Software Engineering (IT314)

LAB-9

Name: Kaival Hemanshu Mehta

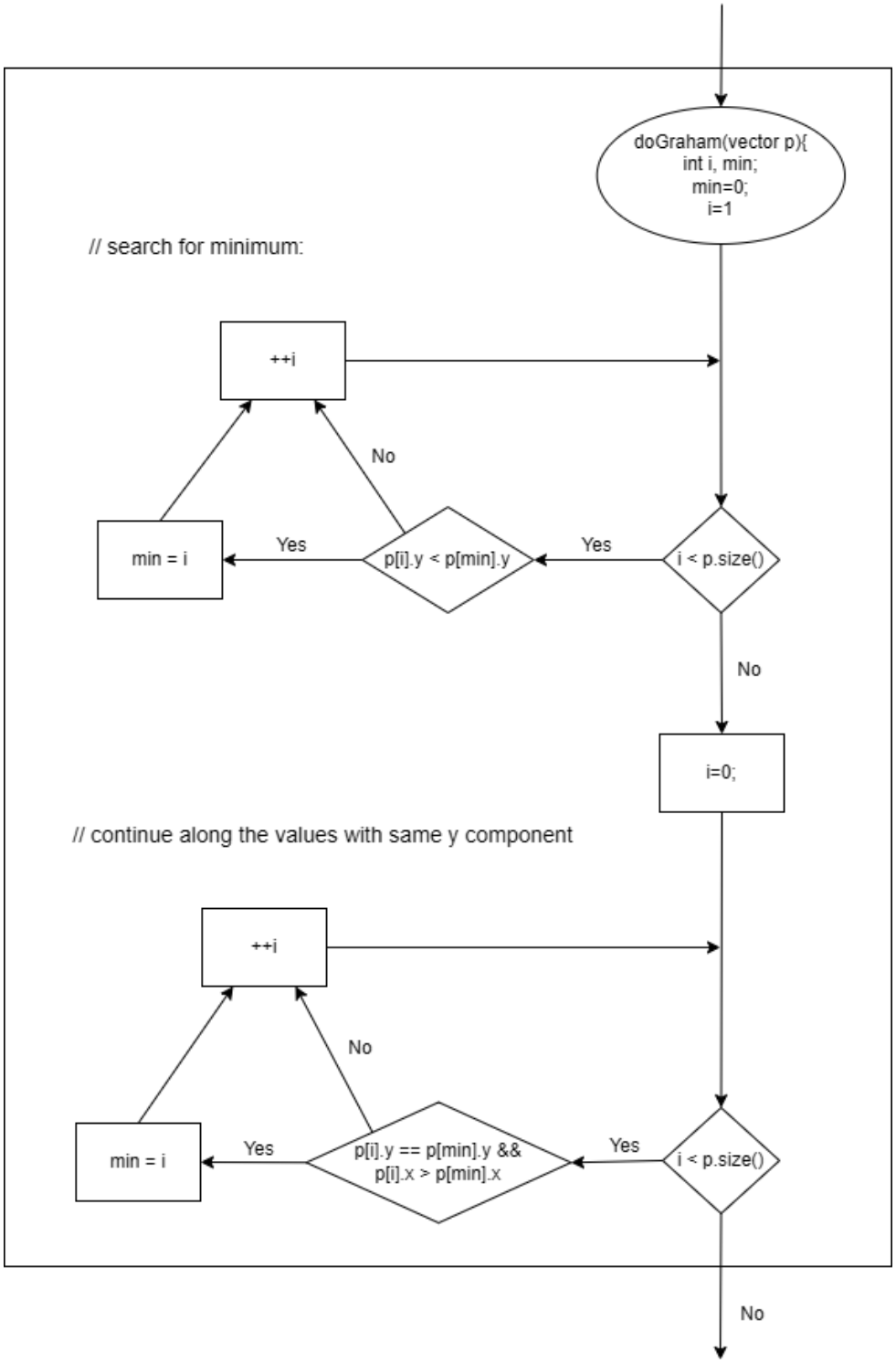
ID: 202201009

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.**

```cpp
#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;
        }
    }
```

```
    // Select the one with the maximum x-coordinate for minimum y-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;
}

int main() {
    // Sample points
    std::vector<Point> points = { Point(2, 4), Point(4, 6), Point(3, 2),
Point(1, 2), Point(6, 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, each line in the code should be executed at least once.

Test Case 1:

- Input: points = [Point(2, 4), Point(3, 2)]
- Expected Outcome: (3, 2) is identified as the bottom-rightmost point.

Test Case 2:

- Input: points = [Point(5, 3), Point(2, 3), Point(1, 5)]
- Expected Outcome: (5, 3) is identified as the bottom-rightmost point.

These test cases ensure that both the search for the minimum y coordinate and the search for the maximum x coordinate with the minimum y are exercised, covering all statements at least once.

b. Branch Coverage

For branch coverage, we need to evaluate each branch (or decision point) as both true and false.

Test Case 1:

- Input: points = [Point(3, 5), Point(4, 2), Point(1, 2), Point(6, 6)]
- Expected Outcome: (4, 2) is identified as the bottom-rightmost point.
- This covers:
    - p[i].y < p[min].y as true when comparing Point(4, 2) with Point(3, 5).
    - p[i].y == p[min].y and p[i].x > p[min].x as false.

Test Case 2:

- Input: points = [Point(2, 4), Point(3, 2), Point(1, 2), Point(5, 5)]
- Expected Outcome: (3, 2) is identified as the bottom-rightmost point.

- This covers:
    - p[i].y < p[min].y as false when comparing Point(1, 2) with Point(3, 2).
    - p[i].y == p[min].y and p[i].x > p[min].x as true.

These cases ensure that every decision (branch) point is evaluated as both true and false.

c. Basic Condition Coverage

For basic condition coverage, each simple boolean expression in a decision should be evaluated as true and false independently.

Test Case 1:

- Input: points = [Point(3, 3), Point(4, 2), Point(2, 2), Point(6, 4)]
- Expected Outcome: (4, 2) is identified as the bottom-rightmost point.
- Covers:
    - p[i].y < p[min].y as true (Point (4, 2) is less than (3, 3)) and false (other points).
    - p[i].x > p[min].x as true (for (4, 2) over (2, 2)) and false (for (2, 2) over (4, 2)).

Test Case 2:

- Input: points = [Point(1, 4), Point(3, 2), Point(5, 2), Point(0, 6)]
- Expected Outcome: (5, 2) is identified as the bottom-rightmost point.
- Covers:
    - p[i].y < p[min].y as false initially.
    - p[i].y == p[min].y as true when comparing (3, 2) and (5, 2), where p[i].x > p[min].x also holds true.

These test cases ensure that each individual condition (p[i].y < p[min].y, p[i].y == p[min].y, p[i].x > p[min].x) is evaluated as both true and 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. Deletion Mutation

Code (Deleted the condition p[i].y == p[min].y in the second loop)

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

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

    // Select the one with the maximum x-coordinate for minimum y-coordinate
    for (int i = 0; i < p.size(); ++i) {
        // Mutation: Removed the condition `p[i].y == p[min].y`
        if (p[i].x > p[min].x) {
            min = i;
        }
    }

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

Expected Outcome

Without the condition p[i].y == p[min].y, the function will select the point with the maximum x coordinate regardless of the y value. This will produce an incorrect result when the bottom-rightmost point should be identified based on both the minimum y value and the maximum x value among those with the same y value.

Test Case:

- Input: points = [Point(2, 4), Point(3, 2), Point(5, 6)]
- Expected Outcome: Original code would output (3, 2), but the mutated code may select (5, 6) as the point with the highest x coordinate.

2. Change Mutation

Code (Changed p[i].y < p[min].y to p[i].y > p[min].y in the first loop)

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

    // Search for the point with the maximum y-coordinate (incorrectly)
    for (int i = 1; i < p.size(); ++i) {
        if (p[i].y > p[min].y) {  // Mutation: changed < to >
            min = i;
```

```
        }
    }

    // Select the one with the maximum x-coordinate for minimum y-coordinate
    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 this mutation, the function mistakenly selects the point with the maximum y coordinate, rather than the minimum. As a result, it will not correctly identify the bottom-rightmost point based on the smallest y value.

Test Case:

- Input: points = [Point(3, 1), Point(2, 5), Point(1, 3)]
- Expected Outcome: Original code would output (3, 1), but the mutated code may select (2, 5) due to the incorrect comparison.

3. Insertion Mutation

Code (Inserted an extra condition && p[i].x < p[min].x in the second loop)

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

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

    // Select the one with the maximum x-coordinate for minimum y-coordinate
    for (int i = 0; i < p.size(); ++i) {
        // Mutation: Added `&& p[i].x < p[min].x` condition
        if (p[i].y == p[min].y && p[i].x > p[min].x && p[i].x < p[min].x) {
            min = i;
        }
    }
```

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

Expected Outcome

With this mutation, the added condition p[i].x < p[min].x creates a contradiction, which will prevent the min value from updating as intended when points have the same y coordinate but different x values. This will lead to the algorithm failing to find the correct bottom-rightmost point in cases where the x coordinate needs to be updated.

Test Cases

- Input: points = [Point(1, 2), Point(3, 2), Point(2, 2)]
- Expected Outcome: Original code would output (3, 2), but the mutated code will fail to update min correctly due to the contradictory condition, and may retain (1, 2) or another incorrect point as the result.

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

For Path Coverage, we need to cover all possible paths, including paths where each loop executes zero, one, or more times.

- **Zero Iterations:**
- Although the loop will always iterate due to the input points, for completeness, you could simulate a scenario where p.size() is zero to check for edge cases.
    - Input: points = []
    - Expected Outcome: The code should handle this gracefully without error (though it is undefined in this current code setup).

- **One Iteration:**
- Test the case where there is only one point.
    - Input: points = [Point(2, 4)]
    - Expected Outcome: (2, 4) is identified as the bottom-rightmost point.

- **Two Iterations:**
- Minimal test case with two points having different y coordinates.
    - Input: points = [Point(2, 4), Point(3, 2)]
    - Expected Outcome: (3, 2) is identified as the bottom-rightmost point.

- **Multiple Iterations:**
- Test case where there are multiple points, with several having the same y coordinate but varying x coordinates.
    - Input: points = [Point(2, 4), Point(4, 6), Point(3, 2), Point(1, 2), Point(6, 5)]
    - Expected Outcome: (3, 2) is identified 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: YES

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

Ans) 1. Statement Coverage: 2 test cases

2. Branch Coverage: 2 test cases

3. Basic Condition Coverage: 2 test cases

4. Path Coverage: 4 test cases

Summary of Minimum Test Cases:

Total: 2 (Statement) + 2 (Branch) + 2 (Basic Condition) + 4 (Path) = 10 test cases