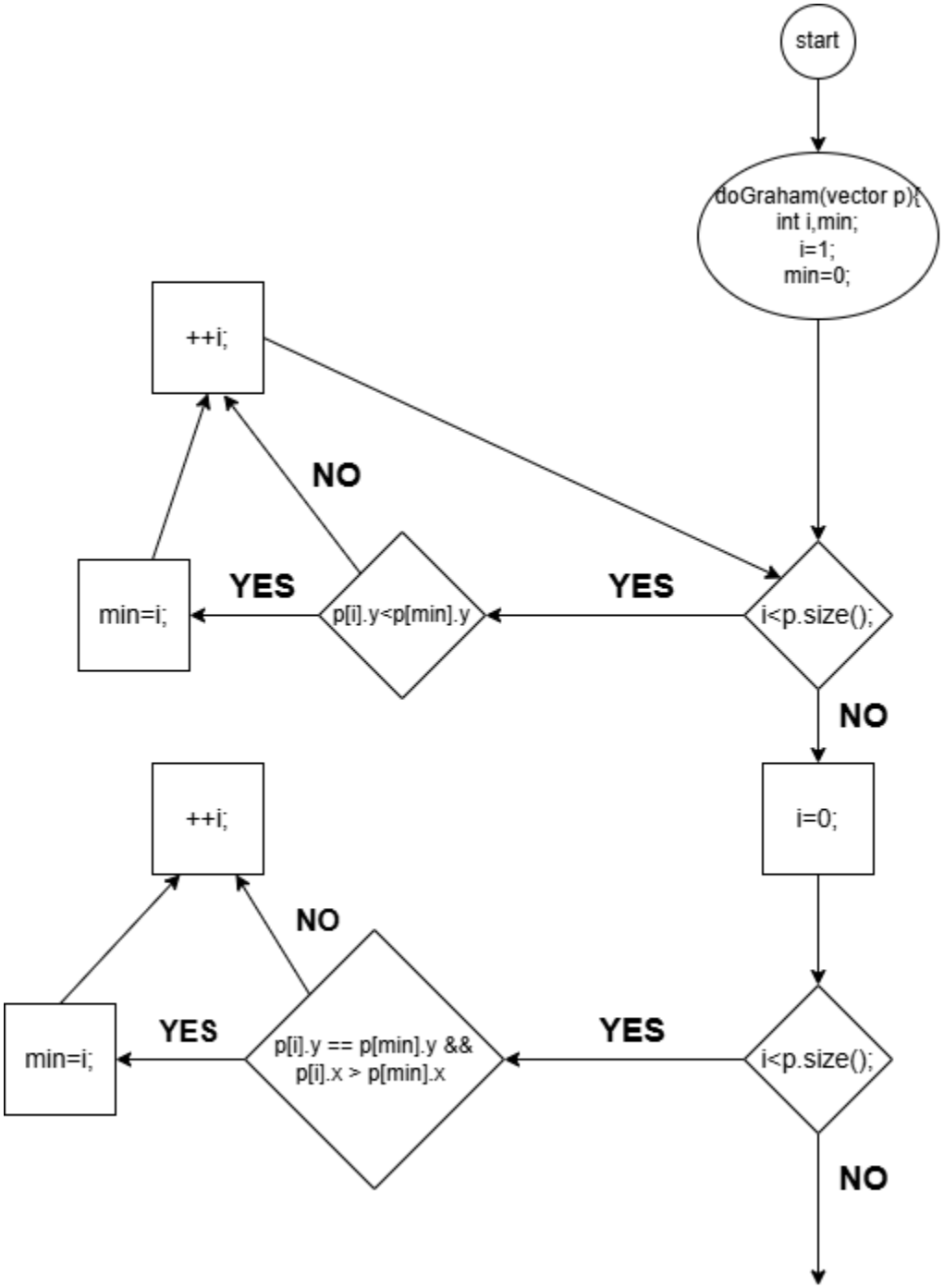
Software Engineering
Lab 9 – Mutation Testing
202201037
Rishi Patel

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



Code in C++

```cpp
#include <iostream>
#include <vector>

struct Point {
    int x, y;
};

// Function to find the rightmost point with the smallest y-coordinate
Point findRightmostLowestPoint(const std::vector<Point>& points) {
    int minIndex = 0;
    for (int i = 1; i < points.size(); ++i) {
        // Check for a lower y-coordinate, or if equal, a higher
x-coordinate
        if (points[i].y < points[minIndex].y ||
            (points[i].y == points[minIndex].y && points[i].x >
points[minIndex].x)) {
            minIndex = i;
        }
    }
    return points[minIndex];
}

int main() {
    // Example points
    std::vector<Point> points = {{3, 4}, {5, 2}, {2, 2}, {5, 2}, {1, 5}};

    // Find the rightmost point with the smallest y-coordinate
    Point rightmostLowest = findRightmostLowestPoint(points);
    std::cout << "Rightmost lowest point: (" << rightmostLowest.x << ", "
<< rightmostLowest.y << ")\n";

    return 0;
}
```

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

a. Statement Coverage.

1. **Test Case 1**: p.size() > 1, where p[i].y < p[min].y in at least one iteration of the loop.
    - This will cover the nodes involved in setting min = i and incrementing i within the first loop.

2. **Test Case 2**: p.size() > 1, where p[i].y == p[min].y but p[i].x > p[min].x in the second loop.
    - This will cover the conditions where both p[i].y == p[min].y and p[i].x > p[min].x are true, leading to an update of min in the second loop.

3. **Test Case 3**: p.size() = 1.
    - This simple case ensures execution of the initial checks and flow through without looping, covering the starting nodes and initial conditions.

These test cases ensure that all statements are executed at least once.

b. Branch Coverage

1. **Test Case 1**: p.size() > 1, where p[i].y < p[min].y for some i.
    - This ensures the true branch of p[i].y <= p[min].y is taken, updating min = i.

2. **Test Case 2**: p.size() > 1, where p[i].y > p[min].y for all i.
    - This will take the false branch in the first condition and skip updating min.

3. **Test Case 3**: p.size() > 1, where p[i].y == p[min].y and p[i].x <= p[min].x in the second loop.
    - This ensures both branches of p[i].y == p[min].y && p[i].x > p[min].x are covered in the second loop.

These test cases ensure that all branches are taken in both true and false forms.

c. Basic Condition Coverage

1. **Test Case 1**: p.size() > 1, where p[i].y < p[min].y.
    - Tests the condition p[i].y <= p[min].y as true and covers one condition.

2. **Test Case 2**: p.size() > 1, where p[i].y > p[min].y.
    - Tests the condition p[i].y <= p[min].y as false.

3. **Test Case 3**: p.size() > 1, where p[i].y == p[min].y and p[i].x > p[min].x.
    - Tests both p[i].y == p[min].y as true and p[i].x > p[min].x as true in the second loop.

4. **Test Case 4**: p.size() > 1, where p[i].y == p[min].y and p[i].x <= p[min].x.
    - Tests p[i].y == p[min].y as true and p[i].x > p[min].x as 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.

1. Code Deletion Mutation

Mutation by deleting code:
Remove the line that updates min in one of the loops, so min is never updated even when the condition is met.

Deletion:

```
if (p[i].y <= p[min].y) {
    // min = i;   // Deletion: Line to update `min` is removed
}
```

Explanation:

- This deletion would mean that min is never updated in the first loop, so the algorithm may fail to find the correct minimum point.
- This mutation might not be detected by the existing test cases if they do not rely on min being correctly updated, or if all points have the same y value, in which case min would stay at the initial value of 0.

2. Code Insertion Mutation

Mutation by inserting code:
Insert a line that modifies i in an unintended way, such as skipping an index by incrementing i again within one of the loops.

Insertion:

```
if (p[i].y <= p[min].y) {
    min = i;
    i++;  // Insertion: Extra increment of `i` inside the loop
}
```

Explanation:

- This insertion may cause the loop to skip certain points, which could lead to an incorrect result if the skipped point is the one with the minimum y value.
- This fault might not be caught by existing test cases if the input data is such that the skipped index doesn't affect the final outcome. For example, if there are many points with the same y values, skipping one of them may not change the minimum calculation.

3. Code Modification Mutation

Mutation by modifying code:
Change the condition to an incorrect comparison operator or modify the logic in such a way that it behaves incorrectly under certain conditions.

Modification:

```
if (p[i].y < p[min].y) {  // Modification: Changed `<=` to `<`
    min = i;
}
```

Explanation:

- This modification changes the condition from <= to <, meaning the algorithm now only updates min if p[i].y is strictly less than p[min].y, ignoring cases where p[i].y equals p[min].y.
- This fault might not be detected if the test cases do not include points with the same y values but different x values. If all points have unique y values, this change might not affect the outcome, so the fault could go unnoticed.

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 executed 0 times

- **Input**: p is an empty vector or has a single element, i.e., p.size() = 0 or p.size() = 1.
- **Expected Outcome**: The function should handle this gracefully, either doing nothing (if empty) or correctly processing a single element without attempting to loop.

This will ensure that both loops are skipped, covering the 0-times execution path.

Test Case 2: Loop executed 1 time

- **Input**: p has exactly 2 elements, i.e., p.size() = 2.
- **Expected Outcome**: The function should process both elements, and each loop should execute only once, comparing the two points in p.

This test case ensures that each loop goes through a single iteration, covering the 1-time execution path.

Test Case 3: Loop executed 2 times

- **Input**: p has exactly 3 elements, i.e., p.size() = 3.
- **Expected Outcome**: The function should process all three points in p, and each loop should iterate twice, comparing all pairs to determine the minimum value.

This test case ensures that each loop goes through two iterations, covering the 2-times execution path.