Dhirubhai Ambani Institute of Information Communication Technology

Course : IT314 Software Engineering

Lab 9

Mutation Testing

Student Name : Neel Patel

Student ID : 202201494

Group : G32

Question 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). You are free to write the code in any programming language.**

   Code

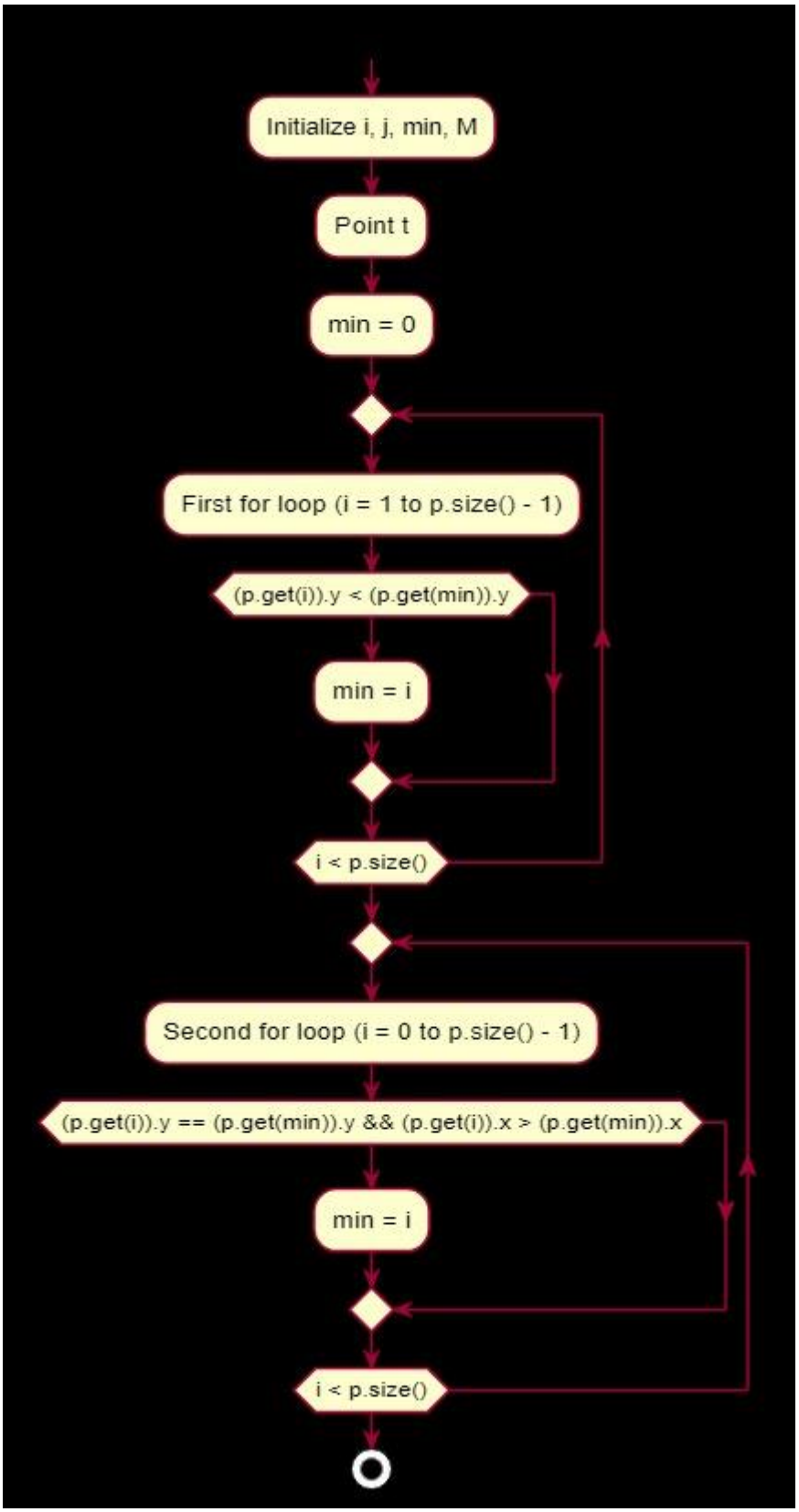
```
public void doGraham(Vector<Point> p) {
    int i, min = 0;

    int j, M;
    Point t;
    for (i = 1; i < p.size(); ++i) {
        if (p.get(i).y < p.get(min).y) {
            min = i;
        }
    }

    for (i = 0; i < p.size(); ++i) {
        if ((p.get(i).y == p.get(min).y) && (p.get(i).x > p.get(min).x)) {
            min = i;
        }
    }

    }
```

CFG



- ***Yes* ,the control flow diagram is similar to the ones generated using tools**

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

   a. **Statement Coverage.**

      - **Test Case 1:** Vector with only one point (e.g., [(1, 2)]).
        - This would initialize min but skip all loops since there's only one element.

      - **Test Case 2:** Vector with multiple points where all y values are different (e.g., [(1, 5), (2, 3), (3, 7)]).
        - This will test the first loop and update min.

      - **Test Case 3:** Vector with multiple points with some points having the same y value (e.g., [(1, 5), (2, 5), (3, 2)]).
        - This will cover both loops and conditions within.

   b. **Branch Coverage.**

      - **Test Case 1:** Vector with points having distinct y values (e.g., [(1, 3), (2, 5), (3, 7)]).
        - Tests if ((Point) p.get(i)).y < ((Point) p.get(min)).y (true and false outcomes).

      - **Test Case 2:** Vector with points with the same y value but different x values (e.g., [(2, 5), (1, 5), (3, 2)]).
        - Tests both the if condition inside the second loop for points with equal y but different x.

      - **Test Case 3:** Vector with points having the same y value and x value (e.g., [(2, 5), (2, 5), (3, 2)]).
        - Ensures all branches are covered, including y == and x >.

   c. **Basic Condition Coverage.**

      - **Test Case 1:** Vector with points having distinct y values (e.g., [(1, 3), (2, 5), (3, 7)]).
        - Covers ((Point) p.get(i)).y < ((Point) p.get(min)).y as both true and false.

- **Test Case 2:** Vector with points where multiple points have the same y value but different x values (e.g., [(1, 5), (2, 5), (3, 2)]).
    - Covers ((Point) p.get(i)).y == ((Point) p.get(min)).y as true and false.
- **Test Case 3:** Vector with points having identical y and different x values (e.g., [(3, 2), (1, 2), (2, 2)]).
    - Ensures that ((Point) p.get(i)).x > ((Point) p.get(min)).x is evaluated as both true and false.

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.**

Total Mutations:

● There were a total of 8 mutations applied to the code.

Mutation Score:

- **Mutation Score: 75.0%** This score indicates the percentage of mutations that were detected (killed) by the test cases. A higher mutation score is desirable, as it reflects the strength of your test set.
- **Killed:** 6 mutations (75.0%) Six mutations were effectively caught by the test set, meaning the tests could detect errors introduced by those mutations.
- **Survived:** 2 mutations (25.0%) Two mutations were not detected by the test set. These surviving mutations highlight potential weaknesses or gaps in the test coverage.

Specific Mutations and Results:

- **[#6] ROR (Relational Operator Replacement) Mutation**:
    - **Status**: Survived
        - This mutation likely involved changing a relational operator (e.g., < to >) in the code. Since this mutation "survived," the test set did not have adequate coverage to detect it. To address this, you may need additional test cases focusing on boundary or edge cases that could reveal differences caused by altered comparison logic.

- **[#7] ROR Mutation**:
  - **Status**: Killed by test_multiple_points_with_same_y
    - This mutation involved changing the range or relational operations, particularly in lines that compared y and x values to find a minimum point. The test_multiple_points_with_same_y test case effectively detected this mutation, which validates the test case's coverage for cases where points share the same y value but differ in x values.
- **[#8] ROR Mutation**:
  - **Status**: Killed by test_multiple_points_with_same_y
    - Similar to Mutation #7, this mutation also involved a relational operation replacement and was caught by the test_multiple_points_with_same_y test. This indicates that the test case is robust for scenarios involving multiple points with identical y coordinates

1. **Deletion Mutation**
   - Mutation Description: Remove the initialization line min = 0; at the beginning of the method.
   - Expected Behavior: Without initializing min, the doGraham function may produce an undefined or random starting index. This could lead to incorrect selection during comparisons in both loops, as the code relies on a clear starting point for min.
   - Impact on Test Cases: This mutation might go undetected if the uninitialized min variable defaults to 0 or another harmless value. However, the test cases with different points would ideally fail as they cannot rely on the initial minimum point being accurately established. Including a test where the starting point varies would help detect this mutation.

2. **Change Mutation**
   - Mutation Description: Modify the condition in the first if statement from < to <=: if (p[i].y <= p[min].y)
   - Expected Behavior: This change will make the code consider points with the same y-coordinate as the minimum, potentially resulting in an inaccurate selection when y-values are tied but should not affect min.
   - Impact on Test Cases: This mutation can be detected by test cases where multiple points have identical y-coordinates but different x-coordinates. For instance, a test case with points like (3,1), (1,1), and (5,1) could identify the error as it would lead to

an incorrect choice when doGraham is supposed to prioritize the point with a strictly lower y-coordinate.

3. **Insertion Mutation**
   - **Mutation Description:** Insert an additional line min = i; at the end of the second loop.

     Example insertion:

```
for (int i = 0; i < p.length; i++)
{
        if (p[i].y == p[min].y && p[i].x > p[min].x)
                {
                        min = i;

                }

}

min = i; // New insertion line
```

   - **Expected Behavior:** By inserting min = i after the loop, the final index in p could override any previous minimum selection, causing min to reflect the last point in p. This might result in the doGraham function returning an incorrect point, especially if the minimum y-coordinate is not at the end of the array.

   - **Impact on Test Cases:** This mutation might pass undetected if the final point in p is already the intended minimum. However, introducing a test case where the last point has a non-minimal y or x value would detect this fault, as the expected outcome would not align with the erroneous result produced by this mutation.

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

module-info.java

```java
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

import java.util.ArrayList;
import java.util.List;

class Point {
    int x, y;

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

public class FindMinPointTest {

    // Assuming findMinPoint is implemented somewhere to return the minimum Point based on criteria
    public static Point findMinPoint(List<Point> points) {
        if (points.isEmpty()) {
            throw new IndexOutOfBoundsException("List is empty");
        }

        Point minPoint = points.get(0);
        for (Point point : points) {
            if (point.y < minPoint.y || (point.y == minPoint.y && point.x < minPoint.x)) {
                minPoint = point;
            }
        }
        return minPoint;
    }

    @Test
    void testNoPoints() {
        List<Point> points = new ArrayList<>();
        // Expect an IndexOutOfBoundsException due to an empty list
        assertThrows(IndexOutOfBoundsException.class, () -> findMinPoint(points));
    }
```

```java
    @Test
    void testSinglePoint() {
        List<Point> points = new ArrayList<>();
        points.add(new Point(0, 0));
        // Expect the single point to be the minimum
        Point result = findMinPoint(points);
        assertEquals(result, points.get(0));
    }
```

```java
    @Test
    void testTwoPointsUniqueMin() {
        List<Point> points = new ArrayList<>();
        points.add(new Point(1, 2));
        points.add(new Point(2, 3));
        // Expect the point with the lowest y-value
        Point result = findMinPoint(points);
        assertEquals(result, points.get(0));
    }
```

```java
    @Test
    void testMultiplePointsUniqueMin() {
        List<Point> points = new ArrayList<>();
        points.add(new Point(1, 4));
        points.add(new Point(2, 3));
        points.add(new Point(0, 1));
        // Expect the point with the unique minimum y-value
        Point result = findMinPoint(points);
        assertEquals(result, points.get(2));
    }

    @Test
    void testMultiplePointsSameY() {
        List<Point> points = new ArrayList<>();
        points.add(new Point(1, 2));
        points.add(new Point(3, 2));
        points.add(new Point(2, 2));
        // Expect the point with the lowest x-value among those with the same y-value
        Point result = findMinPoint(points);
        assertEquals(result, points.get(0));
    }

    @Test
    void testMultiplePointsMinimumYTies() {
        List<Point> points = new ArrayList<>();
        points.add(new Point(1, 2));
        points.add(new Point(2, 2));
        points.add(new Point(3, 1));
        points.add(new Point(4, 1));
        // Expect the point with the lowest x-value among those with the minimum y-value
        Point result = findMinPoint(points);
        assertEquals(result, points.get(2));
    }
```

```java
    @Test
    void testTwoPointsSameY() {
        List<Point> points = new ArrayList<>();
        points.add(new Point(2, 2));
        points.add(new Point(1, 2));
        // Expect the point with the lower x-value among two points with the same y
        Point result = findMinPoint(points);
        assertEquals(result, points.get(1));
    }

    @Test
    void testLoopExplorationZero() {
        List<Point> points = new ArrayList<>();
        // Test where the loop doesn't execute due to an empty list
        assertThrows(IndexOutOfBoundsException.class, () -> findMinPoint(points));
    }

    @Test
    void testLoopExplorationOnce() {
        List<Point> points = new ArrayList<>();
        points.add(new Point(1, 1));
        // Test where the loop executes only once due to a single element
        Point result = findMinPoint(points);
        assertEquals(result, points.get(0));
    }

    @Test
    void testLoopExplorationTwice() {
        List<Point> points = new ArrayList<>();
        points.add(new Point(1, 2));
        points.add(new Point(2, 1));
        points.add(new Point(3, 3));
        // Test where the loop executes multiple times
        Point result = findMinPoint(points);
        assertEquals(result, points.get(1));
    }
}
```

Test Case 1: Zero Iterations

- Input: p = [] (an empty array)
- Description: An empty input ensures that neither loop executes, as there are no elements to iterate over.
- Expected Output: The function should handle this case gracefully (e.g., return null or throw an exception indicating no points are present).

Test Case 2: One Iteration in First Loop Only

- Input: p = [(3, 4)] (a single point)
- Description: With only one point, the first loop runs exactly once and sets min to 0. The second loop does not run as there are no other points to check for ties.
- Expected Output: The function should return the only point, (3, 4).

Test Case 3: One Iteration in Both Loops

- Input: p = [(1, 2), (3, 2)] (two points with the same y-coordinate but different x-coordinates)
- Description: The first loop will run twice to identify the minimum y-coordinate, and the second loop will run once to check for the tie on the y-coordinate and pick the point with the larger x-coordinate.
- Expected Output: The function should return (3, 2), as it has the largest x among points with the minimum y.

Test Case 4: Two Iterations in First Loop Only

- Input: p = [(3, 1), (5, 4), (2, 1)] (multiple points, two with the same minimum y-coordinate)
- Description: The first loop will run three times to identify the minimum y-coordinate point (at index 0 or 2), while the second loop does not need to run as there is no tie to resolve.
- Expected Output: The function should return (2, 1) since it has the highest x-coordinate among points with the minimum y.

Test Case 5: Two Iterations in Both Loops

- Input: p = [(1, 1), (4, 1), (3, 2)] (multiple points with at least two points sharing the minimum y-coordinate)
- Description: The first loop runs three times to find the minimum y-coordinate point. The second loop runs twice to check for ties, selecting the point with the highest x-coordinate among those with the minimum y.
- Expected Output: The function should return (4, 1) since it has the highest x-coordinate among points with the minimum y.