IT314: Software Engineering

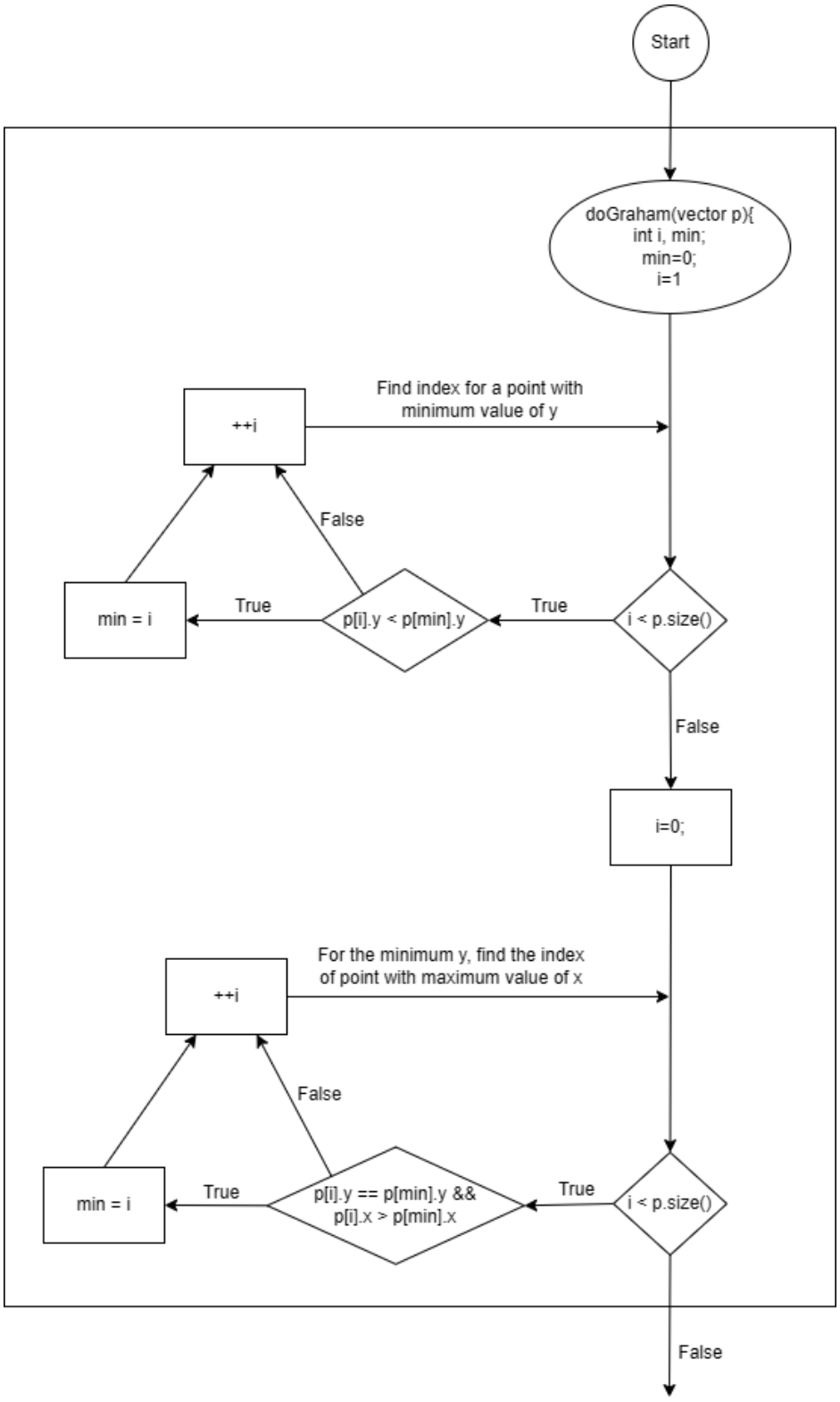
Lab 9: Mutation Testing

Name: Venil Vekariya

Student ID: 202201078

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.

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.



Code:

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

class Point {
public:
    int x, y;

    Point(int x, int y)
    {
        this->x = x;
        this->y = y;
    }
};

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

    return p;
}

int main() {
    // Sample points
    vector<Point> points;
    points.push_back(Point(1, 2));
    points.push_back(Point(3, 4));
    points.push_back(Point(5, 2));
    points.push_back(Point(2, 2));
    points.push_back(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: 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:** Input points: [(1, 2), (3, 4), (5, 2), (2, 2), (0, 5)]
    - This test case will cover statements for initializing min, iterating through p to find the minimum y, and then finding the point with the maximum x for the minimum y.
- **Test Case 2:** Input points: [(5, 5), (4, 4), (3, 3), (2, 2), (1, 1)]
    - This test case ensures all statements are executed, especially when the minimum y point is at the end of the vector.

Expected Outcome:

- Test Case 1 should output the point (5, 2) as the bottom-rightmost point.
- Test Case 2 should output the point (1, 1) as the bottom-rightmost point.

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 Cases for Branch Coverage:

- **Test Case 1:** Input points: [(1, 2), (3, 4), (5, 2), (2, 2), (0, 5)]
    - Covers cases where points with the same y value but different x values exist.
- **Test Case 2:** Input points: [(0, 1), (0, 0), (0, -1)]
    - This test ensures the code handles cases where all x values are the same, testing the condition p[i].x > p[min].x.

Expected Outcome:

- Test Case 1 should output the point (5, 2).

- Test Case 2 should output the point (0, -1) as the bottom-rightmost point.

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 Cases for Basic Condition Coverage:

- **Test Case 1:** Input points: [(1, 2), (3, 4), (5, 2), (2, 2), (0, 5)]
- **Test Case 2:** Input points: [(5, 2), (3, 2), (1, 2)]
    - Ensures that the condition p[i].y == p[min].y holds, but p[i].x > p[min].x will be evaluated multiple times.

Expected Outcome:

- Test Case 1 should output the point (5, 2).
- Test Case 2 should output the point (5, 2) as well, verifying the maximum x for the same minimum y.

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 second for loop which checks for points with the same y-coordinate and selects the one with the maximum x-coordinate.

Code:

```
vector<Point> doGraham(vector<Point>& p) {
    int min = 0;
    for (int i = 0; i < p.size(); ++i) {
        if (p[i].y < p[min].y) {
            min = i;
        }
    }
    // Removed the loop that finds the point with maximum x for the minimum y
    cout << "Bottom-rightmost point: (" << p[min].x << ", " << p[min].y << ")"
<< endl;
    return p;
}
```

Expected Outcome: Without this loop, the code will only find the point with the minimum y value and ignore cases where multiple points have the same y value. This could result in the incorrect point being chosen as the "bottom-rightmost" point.

Test Case Needed: A test case where there are multiple points with the same minimum y value but different x values (e.g., [(1, 2), (3, 4), (5, 2), (2, 2)]) would reveal this issue, as it would not pick the point (5, 2) as expected.

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

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

Code:

```
vector<Point> doGraham(vector<Point>& p) {
    int min = 0;
    for (int i = 0; i < p.size(); ++i) {
        if (p[i].y <= p[min].y) { // Changed < to <=
            min = i;
        }
    }
    for (int i = 0; i < p.size(); ++i) {
        if (p[i].y == p[min].y && p[i].x > p[min].x) {
            min = i;
        }
    }
    cout << "Bottom-rightmost point: (" << p[min].x << ", " << p[min].y << ")"
<< endl;
    return p;
}
```

Expected Outcome: With the <= condition, points with equal y values could replace min, resulting in the last occurrence of the minimum y point being selected rather than the true bottom-rightmost point.

Test Case Needed: Include a test case with multiple points that have the same y value, such as [(2, 1), (1, 1), (3, 1), (0, 1)]. This would expose the issue by showing that the code incorrectly selects the last point (0, 1) instead of the correct bottom-rightmost point (3, 1).

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 in the first for loop.

Code:

```
vector<Point> doGraham(vector<Point>& p) {
    int min = 0;
    for (int i = 0; i < p.size(); ++i) {
        if (p[i].y < p[min].y) {
            min = i;
        }
        min = 0; // Added mutation: resetting min after each iteration
    }
    for (int i = 0; i < p.size(); ++i) {
        if (p[i].y == p[min].y && p[i].x > p[min].x) {
            min = i;
        }
    }
    cout << "Bottom-rightmost point: (" << p[min].x << ", " << p[min].y << ")"
<< endl;
    return p;
}
```

Expected Outcome: Resetting min after each iteration prevents the code from keeping track of the actual minimum point found so far, as min is always reset to 0. This will lead to incorrect results as the code will always output the first point in the list instead of the true minimum.

Test Case Needed: A vector p with points in different positions for minimum values, such as [(5, 5), (1, 0), (2, 3), (4, 2)], will help detect this issue. The expected minimum should persist across iterations, but due to this mutation, it will fail and always select the first point (5, 5).

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 0 times):** Input an empty list of points [].

```
[] // Empty list of points
```

  - **Expected Output:** Should handle this ideally by returning an error message or a special indication (such as "No points provided") without causing runtime errors.
  - **Path Covered:** This path skips the for loops entirely, covering the scenario where there are no elements to process.

- **Test Case 2 (Loop 1 time):** Input a single point [(0, 0)].

```
[(0, 0)] // Single point
```

  - **Expected Outcome:** The output should be: Bottom-rightmost point: (0, 0)
  - **Path Covered:** The first loop will iterate only once, identifying (0, 0) as the minimum y and maximum x point by default. The second loop will also iterate only once, confirming that (0, 0) is the bottom-rightmost point.

- **Test Case 3 (Loop 2 times):** Input two points with the same y but different x values, such as [(1, 0), (2, 0)].

```
[(1, 0), (2, 0)] // Two points with the same y-coordinate but different x-coordinates
```

  - **Expected Outcome:** The output should be: Bottom-rightmost point: (2, 0)
  - **Path Covered:**
    - The first loop will iterate twice. The code will first select (1, 0) as the minimum y point and then replace it with (2, 0) due to the larger x value.
    - The second loop confirms (2, 0) as the bottom-rightmost point since it has the same y but a larger x than (1, 0).

- **Test Case 4 (Loop Multiple times):** Use a standard test case like [(1, 2), (3, 4), (5, 2), (2, 2), (0, 5)].

```
[(1, 2), (3, 4), (5, 2), (2, 2), (0, 5)] // Multiple points with distinct y and x values
```

    - **Expected Outcome:** The output should be: Bottom-rightmost point: (5, 2)
    - **Path Covered:**
        - The first loop iterates over all points to find the minimum y point. It identifies (1, 2) initially but later replaces it with (5, 2) due to the same y and larger x.
        - The second loop iterates over all points again to confirm that (5, 2) is indeed the bottom-rightmost point.

- **Test Case 5 (Loop Multiple times):** Loop Multiple times with multiple points having the same minimum y-value

```
[(2, 1), (1, 1), (3, 1), (0, 1)] // Multiple points with identical y-values but different x-values
```

    - **Expected Outcome:** The output should be: Bottom-rightmost point: (3, 1)
    - **Path Covered:**
        - The first loop iterates over all points and identifies (2, 1) initially but eventually selects (3, 1) as it has the same y but a larger x than others.
        - The second loop confirms (3, 1) as the bottom-rightmost point.

Lab Execution:

1. 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).

Ans) Control Flow Graph Factory Tool: YES

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

Ans)

- Statement Coverage: 2 test cases
- Branch Coverage: 2 test cases
- Basic Condition Coverage: 2 test cases
- Path Coverage: 5 test cases

Summary of Minimum Test Cases:

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