IT314

Software Engineering

Lab 9

Mutation Testing

Name: Maulik Kansara

StudentId: 202201442

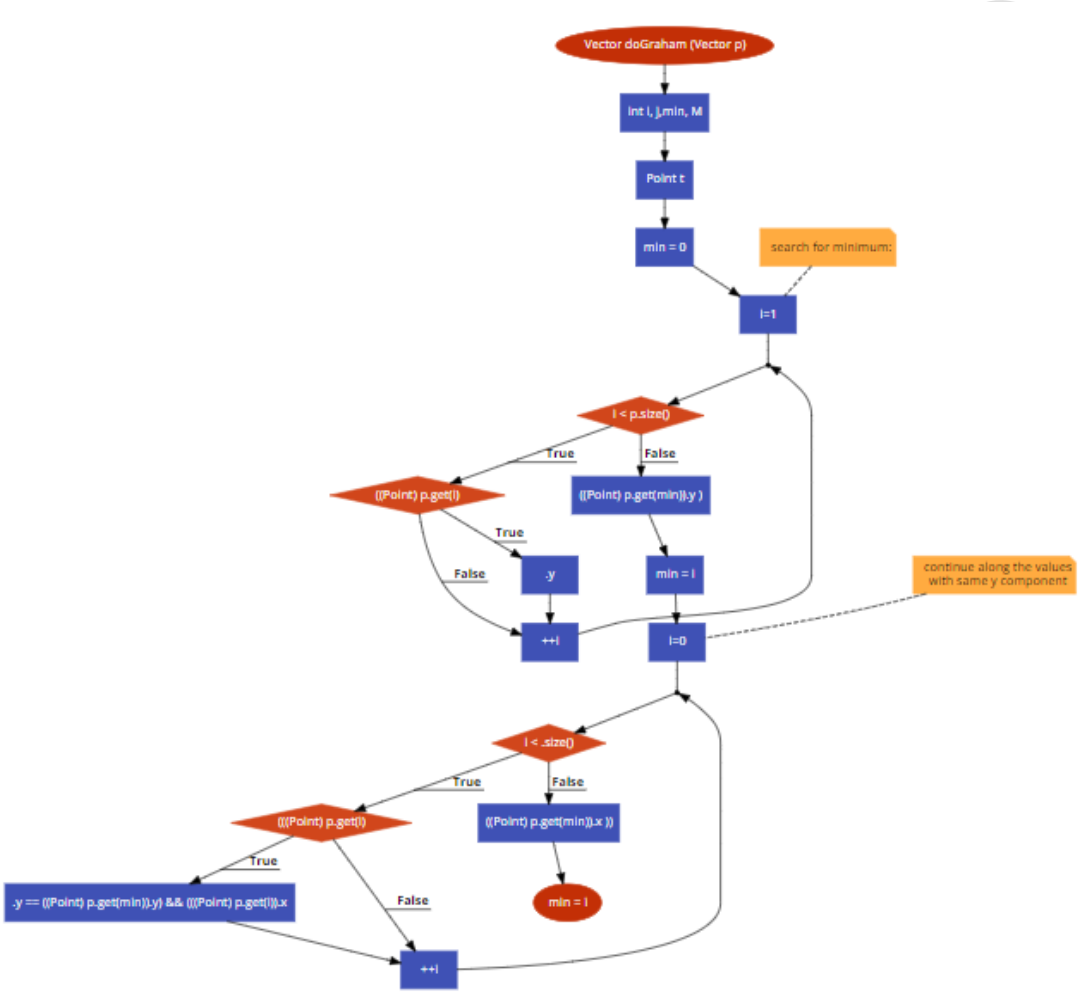
Lab Group: Group 5

Q.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;
            }
        }
```

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.**

   Control Flow Graph:



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. (In your submission document, mention only "Yes" or "No" for each tool).

Yes.

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.

a. Statement Coverage

For Statement Coverage, we must ensure that every line of code is executed at least once.

Test Set:

- Single Point Case: Test with a vector p containing only one point (to ensure the method doesn't fail on edge cases). This will ensure the initialization and first loop's condition is checked, though no iterations occur.
  Input: p = [(0, 0)]
- Multiple Points with Unique Y-Values: Test with multiple points where all points have unique y-coordinates. This ensures the first loop executes fully but the second loop does not alter min because there are no ties in the y-coordinates.
  Input: p = [(1, 3), (2, 1), (3, 4), (4, 5)]
- Multiple Points with Tied Y-Values: Test with points where two or more points have the same minimum y-coordinate to activate the second loop.
  Input: p = [(1, 3), (2, 1), (4, 1), (5, 5)]

b. Branch Coverage.

For Branch Coverage, we must ensure that each branch (true and false paths of each condition) is executed.

Test Set:

- Multiple Points with One Minimum Y-Value: Covers both true and false cases for the condition in the first loop.
  Input: p = [(1, 4), (2, 1), (3, 3)]
- Multiple Points with Tied Minimum Y-Values and Different X-Values: Ensures the second loop condition is evaluated as both true and false.
  Input: p = [(1, 2), (3, 1), (5, 1)]
- Multiple Points with No Ties: Checks false branches in both loops.
  Input: p = [(1, 2), (3, 3), (4, 4)]

c. Basic Condition Coverage

For Basic Condition Coverage, each basic condition within a compound condition should be tested as both true and false independently.

Test Set:

- Points with Different Y-Values: Ensures both true and false evaluations of (p.get(i)).y < (p.get(min)).y).
  Input: p = [(2, 2), (3, 1), (4, 3)]
- Points with Same Y and Different X Values: Ensures the second condition evaluates to true and false.
  Input: p = [(2, 1), (4, 1), (1, 1)]
- Points with No Matching Y-Values: Ensures that both conditions in the second loop evaluate to false.
  Input: p = [(1, 2), (3, 3), (5, 4)]

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

Minimum Test Set:

Test Case	Points Vector (p)	Coverage Criteria Satisfied
1	[(0, 0)]	Statement Coverage
2	[(1, 3), (2, 1), (3, 4), (4, 5)]	Statement Coverage, Branch Coverage, Basic Condition Coverage for first loop
3	[(1, 3), (2, 1), (4, 1), (5, 5)]	Statement Coverage, Branch Coverage, Basic Condition Coverage for second loop

The minimum number of test cases required to cover all three criteria is 3.

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.**

   This part of the exercise is very tricky and interesting. The test cases that you have derived in Step 2 are then used to identify the fault when you make some modifications in the code. Here, you need to insert/delete/modify a piece of code that will result in failure but it is not detected by your test set – derived in Step 2.

Deletion Mutation:

Remove the line min = 0;

Before:

```
def do_graham(self, points: List[Point]) -> List[Point]:
    min = 0
    # search for minimum y-coordinate
    for i in range(1, len(points)):
        if points[i].y <= points[min_index].y:
            min = i
```

After:

```
def do_graham(self, points: List[Point]) -> List[Point]:
    #min = 0
    # search for minimum y-coordinate
    for i in range(1, len(points)):
        if points[i].y <= points[min_index].y:
            min = i
```

Result:

Without initializing min to 0, the code will try to access p.get(min) without a valid starting index, potentially causing a runtime error (e.g., NullPointerException or ArrayIndexOutOfBoundsException) if min is uninitialized or holds an invalid value. However, this error may not be caught by the minimal test set if all test cases contain multiple points, as min might be set during the first iteration of the first loop

Test Case to Detect Deletion Mutation:

- Input: p = [(0, 0)]

Insertion Mutation:

Insert a condition p.size() > 2 in the second loop so that it only executes if there are more than two points in p:

Before:

```
for i in range(len(points)):
    if (points[i].y == points[min_index].y and points[i].x > points[min_index].x ):
        min = i
```

After:

```
for i in range(len(points)):
    if (points[i].y == points[min_index].y and points[i].x > points[min_index].x and p.size()>2 ):
        min = i
```

Result:

Adding this condition will make the second loop ineffective if there are fewer than three points. This could cause the method to return an incorrect min value if there are only two points with the same y-coordinate but different x-coordinates.

Test Case to Detect Insertion Mutation:

- Input: p = [(1, 1), (2, 1)]

Modification Mutation:

Change the < operator to <= in the first loop:

Before:

```
for i in range(1, len(points)):
    if points[i].y < points[min_index].y:
        min = i
```

After:

```
for i in range(1, len(points)):
    if points[i].y <= points[min_index].y:
        min = i
```

Result:

This modification would alter the logic for selecting the minimum y-coordinate. Instead of keeping the first encountered point with the minimum y-value, it would keep the last one, potentially affecting the result if there are multiple points with the same minimum y-value.

Test Case to Detect Modification Mutation:

Input: p = [(1, 1), (3, 1), (2, 2)]

Expected behavior: The point (1, 1) should be chosen as the minimum, not (3, 1).

4. **Create a test set that satisfies the path coverage criterion where every loop is explored at least zero, one or two times .Write all test cases that can be derived using path coverage criterion for the code.**

Test Case 1: No Points (Zero Iterations)

- Input: [ ]
- Expected Output: [ ] (No points to process)
- Path Explanation: This case checks the path where the outer loop never executes because there are no points.

Test Case 2: One Point (Zero Iterations)

- Input: [(0, 0)]
- Expected Output: [(0, 0)] (Single point forms a trivial convex hull)
- Path Explanation: The outer loop runs once but does not enter any condition that alters the output significantly.

Test Case 3: Two Points (One Iteration)

- Input: [(1, 2), (2, 3)]
- Expected Output: [(1, 2), (2, 3)] (Line segment forms a convex hull)
- Path Explanation: This path checks the outer loop iterating once, checking the orientation only once.

Test Case 4: Three Points (One Iteration with Orientation Check)

- Input: [(0, 0), (1, 1), (2, 0)]
- Expected Output: [(0, 0), (2, 0)] (This forms a convex hull; middle point is inside)
- Path Explanation: The outer loop iterates through all points, and the inner orientation check eliminates one point.

Test Case 5: Four Points (Two Iterations with Orientation Checks)

- Input: [(0, 0), (2, 2), (2, 0), (0, 2)]

- Expected Output: [(0, 0), (2, 0), (2, 2), (0, 2)] (All points are part of the convex hull)
- Path Explanation: This path involves multiple iterations of both loops, covering cases where the orientation checks keep points in or out of the hull.

Test Case 6: Five Points (Two Iterations with Rejections)

- Input: [(0, 0), (1, 1), (2, 2), (1, 0), (0, 1)]
- Expected Output: [(0, 0), (1, 0), (2, 2), (0, 1)] (Middle point eliminated)
- Path Explanation: This case examines multiple iterations and how orientation influences hull composition.

Test Case 7: More Complex Shape (Multiple Iterations)

- Input: [(0, 0), (3, 1), (1, 2), (2, 1), (1, 0), (2, 0), (3, 0)]
- Expected Output: [(0, 0), (3, 1), (3, 0), (2, 0), (1, 0)] (Complex polygon shape)
- Path Explanation: This test checks how multiple iterations affect which points are included in the hull.