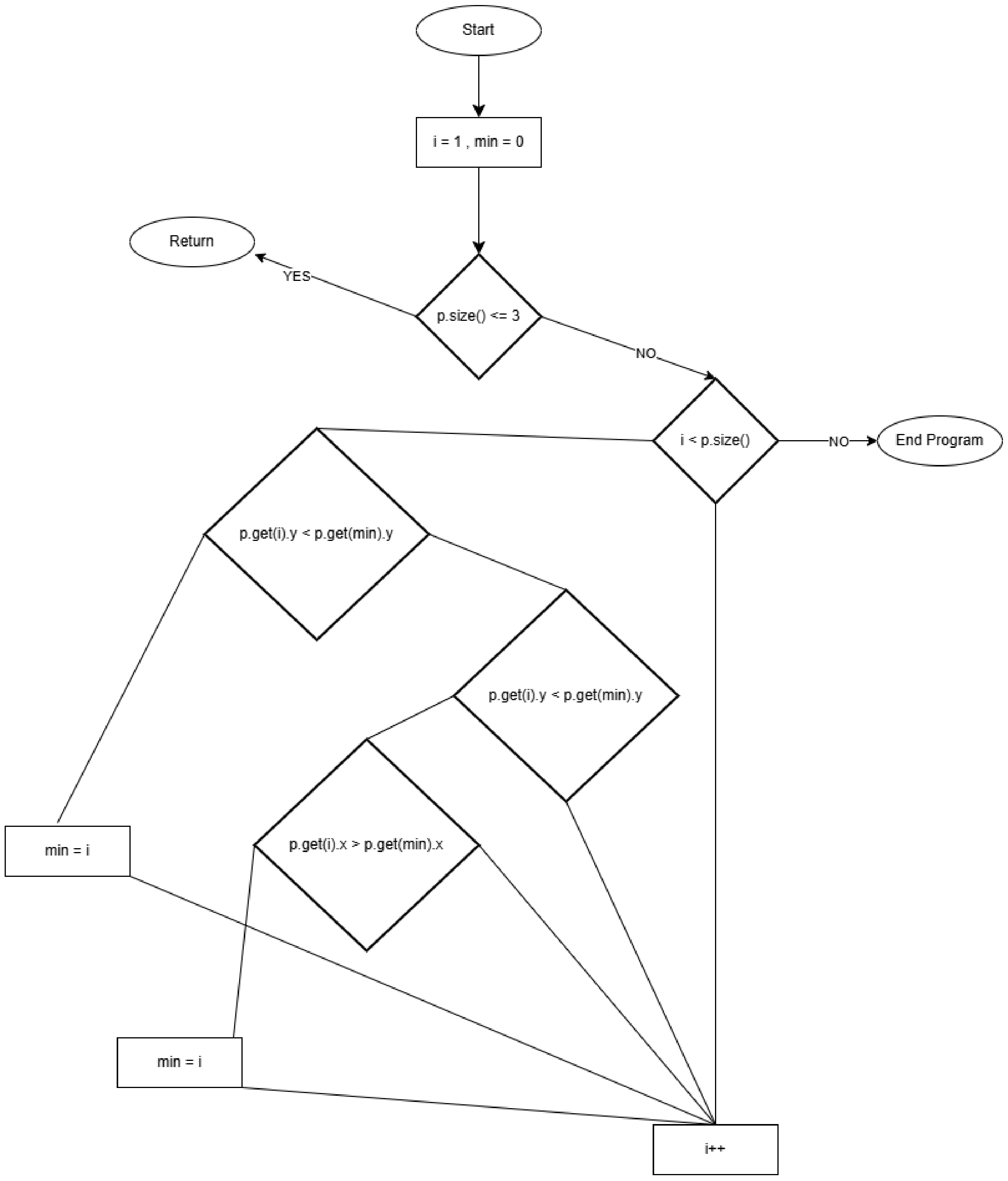
Software Engineering
Lab - 9
Mutation Testing

Name : Tathya Prajapati
ID : 202201170

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.

Ans :-


Start
i = 1 , min = 0
p.size() <= 3
YES
Return
NO
i < p.size()
NO
End Program
p.get(i).y < p.get(min).y
p.get(i).y < p.get(min).y
p.get(i).x > p.get(min).x
min = i
min = i
i++

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++;
        }
    }
};

int 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 that every line in the DoGraham method is executed at least once.

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

- **Input:** A vector `p` containing three or fewer points (e.g., (0, 0), (1, 1), (2, 2)).
- **Expected Result:** The method terminates immediately, validating the initial return statement.

Test Case 2: `p.size() > 3`, with points having distinct y and x coordinates.

- **Input:** A vector `p` with points such as (0, 0), (1, 2), (2, 3), (3, 4).
- **Expected Result:** The loop processes each point to identify the one with the smallest y coordinate, thus executing the body of the loop along with the if-else statements.

Test Case 3: `p.size() > 3`, with points sharing the same y coordinates but differing x values.

- **Input:** Points like (1, 1), (2, 1), (0, 1), (3, 2).
- **Expected Result:** The loop locates the point with the smallest x value among those with identical y values, demonstrating the execution of both y and x comparison operations.

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

- **Input:** Points such as (1, 1), (1, 1), (1, 1), (1, 1).
- **Expected Result:** The loop runs through all points without modifying min since all points are identical, confirming that the loop concludes without any updates to min.

b) Branch Coverage :

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

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

- **Input:** A vector `p` that contains three or fewer points (e.g., (0, 0), (1, 1), (2, 2)).
- **Expected Outcome:** The method exits immediately, confirming 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 one with the lowest y coordinate, executing the loop's body along with the if-else conditions.

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

- **Input:** Points like (1, 1), (2, 1), (0, 1), (3, 2).
- **Expected Outcome:** The loop identifies the point with the smallest x value among those with the same y value, illustrating the execution of both y and x comparison statements.

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

- **Input:** Points such as (1, 1), (1, 1), (1, 1), (1, 1).
- **Expected Outcome:** The loop iterates through all points without changing min, as all points are the same, ensuring the loop completes without any updates to min.

c) Basic Condition Coverage.

Objective: Independently evaluate each atomic condition within the method to cover 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` containing three or fewer points, such as (0, 0), (1, 1), (2, 2).
- Expected Outcome: Confirms the true condition of `p.size() <= 3`.

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

- Input: Points such as (0, 0), (1, 1), (2, 2), (3, 3).
- Expected Outcome: Validates the true outcome for `p[i].y < p[min].y`, as each point has a greater y value than the initial minimum.

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: Confirms the true outcome for `p[i].y == p[min].y` and presents both true and false outcomes for `p[i].x < p[min].x`.

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

- Input: Points such as (1, 1), (1, 1), (1, 1), (1, 1).
- Expected Outcome: Returns false outcomes for `p[i].y < p[min].y`, `p[i].y == p[min].y`, and `p[i].x < p[min].x`, indicating that no changes to min occur.

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: If this condition is absent, the method may execute even when `p.size() <= 3`, potentially leading to unnecessary or erroneous calculations in scenarios where the input size is inadequate. Our existing tests, which only verify when `p.size() > 3`, may overlook this, allowing the mutation to remain unnoticed.

Test Case Needed: A test case where `p.size()` is exactly 3 (for example, with points (0, 0), (1, 1), and (2, 2)) would be useful in identifying this issue, as it would trigger 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)
        {
            <= 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 that have equal y values could incorrectly replace `min`, potentially leading to the wrong selection of the minimum point. Given that the current test cases may not adequately evaluate multiple points with the same y values, this issue could go unnoticed.

Test Case Needed: It would be beneficial to include a test case with several points having identical y values (for instance, (2, 1), (1, 1), (3, 1), (0, 1)) to verify that the smallest x value is chosen correctly. This would highlight the incorrect behavior of selecting the last instance rather than the true minimum**.**

3) Insertion Mutation: Introduce additional statements or conditions within the code.
Mutation Example: Add a line that resets the `min` index at the end of each iteration of the loop.

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;
    }
}
```

Expected Outcome: Resetting `min` after each iteration prevents the code from accurately tracking the actual minimum point discovered so far, as `min` is constantly reset to 0. This can result in an incorrect outcome if the sequences of points do not have the smallest y or x at the beginning.

Test Case Needed: A vector `p` containing points in various positions for minimum values, such as [(5, 5), (1, 0), (2, 3), (4, 2)], will help identify this issue. The correct `min` should be retained across iterations, which will fail because of this mutation.

Task - 4: Develop a test suite that meets the path coverage criterion, ensuring that every loop is traversed 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 will exit immediately without entering the loop.
- **Path Covered:** This covers the scenario where the loop condition fails from the outset, preventing the loop from executing.

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 situation where the loop is bypassed.
- **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 verifies `p.size() > 3`, then enters the loop. During the first iteration, it evaluates (1, 1), updates `min` to the index of the lowest y or x, and then exits.
- **Path Covered:** Covers the scenario where the loop executes once before finishing.

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 examines the first two points in the loop to identify the minimum. After two iterations, it updates `min` and prepares to continue or exit.
- **Path Covered:** This covers the situation where the loop executes exactly twice.

Lab Execution:

Q1) After generating the control flow graph, verify if your CFG aligns with the CFG produced by the Control Flow Graph Factory Tool and the Eclipse flow graph generator.
=> YES

Q2) Determine the minimum number of test cases needed to cover the code using the previously mentioned criteria.

- **Statement Coverage:** 3
- **Branch Coverage:** 3
- **Basic Condition Coverage:** 3
- **Path Coverage:** 3

Summary of Minimum Test Cases:

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