Dhirubhai Ambani
Institute of Information and Communication Technology

IT–314 SOFTWARE ENGINEERING

Lab – 09: Mutation Testing

Student Name: Ishan Savaliya

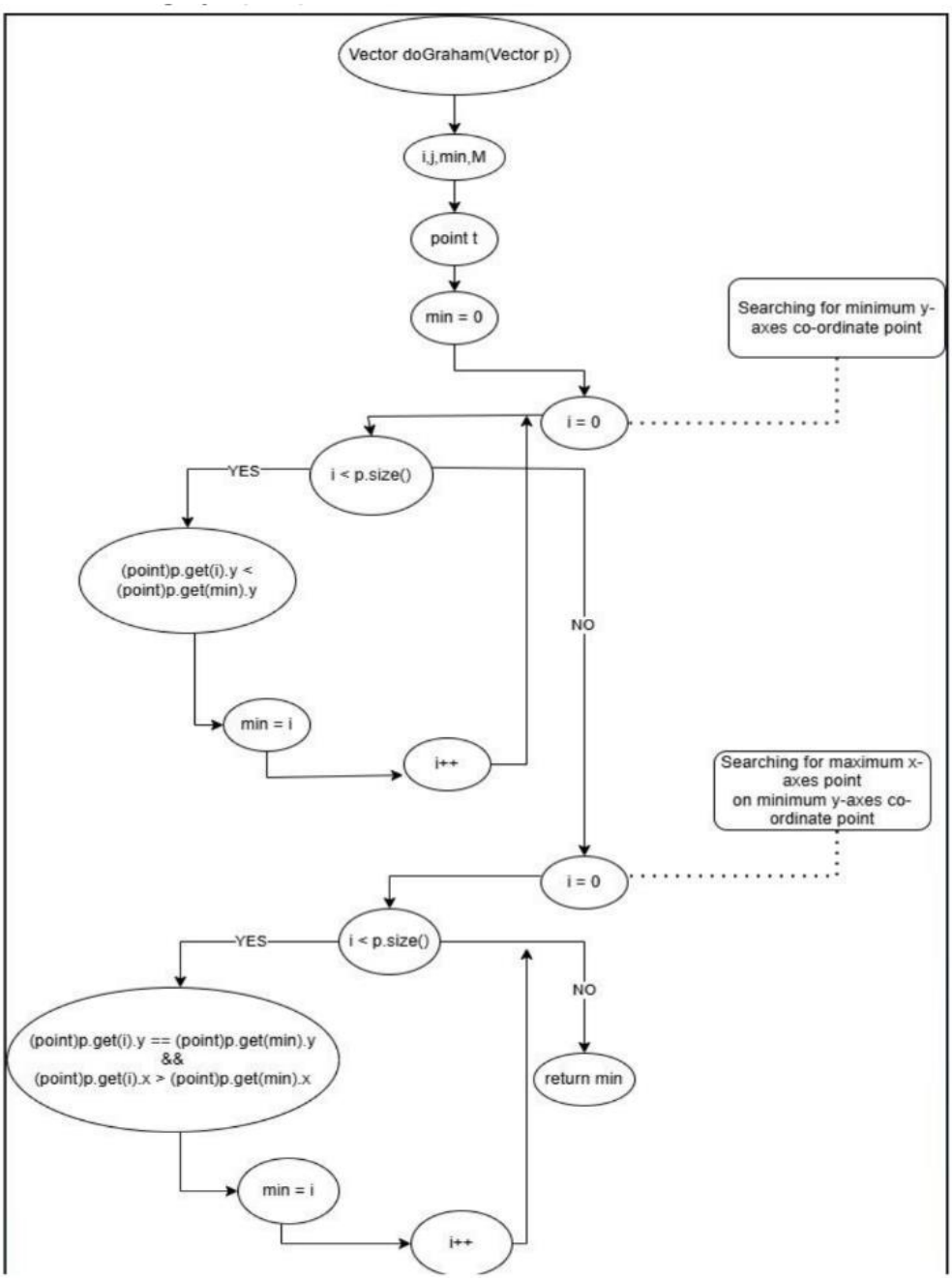
Student ID: 202201087

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.

```
#include <iostream>
#include <vector>
#include <algorithm>

class Point {
public:
    int x, y;

    Point(int x, int y) : x(x), y(y) {}

    friend std::ostream& operator<<(std::ostream& os, const Point& p) {
        os << "Point(x=" << p.x << ", y=" << p.y << ")";
        return os;
    }
};

// Define the do_graham function
Point do_graham(const std::vector<Point>& points) {
    int min_idx = 0;

    // Find the point with the minimum y-coordinate
    for (int i = 1; i < points.size(); i++) {
        if (points[i].y < points[min_idx].y) {
            min_idx = i;
        }
        // If there are points with the same y-coordinate, choose the one with
the minimum x-coordinate
        else if (points[i].y == points[min_idx].y && points[i].x <
points[min_idx].x) {
            min_idx = i;
        }
    }

    // Return the identified minimum point for clarity
    return points[min_idx];
}
```

```
int main() {
    std::vector<Point> points = {{1, 3}, {2, 2}, {3, 1}, {0, 1}, {1, 1}};

    Point minPoint = do_graham(points);
    std::cout << "Minimum point: " << minPoint << std::endl;

    return 0;
}
```

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.

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

class Point {
public:
    int x, y;

    Point(int x, int y) : x(x), y(y) {}

    friend std::ostream& operator<<(std::ostream& os, const Point& p) {
        os << "Point(x=" << p.x << ", y=" << p.y << ")";
        return os;
    }
};

// Function to find the minimum point based on y-coordinate (and x-coordinate if y is the same)
Point do_graham(const std::vector<Point>& p) {
    int min_idx = 0;

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

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

    // Return the identified minimum point
    return p[min_idx];
}
```

```
// Define the test cases
void run_tests() {
    std::vector<std::vector<Point>> test_cases = {
        // Test case 1 - Statement Coverage
        {Point(2, 3), Point(1, 2), Point(3, 1)},

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

        // Test cases for Basic Condition Coverage
        {Point(2, 3), Point(1, 2), Point(3, 1)}, // p[i].y < p[min_idx].y is
True
        {Point(1, 3), Point(2, 3), Point(3, 3)}, // p[i].y < p[min_idx].y is
False
        {Point(2, 2), Point(1, 2), Point(3, 2)}, // p[i].y == p[min_idx].y is
True, p[i].x < p[min_idx].x is True
        {Point(3, 2), Point(4, 2), 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 (size_t i = 0; i < test_cases.size(); ++i) {
        Point min_point = do_graham(test_cases[i]);
        std::cout << "Test Case " << (i + 1) << ": Input Points = ";
        for (const auto& point : test_cases[i]) {
            std::cout << point << " ";
        }
        std::cout << ", Minimum Point = " << min_point << std::endl;
    }
}

int main() {
    // Run the tests
    run_tests();
    return 0;
}
```

Output :

input

```
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=3, 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=1, 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=1, 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=2, 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 (p[i].y < p[min_idx].y) {
    min_idx = i;
}


// Mutated - deleted the condition check

min_idx = i
```

Statement Coverage Analysis:

- **Impact of Removing the Condition Check:** If the condition check is omitted, the code will consistently assign `i` to `min`, which could yield an incorrect outcome. This mistake may go undetected if the test cases only validate that `min` has been assigned, without checking that it holds the correct minimum value of `y`.
- **Risk of Undetected Errors:** If the tests focus solely on verifying that `min` is assigned, rather than confirming it is assigned the actual minimum value, this error may remain hidden.

b. Change Mutation:

```
// Original
if (p[i].y < p[min_idx].y)
// Mutated - changed < to <=

if (p[i].y <= p[min_idx].y)
```

Branch Coverage Analysis:

- **Effect of Changing < to <=:** Altering the < operator to <= could lead the code to assign min = i even when p[i].y is equal to p[min_idx].y, which may result in an incorrect selection of the minimum point.
- **Risk of Undetected Faults:** If the test set does not explicitly include cases where p[i].y is equal to p[min_idx].y, this change could introduce an undetected fault, as the subtle difference in behavior might not be captured by the existing tests.

b. Insertion Mutation:

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

Basic Condition Coverage Analysis:

- **Impact of Adding an Unnecessary Increment (i + 1):** Introducing an unnecessary increment (such as i + 1) modifies the intended assignment, which may cause min to reference an incorrect index. This could even lead to accessing elements outside the array bounds.
- **Risk of Undetected Errors:** If the test cases do not verify that min is assigned precisely to the expected index (without any additional increments), this change might go unnoticed. Tests that only confirm min is assigned, without ensuring its correctness, may fail to detect this issue.

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 terminate immediately without any processing. This covers the scenario where the vector size is zero, resulting in an early exit from the method.

Test Case 2: Loop Explored Once

- **Input:** A vector with a single point.
- **Test:** `Vector<Point> p = new Vector<Point>(); p.add(new Point(0, 0));`
- **Expected Result:** The loop should not be entered as `p.size()` is 1. The method should effectively leave the vector unchanged by swapping the only point with itself. This case validates the behavior when the loop condition is met only once.

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 method should enter the loop, compare the two points, and identify the second point as having a lower y-coordinate. The `minY` variable should update to `1`, and a swap should move the second point to the front of the vector.

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. It should determine that the second point has the lowest y-coordinate, updating `minY` to `1`. A swap should place the point `(1, 0)` at the front of the vector, covering the scenario where the loop iterates multiple times.

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.

Ans. Control Flow Graph Factory :- YES

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

Ans. 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