Name - Shourya Nahar

Student Id - 202201296

IT314: Lab 9

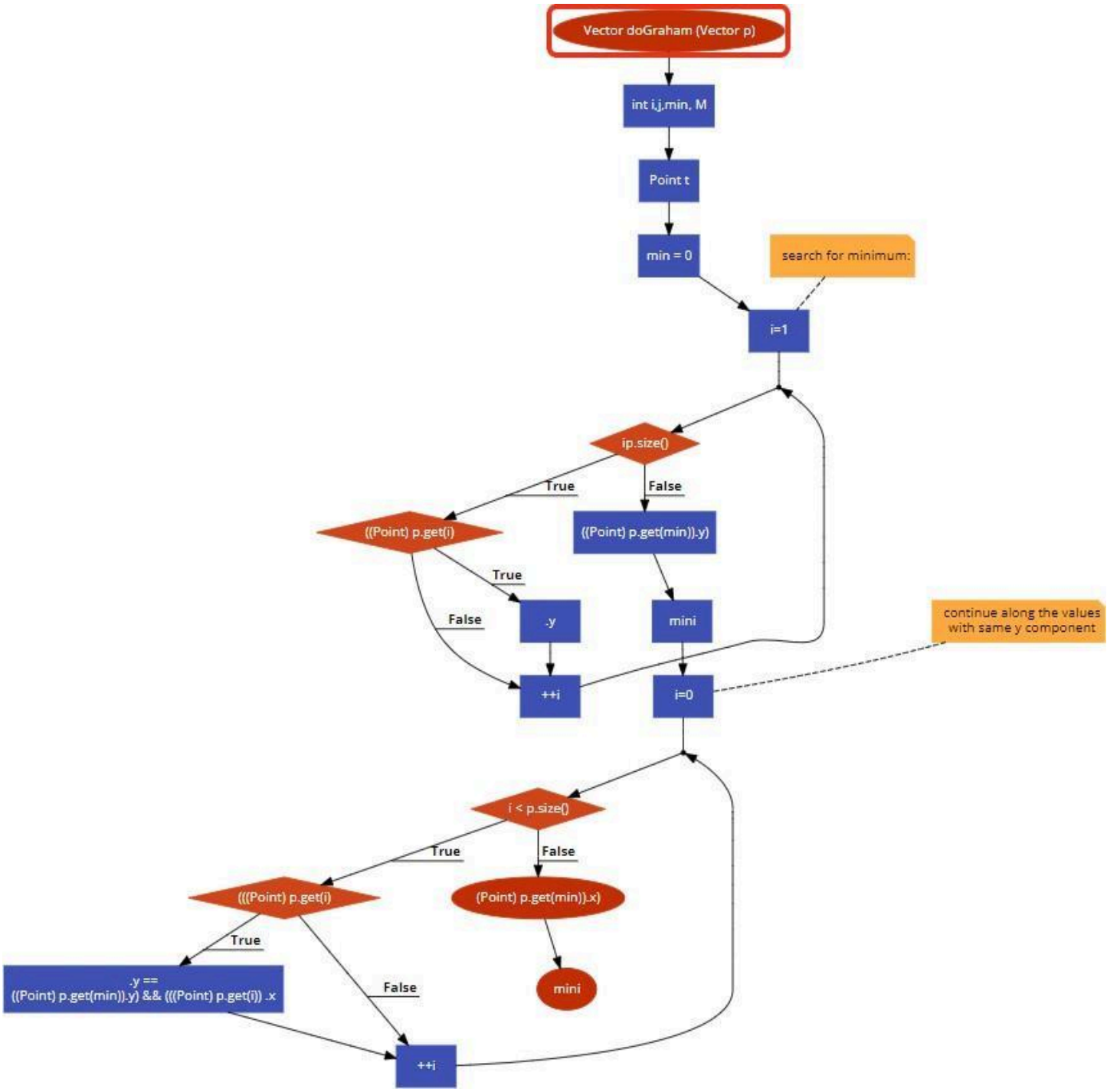
Mutation Testing

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.

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.

Control Flow Graph:



C++ Code

```
#include <vector>
#include <bits/stdc++.h>
using namespace std;

class Point {
public:
    int x, y;

    Point(int x = 0, int y = 0) : x(x), y(y) {}
};

class ConvexHull {
public:
    int doGraham(std::vector<Point>& p) {
        int i, j, min, M;
        min = 0;

        // Search for the point with the minimum y-coordinate
        for (i = 1; i < p.size(); ++i) {
            if (p[i].y < p[min].y) {
                min = i;
            }
        }

        // Continue along the points with the same y-coordinate
        for (i = 0; i < p.size(); ++i) {
            if ((p[i].y == p[min].y) && (p[i].x > p[min].x)) {
                min = i;
            }
        }

        // Return the index of the minimum point for verification
        return min;
    }
};

int main() {
    vector<Point> points = { {2, 3}, {1, 2}, {3, 2}, {2, 1}, {4, 2} };
```

```
    ConvexHull ch;
    int minIndex = ch.doGraham(points);
    cout << "Index of minimum point: " << minIndex << "\n";
    cout << "Minimum point coordinates: (" << points[minIndex].x << ", "
<< points[minIndex].y << ")\n";

    return 0;
}
```

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

a. Statement Coverage Test Cases for Statement Coverage:

Test Case 1: One point only. This ensures that the function can handle the minimal input case without iterating through the loops multiple times.

- **Input:** Points = (0,0)
- **Expected Output:** (0, 0)

Test Case 2: Multiple points with distinct y-coordinates. This tests the main logic of finding the minimum y-coordinate.

- **Input: Points =** (1,2),(3,4),(0,5)
- **Expected Output:** (1, 2)

Test Case 3: Multiple points with the same y-coordinate. This ensures that the function selects the point with the largest x-coordinate when y-coordinates are the same.

- **Input: Points =** (1,2),(3,2),(0,2)
- **Expected Output:** (3, 2)

Test Case 4: Points with identical coordinates. This tests if the function can handle multiple points that are exactly the same.

- **Input: Points =** (1,1)

- **Expected Output:** (1, 1) (any one of the identical points)

Test Case 5: Points with negative coordinates. This ensures the function correctly handles points in all quadrants.

- **Input: Points =** (2,−3),(−1,−4),(0,−5),(3,−2)
- **Expected Output:** (0, -5) (point with the smallest y-coordinate)

b. Branch Coverage

1. **Test Case 1:** Points with negative y-coordinates, ensuring the `if` condition in the first loop evaluates both `True` and `False`.
    - **Input: Points** = (1,−3),(0,−4),(2,−1)
    - **Expected Outcome:** Minimum point is (0, -4)
2. **Test Case 2:** Multiple points with the same y-coordinate and the same x-coordinate, ensuring both `if` conditions evaluate correctly even when points have identical coordinates.
    - **Input: Points** = (2,2),(2,2)
    - **Expected Outcome:** Minimum y-coordinate point is any of the points with y = 2, but the largest x should be selected, resulting in (3, 2)
3. **Test Case 3:** A mix of points where only one point has the minimum y-coordinate, ensuring both loops process fully and handle mixed cases.
    - **Input: Points** = (5,5),(6,3),(4,3),(7,2),(3,4)
    - **Expected Outcome:** Minimum point is (7, 2)

These cases are designed to thoroughly test each branch, ensuring that both `True` and `False` outcomes are evaluated in each conditional statement. This helps confirm that all possible branches in the code are covered.

c. Basic Condition Coverage

1. **Test Case 5:** All points have the same y-coordinate, but only one point has the minimum x-coordinate, ensuring p[i].y == p[min].y is True and p[i].x > p[min].x evaluates as False for one point.
    - **Input: Points =** (5,1),(3,1),(7,1)
    - **Expected Outcome:** Since all have the same y-coordinate, the point with the largest x is selected: (7, 1).
2. **Test Case 6:** Points have distinct y-coordinates, ensuring p[i].y < p[min].y evaluates `True` and `False` at different points in the first loop.
    - **Input: Points** = (2,4),(1,5),(3,3)
    - **Expected Outcome:** Minimum point is (3, 3).
3. **Test Case 7:** Points with identical y-coordinates but one point with a significantly larger x-coordinate, ensuring p[i].y == p[min].y is True and p[i].x > p[min].x is True for that point.
    - **Input: Points** = (2,0),(8,0),(5,0)
    - **Expected Outcome:** Since all points have the same y-coordinate, the one with the largest x is chosen: (8, 0).
4. **Test Case 8:** Points with a mix of negative y-coordinates and x- coordinates, testing both p[i].y < p[min].y and p[i].y == p[min].y && p[i].x > p[min].x in varied scenarios.
    - **Input: Points =** (−1,−2),(0,−1),(−3,−1)
    - **Expected Outcome:** Minimum point based on y-coordinate is (-1, -2). Since there's no point with the same y-coordinate, (-1, -2) is selected.

These cases are designed to ensure that each condition (p[i].y < p[min].y, p[i].y == p[min].y, and p[i].x > p[min].x) is evaluated as both True and False independently, thereby achieving Basic Condition Coverage for the code.

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.

```
#include <iostream>
#include <vector>
using namespace std;

class Point {
public:
    int x, y;
    Point(int x = 0, int y = 0) : x(x), y(y) {}
};

// Function to process points and find the one with the largest
x-coordinate among points with the minimum y-coordinate
vector<Point> doGraham(vector<Point>& p) {
    int minIdx = 0;

    // Find the point with the smallest y-coordinate. If there are
multiple, choose the one with the largest x-coordinate.
    for (int i = 1; i < p.size(); ++i) {
        if (p[i].y < p[minIdx].y || (p[i].y == p[minIdx].y && p[i].x >
p[minIdx].x)) {
            minIdx = i;
        }
    }

    // Swap the minimum point to the beginning of the vector
    if (minIdx != 0) {
        swap(p[0], p[minIdx]);
    }

    return p;
}

int main() {
    int n
```

```
    cout << "Enter the number of points: ";
    cin >> n;

    vector<Point> points;
    cout << "Enter the points (x y):\n";
    for (int i = 0; i < n; ++i) {
        int x, y;
        cin >> x >> y;
        points.push_back(Point(x, y));
    }

    points = doGraham(points);

    cout << "Processed points:\n";
    for (const auto& point : points) {
        cout << "(" << point.x << ", " << point.y << ")\n";
    }

    return 0;
}
```

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

Test Cases for Path Coverage

Test Case 1: Zero Iterations for Both Loops

- **Description:** No points are provided, so both loops have zero iterations.
- **Input: Points =** []
- **Expected Output:** Since there are no points, doGraham should return an empty vector or handle this differently (depending on implementation).

Test Case 2: Zero Iterations for the First Loop, One Iteration for the Second Loop

- **Description:** Only one point is provided, so only the second loop runs once.
- **Input: Points =** (5,5)
- **Expected Output:** Returns the same single point, (5,5).

Test Case 3: One Iteration for the First Loop, Two Iterations for the Second Loop

- **Description:** Two points are provided with the same y-coordinate but different x-coordinates, so the first loop iterates once to find the minimum based on the x-coordinate, and the second loop runs twice.
- **Input: Points =** (2,3), (4,3)
- **Expected Output:** Since both have the same y-coordinate, the point with the largest x is selected, (4,3).

Test Case 4: Two Iterations for the First Loop, Three Iterations for the Second Loop

- **Description:** Three points are provided with different y-coordinates, so the first loop iterates twice to find the minimum, and the second loop iterates thrice.
- **Input: Points =** (1,4), (2,2), (3,3)
- **Expected Output:** The minimum y-coordinate point is (2,2).

Lab Execution (how to perform the exercises): Use unit Testing framework, code coverage and mutation testing tools to perform the exercise.

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

Control Flow Graph Factory Tool - Yes
Eclipse flow graph generator - Yes

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

Test Case	Description	Input	Coverage	Expected Output
1	Zero iterations for both loops	[]	Path Coverage (zero iterations)	[]
2	One iteration for the second loop	(5,5)	Branch, Basic Condition, Path Coverage	(5,5)
3	One iteration for the first loop, two for the second	(2,3), (4,3)	Statement, Branch, Basic Condition, Path Coverage	(4,3)

4	Two iterations for the first loop, three for the second	(1,4), (2,2), (3,3)	Statement, Branch, Basic Condition, Path Coverage	(2,2)

3. 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.

Write/identify a mutation code for each of the three operation separately, i.e., by deleting the code, by inserting the code, by modifying the code.

Deleting the Code

```
for (int i = 1; i < p.size(); ++i) {
    if (p[i].y < p[min].y) {
        min = i;
    }
}
```

Inserting the Code

```
for (int i = 0; i < p.size(); ++i) {
    if (p[i].y == p[min].y && p[i].x > p[min].x) {
        min = i;
    }
    if (true) {  // Inserted redundant condition
        min = (min + 1) % p.size();     // Alters the `min`
value unexpectedly
    }
}
```

Modification of the Code

```
for (int i = 1; i < p.size(); ++i) {
    if (p[i].y <= p[min].y) {      // Modified from `<` to
        `<=` min = i;
    }
}
```

4. Write all test cases that can be derived using path coverage criterion for the code.

Test Case Input Point

Test Case	Input Points	Expected Output
1	(0, 0), (3, 2), (1, -1), (2, 3)	(1, -1)
2	(4, 4), (3, 3), (2, 2), (1, 1)	(1, 1)
3	(5, 5), (5, 5), (4, 4), (3, 3)	(3, 3)
4	(-1, -1), (0, 1), (2, 0), (1, -2), (3, 3)	(1, -2)
5	(10, 10), (9, 5), (8, 8), (6, 6), (7, 0)	(7, 0)