NAME – Denil Antala

ID - 202201090

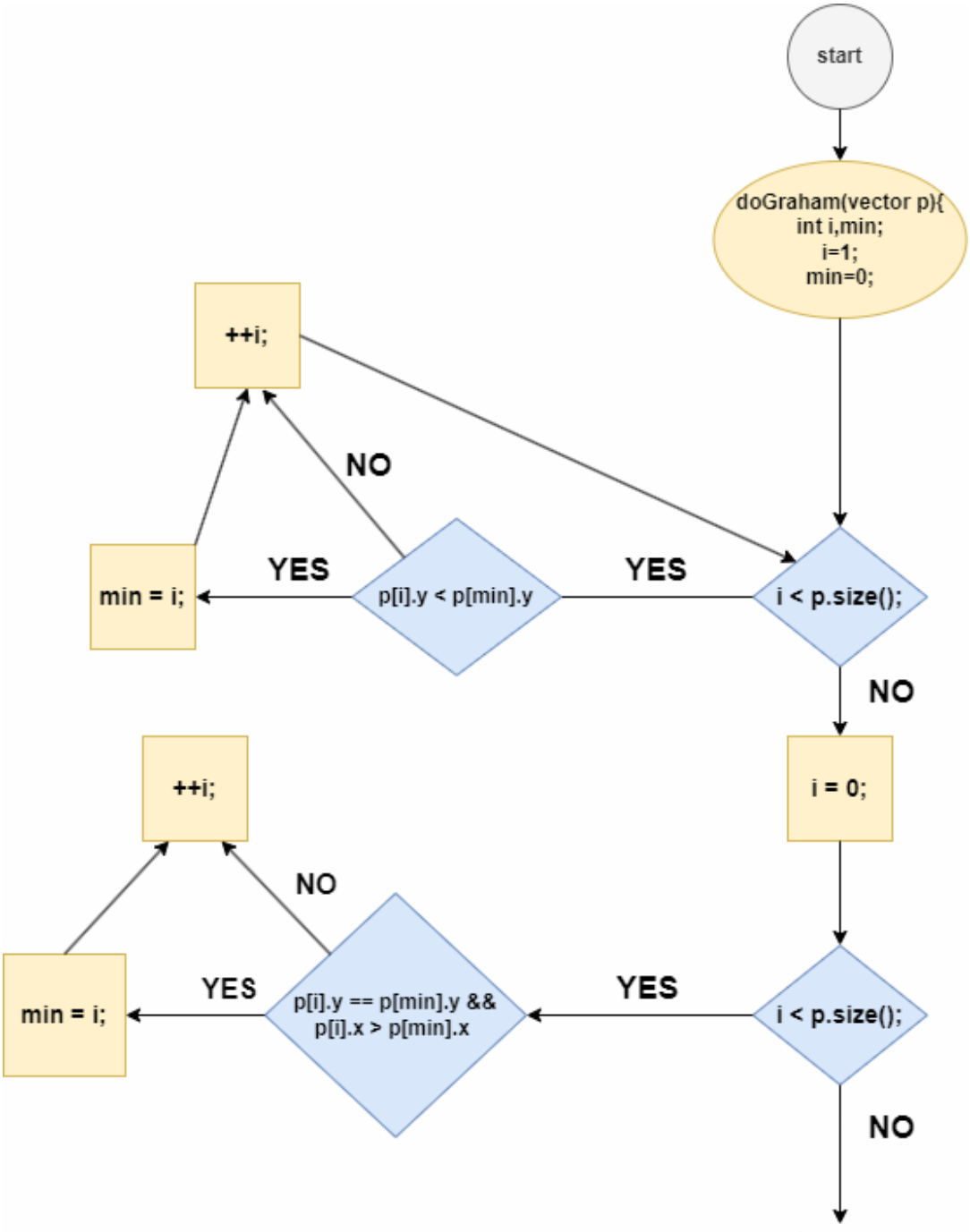
LAB - SWE(09)

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. Thisexercise is concerned with structural testing of code, so the focus is on creating test sets that satisfy some particular coverage criteria.

Task - 1:

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 DIAGRAM



C++ CODE :

```
#include <bits/stdc++.h>
```

```cpp
using namespace std;
#define ll long long
#define ld long double
#define pb push_back

class pt
{
public:

    double x, y;

    pt(double x, double y)
    {
        this->x = x;
        this->y = y;
    }
};

class ConvexHull
{
public:
    void DoGraham(vector<pt> &p)
    {
        ll i = 1;
        ll min = 0;

        if (p.size() <= 3)
        {
            return;
        }
        while (i < p.size())
        {
            if (p[i].y < p[min].y)
            {
                min = i;
            }
            else if (p[i].y == p[min].y)
            {
                if (p[i].x < p[min].x)
                {
                    min = i;
                }
            }
            i++;
        }
    }
};



int32_t main()
{
    vector<pt> polls;
```

```
    polls.pb(pt(0, 0));
    polls.pb(pt(1, 1));
    polls.pb(pt(2, 2));

    ConvexHull hull;

    hull.DoGraham(polls);
}
```

Task - 2:

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

a) Statement Coverage:

Objective: Ensure every line in the DoGraham method is executed at least once.

Test Case 1: p.size() <= 3.

- Input: A vector p with fewer than or equal to 3 points (e.g., (0, 0), (1,1), (2, 2)).
- Expected Outcome: The method returns immediately, covering the initial return statement.

Test Case 2: p.size() > 3, with points having unique y and x values.

- Input: A vector p with points such as (0, 0), (1, 2), (2, 3), (3, 4).
- Expected Outcome: The loop iterates through each point to find the point with the smallest y, covering the loop body and if-else conditions.

Test Case 3: p.size() > 3, with points having the same y values but different x values.

- Input: Points like (1, 1), (2, 1), (0, 1), (3, 2).

- Expected Outcome: The loop finds the point with the smallest x among those with the same y, covering both y and x comparison statements.

Test Case 4: p.size() > 3, with all points having identical y and x values.

- Input: Points like (1, 1), (1, 1), (1, 1), (1, 1).
- Expected Outcome: The loop iterates without any change to min because all points are identical, ensuring the loop completes without updating min.

b) Branch Coverage:

Objective: Ensure each branch outcome (true/false) for all decision points is tested.

Test Case 1: p.size() <= 3.

- Input: A vector p with 3 or fewer points, such as (0, 0), (1, 1), (2, 2).
- Expected Outcome: Method returns immediately, covering the false branch for the loop entry.

Test Case 2: p.size() > 3, with all y values distinct.

- Input: Points such as (0, 0), (1, 1), (2, 2), (3, 3).
- Expected Outcome: True branch of the main if condition (p[i].y < p[min].y) is covered, as each y is unique.

Test Case 3: p.size() > 3, with some points having the same y values but different x values.

- Input: Points like (1, 1), (2, 1), (0, 1), (3, 2).
- Expected Outcome: Covers true for if (when y is lower) and true/false for else if (same y, different x).

Test Case 4: p.size() > 3, with multiple points having the same y and x values.

- Input: Points like (1, 1), (1, 1), (1, 1), (1, 1).
- Expected Outcome: The loop iterates, covering the false branch for all comparisons since no min update occurs.

c) Basic Condition Coverage:

Objective: Test each atomic condition within the method independently to achieve all possible outcomes.

Conditions:

1. p.size() <= 3
2. p[i].y < p[min].y
3. p[i].y == p[min].y
4. p[i].x < p[min].x

Test Case 1: p.size() <= 3.

- Input: A vector p with 3 or fewer points, e.g., (0, 0), (1, 1), (2, 2).
- Expected Outcome: Tests the true condition of p.size() <= 3.

Test Case 2: p.size() > 3, with points having distinct y values.

- Input: Points like (0, 0), (1, 1), (2, 2), (3, 3).
- Expected Outcome: True outcome for p[i].y < p[min].y as each point has a higher y than the initial min.

Test Case 3: p.size() > 3, with points having the same y but different x values.

- Input: Points like (0, 1), (2, 1), (3, 1), (1, 0).
- Expected Outcome: True outcome for p[i].y == p[min].y and both true/false outcomes for p[i].x < p[min].x.

Test Case 4: p.size() > 3, with identical y and x values.

- Input: Points like (1, 1), (1, 1), (1, 1), (1, 1).
- Expected Outcome: False outcomes for p[i].y < p[min].y, p[i].y == p[min].y, and p[i].x < p[min].x as no change to min occurs.

Task - 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 condition if (p.size() <= 3) at the start of the method.

CODE:

```
void DoGraham(vector<pt> &p)
{
    ll i = 1;
```

```
    ll min = 0;

    // Removed condition check for p.size() <= 3
    while (i < p.size())
    {
        if (p[i].y < p[min].y)
        {
            min = i;
        }
        else if (p[i].y == p[min].y)
        {
            if (p[i].x < p[min].x)
            {
                min = i;
            }
        }
        i++;
    }
}
```

Expected Outcome: Without this condition, the method may run even if p.size() <= 3, potentially leading to unnecessary or incorrect calculations when the input size is insufficient. Our existing tests, which only verify cases where p.size() > 3, could miss this, allowing the mutation to go unnoticed.

Test Case Needed: A test case where p.size() is exactly 3 (for example, points (0,0), (1,1), and (2,2)) would help detect this issue by revealing any unnecessary calculations that would now run.

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).

CODE:

```
void DoGraham(vector<pt> &p)
{
    ll i = 1;
    ll min = 0;
    if (p.size() <= 3)
    {
        return;
    }
    while (i < p.size())
    {
        if (p[i].y <= p[min].y)
        { // Changed < to <=
            min = i;
        }
        else if (p[i].y == p[min].y)
        {
            if (p[i].x < p[min].x)
            {
                min = i;
            }
        }
        i++;
    }
}
```

Expected Outcome: Adding the <= condition could cause points with the same y values to incorrectly replace min, leading to the wrong choice for the minimal point. Current test cases may not fully check scenarios with multiple points sharing the same y value, so this issue might go unnoticed.

Test Case Needed: Add a test case with several points that have the same y value (e.g., (2, 1), (1, 1), (3, 1), (0, 1)) to ensure that the point with the smallest x is correctly chosen. This would help reveal any

incorrect behavior, such as selecting the last instance rather than the true minimum.

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.

CODE:

```
void DoGraham(vector<pt> &p)
{
    ll i = 1;
    ll min = 0;
    if (p.size() <= 3)
    {
        return;
    }
    while (i < p.size())
    {
        if (p[i].y < p[min].y)
        {
            min = i;
        }
        else if (p[i].y == p[min].y)
        {
            if (p[i].x < p[min].x)
            {
                min = i;
            }
        }
        i++;
        min = 0; // Added mutation: resetting min after each iteration
    }
}
```

Expected Outcome: Resetting min after each iteration prevents the code from keeping track of the true minimum point found so far, as

min is always reset to 0. This could lead to incorrect results, especially if the smallest y or x value doesn't appear at the beginning of the sequence.

Test Case Needed: Use a vector p with points where the minimum values are in different positions, like [(5, 5), (1, 0), (2, 3), (4, 2)], to catch this issue. The correct minimum should stay consistent across iterations, which will fail with this bug.

Task - 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: p.size() = 0

- Input: p = [] (empty vector)
- Expected Outcome: Since p.size() == 0, the method will exit immediately without entering the loop.
- Path Covered: Covers the case where the loop condition is false from the start, so the loop doesn't run.

Test Case 2: p.size() = 1 (No iterations)

- Input: p = [(0, 0)] (single point)
- Expected Outcome: Since p.size() <= 3, the method will return immediately, skipping the loop.
- Path Covered: Ensures early exit when p.size() <= 3.

Test Case 3: p.size() = 4 with one loop iteration

- Input: p = [(0, 0), (1, 1), (2, 2), (3, 3)]
- Expected Outcome: The method checks p.size() > 3, enters the loop, evaluates (1, 1) in the first iteration, updates min to the index with the lowest y or x, and exits.
- Path Covered: Covers the scenario where the loop runs only once before completing.

Test Case 4: p.size() = 4 with two loop iterations

- Input: p = [(0, 0), (1, 2), (2, 1), (3, 3)]
- Expected Outcome: The method evaluates the first two points in the loop to find the minimum. After two iterations, it updates min and then either continues or exits.
- Path Covered: Covers the scenario where the loop runs exactly twice.

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 flowgraph generator.

=> YES

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

Ans. Statement Coverage: 3

1. Branch Coverage: 3
2. Basic Condition Coverage: 3
3. Path Coverage: 3

Summary of Minimum Test Cases:

● `Total: 3 (Statement) + 3 (Branch) + 2 (Basic Condition) + 3 (Path) = 11 test cases`