: `Vector<Point> p = new Vector<Point>();`
Expected Result: The method should exit immediately without executing the loop. This case verifies the behavior when the vector has zero elements, leading to the loop being bypassed entirely.

Test Case 2: Single Loop Iteration
Input: A vector containing one point.
Test: `Vector<Point> p = new Vector<Point>(); p.add(new Point(0, 0));`
Expected Result: The method should not execute the loop since the vector size is one. The lone point should remain unchanged after a redundant swap, confirming that the method behaves correctly when the loop is only set to iterate once.

Test Case 3: Two Iterations in the Loop
Input: A vector with two points, where the first point has a greater y-coordinate than the second.
Test: `Vector<Point> p = new Vector<Point>(); p.add(new Point(1, 1)); p.add(new Point(0, 0));`
Expected Result: The method will enter the loop, compare the two points, and

determine that the second point has a lower y-coordinate. Consequently, the index for the minimum y-coordinate (`minY`) will be updated to 1, and the points will be swapped, placing the second point at the beginning of the vector. This case ensures that the loop operates twice.

Test Case 4: Multiple Loop Iterations
Input: A vector containing multiple points.
Test: `Vector<Point> p = new Vector<Point>(); p.add(new Point(2, 2)); p.add(new Point(1, 0)); p.add(new Point(0, 3));`
Expected Result: The loop should iterate through all three points. During the iterations, the second point will have the smallest y-coordinate, leading to an update of `minY` to 1. After the swap, the second point with coordinates (1, 0) should occupy the front position in the vector. This test confirms that the loop performs multiple iterations and updates accurately.

Lab Execution:

1. After generating the control flow graph, check whether your CFG match with the CFG generated by Control Flow Graph Factory Tool and Eclipse flow graph generator.

Control Flow Graph Factory :- YES
Eclipse flow graph generator :- YES

2. Devise the minimum number of test cases required to cover the code using the aforementioned criteria.

Statement Coverage: 3 test cases
Branch Coverage: 4 test cases
Basic Condition Coverage: 4 test cases
Path Coverage: 3 test cases
Summary of Minimum Test Cases:
Total: 3 (Statement) + 4 (Branch) + 4 (Basic Condition) + 3 (Path) = 14 test cases

Q3) and Q4) Same as Part I