IT-314: Software Engineering

Lab Assignment: 9 – Mutation Testing

Name: Darshak Kukadiya
Id: 202201180
Group Number: 8

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.

```
// Define the Point class
class Point {
    int x, y;

    // Constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // Representation method
    @Override
    public String toString() {
        return "Point(x=" + x + ", y=" + y + ")";
    }
}

// Define the doGraham function
public class GrahamAlgorithm {

    public static Point doGraham(Point[] points) {
        int minIdx = 0;

        // Find the point with the minimum y-coordinate
        for (int i = 1; i < points.length; i++) {
            if (points[i].y < points[minIdx].y) {
                minIdx = i;
```

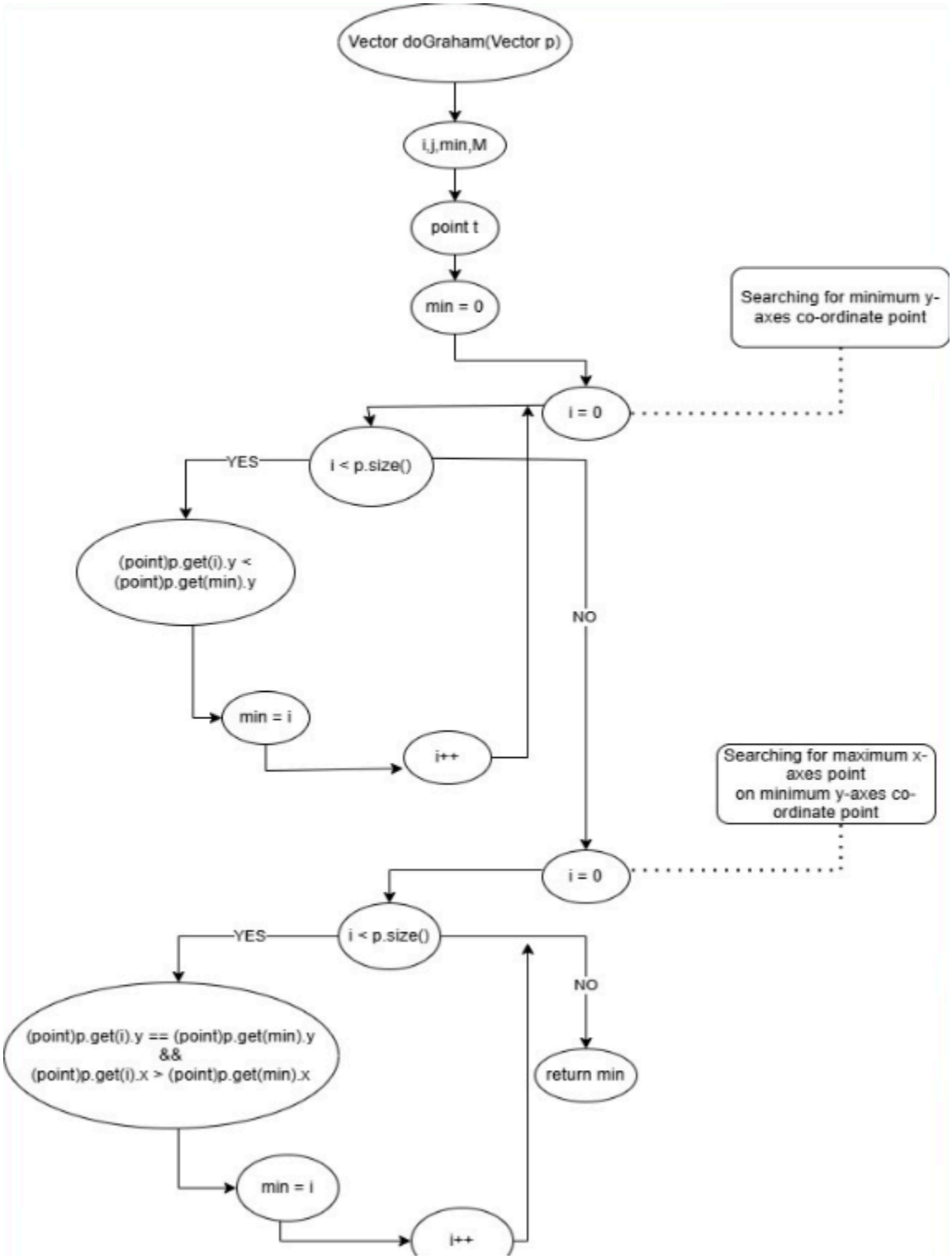
```
        }
    }

    // If there are points with the same y-coordinate, choose the one with the minimum x-coordinate
    for (int i = 0; i < points.length; i++) {
        if (points[i].y == points[minIdx].y && points[i].x > points[minIdx].x) {
            minIdx = i;
        }
    }

    // Returning the identified minimum point for clarity
    return points[minIdx];
  }

  public static void main(String[] args) {
    // Test the code
    Point[] points = { new Point(1, 2), new Point(3, 4), new Point(2, 1), new Point(5, 0) };
    System.out.println("Minimum point: " + doGraham(points));
  }
}
```

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



2. Construct test sets for your flow graph that are adequate for the following
criteria:
a. Statement Coverage.
b. Branch Coverage.
c. Basic Condition Coverage.

```
// Define the Point class
class Point {
    int x, y;

    // Constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    // Representation method
    @Override
    public String toString() {
        return "Point(x=" + x + ", y=" + y + ")";
    }
}

// Define the GrahamAlgorithm class
public class GrahamAlgorithm {

    // Method to find the point with the minimum y-coordinate (or minimum x if
y's are equal)
    public static Point doGraham(Point[] points) {
        int minIdx = 0;
```

```
        // Find the point with the minimum y-coordinate
        for (int i = 1; i < points.length; i++) {
            if (points[i].y < points[minIdx].y) {
                minIdx = i;
            }
        }

        // If there are points with the same y-coordinate, choose the one with the minimum x-coordinate
        for (int i = 0; i < points.length; i++) {
            if (points[i].y == points[minIdx].y && points[i].x > points[minIdx].x) {
                minIdx = i;
            }
        }

        // Returning the identified minimum point for clarity
        return points[minIdx];
    }

    // Test cases to verify the doGraham function
    public static void runTests() {
        Point[][] testCases = {
            // Test case 1 - Statement Coverage
            { new Point(2, 3), new Point(1, 2), new Point(3, 1) },

            // Test cases for Branch Coverage
            { new Point(2, 3), new Point(1, 2), new Point(3, 1) }, // Branch True in both conditions
            { new Point(3, 3), new Point(4, 3), new Point(5, 3) }, // Branch False in both conditions

            // Test cases for Basic Condition Coverage
```

```
        { new Point(2, 3), new Point(1, 2), new Point(3, 1) }, // p[i].y < p[min_idx].y
is True
        { new Point(1, 3), new Point(2, 3), new Point(3, 3) }, // p[i].y < p[min_idx].y
is False
        { new Point(2, 2), new Point(1, 2), new Point(3, 2) }, // p[i].y ==
p[min_idx].y is True, p[i].x < p[min_idx].x is True
        { new Point(3, 2), new Point(4, 2), new Point(2, 2) }  // p[i].y ==
p[min_idx].y is True, p[i].x < p[min_idx].x is False
    };

    // Run each test case
    for (int i = 0; i < testCases.length; i++) {
        Point minPoint = doGraham(testCases[i]);
        System.out.println("Test Case " + (i + 1) + ": Input Points = " +
java.util.Arrays.toString(testCases[i]) + ", Minimum Point = " + minPoint);
    }
  }

  // Main method to execute the tests
  public static void main(String[] args) {
    runTests();
  }
}
```

Output:

```
Test Case 1: Input Points = [Point(x=2, y=3), Point(x=1, y=2), Point(x=3, y=1)], Minimum Point = Point(x=3, y=1)
Test Case 2: Input Points = [Point(x=2, y=3), Point(x=1, y=2), Point(x=3, y=1)], Minimum Point = Point(x=3, y=1)
Test Case 3: Input Points = [Point(x=3, y=3), Point(x=4, y=3), Point(x=5, y=3)], Minimum Point = Point(x=5, y=3)
Test Case 4: Input Points = [Point(x=2, y=3), Point(x=1, y=2), Point(x=3, y=1)], Minimum Point = Point(x=3, y=1)
Test Case 5: Input Points = [Point(x=1, y=3), Point(x=2, y=3), Point(x=3, y=3)], Minimum Point = Point(x=3, y=3)
Test Case 6: Input Points = [Point(x=2, y=2), Point(x=1, y=2), Point(x=3, y=2)], Minimum Point = Point(x=3, y=2)
Test Case 7: Input Points = [Point(x=3, y=2), Point(x=4, y=2), Point(x=2, y=2)], Minimum Point = Point(x=4, y=2)
```

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.

a. Deletion Mutation:

```
// Original
if (points[i].y < points[minIdx].y) {
   minIdx = i;
}

// Mutated - deleted the condition check
        minIdx = i
```

Analysis for Statement Coverage:

- **Impact of Removing the Condition Check:** If the condition check is removed, the code will assign i to min on every iteration. This would likely result in an incorrect final value for min, as the point with the smallest y value would not necessarily be selected. However, this issue could go undetected if test cases only verify that min was assigned a value, without specifically checking that it represents the correct minimum y value.
- **Risk of Undetected Errors:** If the test cases merely confirm that min has been assigned without verifying that it holds the correct minimum value, any errors in the selection of the minimum could remain unnoticed

b. Change Mutation:

```
// Original
```

```
if (points[i].y < points[minIdx].y)

// Mutated - changed < to <=
if (points[i].y <= points[minIdx].y)
```

Analysis for Branch Coverage:

- **Effect of Mutation:** Changing < to <= could cause the code to assign min = i even when p[i].y is equal to p[min_idx].y, potentially selecting the wrong point as the minimum. This may lead to incorrect results if points with equal y values are handled incorrectly.
- **Risk of Undetected Faults**: If the test cases do not explicitly check scenarios where p[i].y equals p[min_idx].y, this mutation may introduce a subtle error that goes unnoticed, as the faulty behavior only occurs under specific conditions.

c. Insertion Mutation:

```
// Original
 min_idx = i

// Mutated - added unnecessary increment
 min_idx = i + 1
```

Analysis for Basic Condition Coverage:

- **Effect of Adding an Unnecessary Increment (i + 1):** Introducing an unnecessary increment (i + 1) alters the intended assignment, causing min to potentially point to an incorrect index. This could even result in an out-of-bounds error if min is set to an index beyond the array length.

- **Risk of Undetected Faults:** If the test cases do not explicitly verify that min is assigned to the correct index without any additional increments, this mutation might go unnoticed. Tests that simply confirm that min has been assigned, without validating its correctness, may miss this error.

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: Loop Explored Zero Times

- **Input:** An empty vector p.
- **Test:**

  Vector<Point> p = new Vector<Point>();

- **Expected Result:** The method should return immediately without any processing, as the vector size is zero. This tests the condition where the loop does not execute at all due to an empty vector.

Test Case 2: Loop Explored Once

- **Input:** A vector containing a single point.
- **Test:**
  Vector<Point> p = new Vector<Point>();

  p.add(new Point(0, 0));

- **Expected Result**: The loop should not execute since p.size() is 1. The method should swap the first point with itself, leaving the vector unchanged. This case verifies behavior when there is only one element in the vector.

Test Case 3: Loop Explored Twice

- **Input:** A vector with two points, where the first point has a higher 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 loop should compare the two points and identify that the second point has a lower y-coordinate. The minimum index (minY) should be updated to 1, and a swap should place the second point at the start of the vector. This case covers a two-iteration loop with a necessary swap.

Test Case 4: Loop Explored More Than Twice

- **Input:** A vector with 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 over all three points. The second point, with the lowest y-coordinate, will update minY to 1. A swap will move the point (1, 0) to the front of the vector. This case tests a multi-iteration loop to find and move the minimum y-coordinate point.

Lab Execution

Q1). After generating the control flow graph, check whether your CFG matches with the CFG generated by Control Flow Graph Factory Tool and Eclipse flow graph generator.

-> Control Flow Graph Factory: Yes

Q2).Devise minimum number of test cases required to cover the code using the aforementioned criteria.

-> Statement Coverage: 3 test cases

1. Branch Coverage: 3 test cases
2. Basic Condition Coverage: 3 test cases
3. Path Coverage: 3 test cases

-> Summary of Minimum Test Cases:

Total: 3 (Statement) + 3 (Branch) + 2 (Basic Condition) + 3 (Path) = 11 test Cases