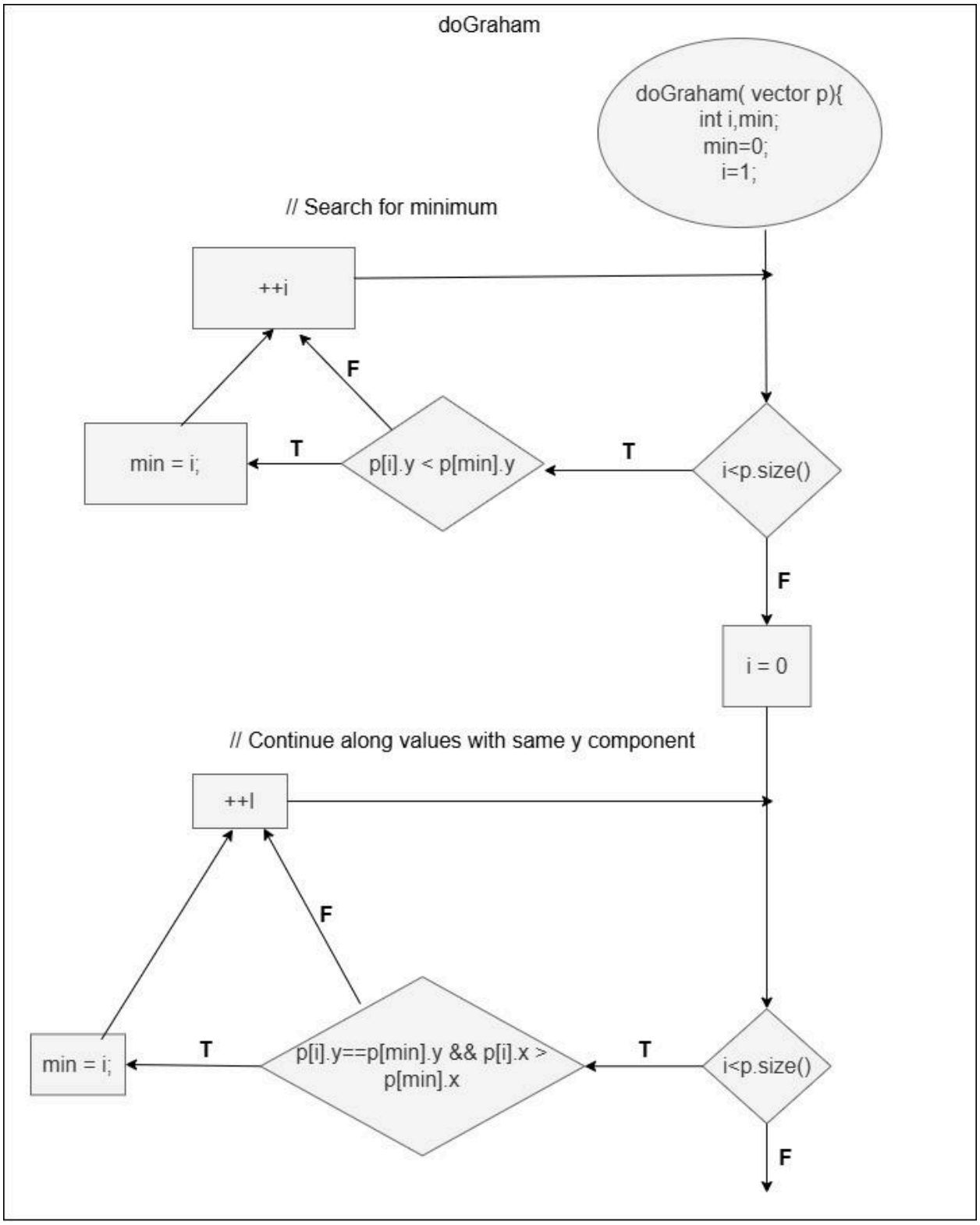
Software Engineering
LAB 9

Name: Jainil Patel
SID: 202201030

Q1) 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.

- **Part-1 : Control Flow Graph**



C++ Code:

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

class Point {
public:
    int x, y;

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

std::vector<Point> doGraham(std::vector<Point>& p) {
    int min = 0;

    // Search for the point with the minimum y-coordinate
    for (int i = 0; i < p.size(); i++) {
        if (p[i].y < p[min].y) {
            min = i;
        }
    }

    // Check for points with the same y-coordinate, and select the one
with the maximum x-coordinate
    for (int i = 0; i < p.size(); i++) {
        if (p[i].y == p[min].y && p[i].x > p[min].x) {
            min = i;
        }
    }

    // Output the bottom-rightmost point
    std::cout << "Bottom-rightmost point: (" << p[min].x << ", " <<
p[min].y << ")" << std::endl;

    return p;
}

int main() {
    // Sample points
```

```
    std::vector<Point> points = { Point(1, 2), Point(3, 4), Point(5, 2),
Point(2, 2), Point(0, 5) };

    // Find the bottom-rightmost point
    doGraham(points);

    return 0;
}
```

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.

- To achieve statement coverage, we need to ensure that every statement in the code is executed at least once. Here are some test cases to ensure that:

  Test Case 1:
  points = [(1, 2), (3, 4), (5, 2), (2, 2), (0, 5)]

  Output : (5,2)

- This test case will cover all statements in the function:

- The first `for` loop finds the point with the minimum y-coordinate (initially `min = 0`).
- The second `for` loop checks for points with the same y-coordinate and finds `(5, 2)` as the bottom-rightmost point.

Test Case 2:
points = [(2, 3), (3, 3), (1, 1), (5, 2), (6, 2)]

output: (6, 2)

- This test case also covers all statements:

- The first `for` loop finds the point with the minimum y-coordinate, which is initially `(1, 1)` since it has the lowest y-value.
- The second `for` loop checks for points with the same y-coordinate, finding `(6, 2)` as the bottom-rightmost point among points with y=2.

b. Branch Coverage.

- To achieve branch coverage, we need to ensure that all branches (true/false paths) in the code are executed. Here are the conditions we need to cover:

  if (p[i].y < p[min].y) (both true and false cases)
  if (p[i].y == p[min].y && p[i].x > p[min].x) (both true and false cases)

  Test Case 1:
  points = [(1, 2), (3, 4), (5, 2), (2, 2), (0, 5)]

  output: (5,2)

  - The condition `p[i].y < p[min].y` is evaluated multiple times. For some points, this condition is `true` (e.g., when `i = 3`, point `(2, 2)` has a smaller y-coordinate than the initial `min`).
  - The compound condition `p[i].y == p[min].y && p[i].x > p[min].x` is also evaluated as `true` for point `(5, 2)` since

it has the same y-coordinate as `(1, 2)` but a larger x-coordinate.

Test Case 2
points = [(7, 1), (3, 3), (4, 1), (2, 5), (8, 1)]

output: (8,1)

- The condition `p[i].y < p[min].y` is evaluated as `false` for points like `(3, 3)` and `(2, 5)` and `true` when initially setting `(7, 1)` as the minimum.
- The compound condition `p[i].y == p[min].y && p[i].x > p[min].x` evaluates as `true` for point `(8, 1)`, as it has the same y-coordinate as `(7, 1)` but a larger x-coordinate.

c. Basic Condition Coverage.

- For basic condition coverage, we need to test each individual condition within compound statements to evaluate both true and false outcomes independently.

  Test Case 1:
  points = [(1, 2), (3, 4), (5, 2), (2, 2), (0, 5)]

  output: (5,2)

- For the compound condition `p[i].y == p[min].y && p[i].x > p[min].x`:
- `p[i].y == p[min].y` is true for points with y=2y = 2y=2 and false for others.
- `p[i].x > p[min].x` is true for `(5, 2)` and false for `(1, 2)`.

Test Case 2:
points = [(4, 4), (2, 3), (6, 3), (1, 1), (3, 1)]

output : (3,1)

- For the condition `p[i].y == p[min].y && p[i].x > p[min].x`:
- `p[i].y == p[min].y` is true for points with y=1y = 1y=1 (i.e., `(1, 1)` and `(3, 1)`) and false for others like `(4, 4)` and `(2, 3)`.
- `p[i].x > p[min].x` is true for `(3, 1)` and false for `(1,1)`.

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) Deletion Mutation: Remove specific conditions or lines in the code.

Mutation Example: Remove the initial loop that searches for the point with the minimum y-coordinate.

```
std::vector<Point> doGraham(std::vector<Point> &p)

{

    int min = 0;

    // Removed the loop that finds the point with the minimum y-coordinate

    // Check for points with the same y-coordinate, and select the one with the
maximum x-coordinate

    for (int i = 0; i < p.size(); i++)
```

```
    {
        if (p[i].y == p[min].y && p[i].x > p[min].x)
        {
            min = i;
        }
    }
    std::cout << "Bottom-rightmost point: (" << p[min].x << ", " << p[min].y << ")" << std::endl;
    return p;
}
```

Expected Outcome: Without the first loop, the code will fail to find the true minimum y-value and only evaluate the points based on having the same y-value as the initial `min` point (index 0). This would result in the function potentially outputting an incorrect point as the "bottom-rightmost" if the first point in p is not actually the one with the lowest y-value.

Test Case Needed: A test case where there are multiple points with different y-values (e.g., `[(1, 3), (2, 2), (5, 4), (3, 2)]`) would reveal this issue. Without the first loop, the function might incorrectly select `(1, 3)` (assuming it's the first point) instead of `(2, 2)` or `(3, 2)`, which have the lowest y-values.

2) Change Mutation: Alter a condition, variable, or operator within the code.

Mutation Example: Change the < operator to <= in the condition `if (p[i].y < p[min].y)` in the first loop.

```
std::vector<Point> doGraham(std::vector<Point>& p) {

    int min = 0;

    // Search for the point with the minimum y-coordinate

    for (int i = 0; i < p.size(); i++) {

        if (p[i].y <= p[min].y) { // Changed < to <=

            min = i;

        }

    }

    // Check for points with the same y-coordinate, and select the one with the
maximum x-coordinate

    for (int i = 0; i < p.size(); i++) {

        if (p[i].y == p[min].y && p[i].x > p[min].x) {

            min = i;

        }

    }

    std::cout << "Bottom-rightmost point: (" << p[min].x << ", " << p[min].y <<
")" << std::endl;

    return p;

}
```

Expected Outcome: With the <= condition, points with equal y-values could replace `min`, potentially resulting in an incorrect selection of the minimum point. If the current tests don't thoroughly test multiple points with identical y-values, this could go undetected.

Test Case Needed: Include a test case with multiple points having the same y-value (e.g., `[(2, 1), (1, 1), (3, 1), (0, 1)]`) to ensure

the correct point is selected. This would expose incorrect behavior by checking if the rightmost point is chosen.

3) Insertion Mutation: Add extra statements or conditions to the code.

Mutation Example: Insert a line that resets the `min` index at the end of each loop iteration.

```
std::vector<Point> doGraham(std::vector<Point>& p) {

    int min = 0;

    // Search for the point with the minimum y-coordinate

    for (int i = 0; i < p.size(); i++) {

        if (p[i].y < p[min].y) {

            min = i;

        }

    }

    // Check for points with the same y-coordinate, and select the one with the
maximum x-coordinate

    for (int i = 0; i < p.size(); i++) {

        if (p[i].y == p[min].y && p[i].x > p[min].x) {

            min = i;

        }

        min = 0; // Added mutation: resetting min after each iteration

    }

    std::cout << "Bottom-rightmost point: (" << p[min].x << ", " << p[min].y <<
")" << std::endl;

    return p;

}
```

Expected Outcome: Resetting `min` after each iteration prevents the function from keeping track of the actual minimum point found so far, as `min` is always reset to 0. This can lead to incorrect results, as the code will fail to correctly identify the bottom-rightmost point.

Test Case Needed: Use a vector p with points in different positions for minimum values, such as `[(5, 5), (1, 0), (2, 3), (4, 2)]`. The correct `min` should persist across iterations, which will fail due to this mutation, making the function incorrectly output `(5, 5)` instead of `(1, 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 Case 1: Empty Input

- Input: `p = []`
- Expected Output: Nothing is printed (or a message indicating no points, depending on error handling).
- Path Covered:
    - Both loops are skipped because `p.size()` is zero.
    - This path tests the scenario where the function handles an empty vector without performing any iterations.

Test Case 2: Single Point

- Input: `p = [Point(1, 1)]`
- Expected Output: `Bottom-rightmost point: (1, 1)`
- Path Covered:

    - First loop (`for (int i = 0; i < p.size(); i++)`): Runs exactly once, identifying the only point as the minimum y-coordinate point.
    - Second loop (`for (int i = 0; i < p.size(); i++)`): Runs once, but no point has a greater x-coordinate with the same y-coordinate, so `min` remains the same.
    - This path tests the scenario where each loop runs exactly once with a single element.

Test Case 3: Multiple Points with Unique Y-Values

- Input: `p = [Point(1, 3), Point(2, 2), Point(5, 4), Point(3, 5)]`
- Expected Output: `Bottom-rightmost point: (2, 2)`
- Path Covered:
    - First loop: Iterates over all points, identifying the point `(2, 2)` as having the minimum y-coordinate.
    - Second loop: Iterates over all points but does not modify `min` since no other points have the same y-coordinate as `(2, 2)`.
    - This path covers the case where both loops run multiple times, with no modifications in the second loop.

Test Case 4: Multiple Points with the Same Minimum Y-Coordinate

- Input: `p = [Point(1, 2), Point(5, 2), Point(3, 2), Point(0, 3)]`
- Expected Output: `Bottom-rightmost point: (5, 2)`
- Path Covered:
    - First loop: Identifies the point `(1, 2)` as the initial minimum y-coordinate.

- Second loop: Finds multiple points with the same y-coordinate (`y = 2`). Among these, it updates `min` to the point `(5, 2)` due to its higher x-coordinate.
- This path tests the second loop's behavior when it encounters multiple points with the same y-coordinate and selects the one with the maximum x-coordinate.

Test Case 5: Two Points with the Same Y-Coordinate but Different X-Coordinates

- Input: `p = [Point(2, 2), Point(4, 2)]`
- Expected Output: `Bottom-rightmost point: (4, 2)`
- Path Covered:
  - First loop: Runs twice, identifying `(2, 2)` as the initial minimum y-coordinate point.
  - Second loop: Runs twice, updating `min` to `(4, 2)` because it has a larger x-coordinate with the same y-coordinate.
  - This path covers the scenario where both loops execute exactly twice, with the second loop performing an update based on x-coordinate.

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.

-> Yes

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

-> Statement Coverage: 2
Branch Coverage: 2
Basic Condition Coverage: 2
Path Coverage: 5

- Summary of Minimum Test Cases:

  Total: 2 (Statement) + 2 (Branch) + 2 (Basic Condition) + 5 (Path) = 11 test cases