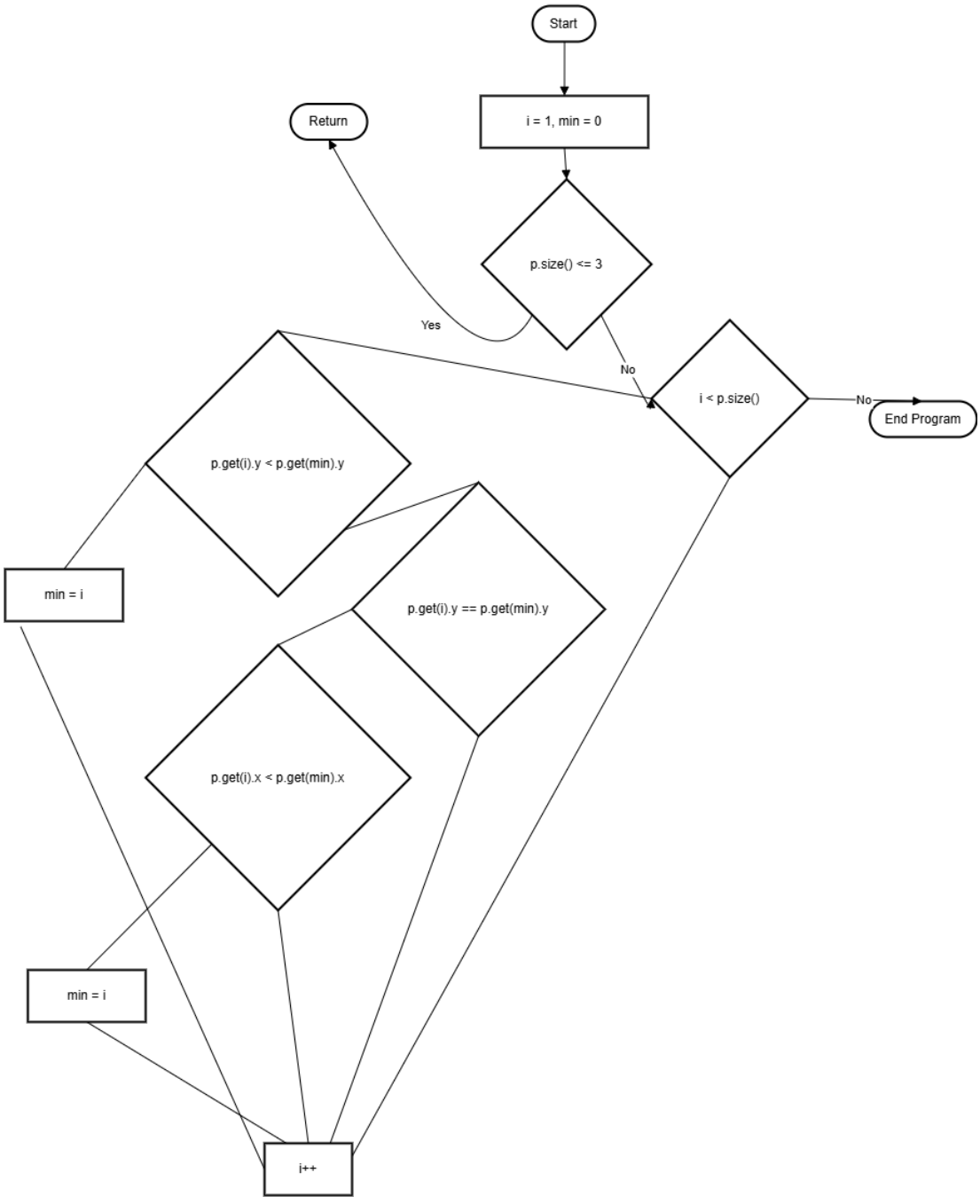
Lab- 9
Mutation Testing

ID: 202201194
Name: Kenil Sarang

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 isa 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.

Task - 1: Control flow graph:



C++ Code**:**

```
#include <bits/stdc++.h>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 theinitial 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 findthe 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 butdifferent 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 withoutupdating 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 thefalse 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 butdifferent 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/falsefor else if (same y, different x).

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

- Input: Points like (1, 1), (1, 1), (1, 1), (1, 1).
- Expected Outcome: The loop iterates, covering the false branchfor 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 eachpoint has a higher y than the initial min.

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

- Input: Points like (0, 1), (2, 1), (3, 1), (1, 0).
- Expected Outcome: True outcome for p[i].y == p[min].yand 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 themethod.

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 will attempt to runeven if p.size() <= 3, which could cause unnecessary or incorrect calculations in cases where the input size is insufficient. Our current tests

that only check when p.size() > 3 might miss this, allowing the mutation togo undetected.

Test Case Needed: A test case where p.size() is exactly 3 (such as points(0,0), (1,1), and (2,2)) would help detect this issue, as it would now run unnecessary calculations.

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: With the <= condition, points with equal y valuescould inappropriately replace min, which might result in the incorrect selection of the minimal point. Since the current test cases may not thoroughly test multiple points with identical y values, this could go undetected.

Test Case Needed: Include a test case with multiple points with identical yvalues (e.g., (2, 1), (1, 1), (3, 1), (0, 1)) to ensure the smallest x is selectedcorrectly. This would expose the incorrect behavior of selecting the last instance instead of the actual 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 eachloop iteration.

```
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
    }
}
```

Code:

Expected Outcome: Resetting min after each iteration prevents the codefrom keeping track of the actual minimum point found so far, as min is always reset to 0. This can lead to an incorrect result if not tested with sequences of points where the smallest y or x is not at the beginning.

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 correct min should persist across iterations, which will fail due tothis mutation.

Task - 4: Create a test set that satisfies the path coveragecriterion where every loop is explored at least zero,
one or two times.

Test Case 1: p.size() = 0 (Zero iterations).

- Input: p = [] (empty vector).
- Expected Outcome: Since p.size() == 0, the method willexit immediately without entering the loop.
- Path Covered: This covers the path where the loop condition fails initially, so the loop is not executed.

Test Case 2: p.size() = 1 (Zero iterations).

- Input: p = [(0, 0)] (a single point).
- Expected Outcome: Since p.size() <= 3, the method will return immediately, covering the case where the loop is skipped.
- Path Covered: Ensures that the method exits early due to the condition p.size() <= 3.

Test Case 3: p.size() = 4 with the loop executing once.

- Input: p = [(0, 0), (1, 1), (2, 2), (3, 3)].
- Expected Outcome: The method checks p.size() > 3, then enters theloop. On the first iteration, it evaluates (1, 1), updates min to the index with the lowest y or x, and exits.

- Path Covered: Covers the path where the loop executes once before completion.

Test Case 4: p.size() = 4 with the loop executing two times.

- 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 prepares to continue or exit.
- Path Covered: This covers the path where the loop executes exactlytwice.

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.

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