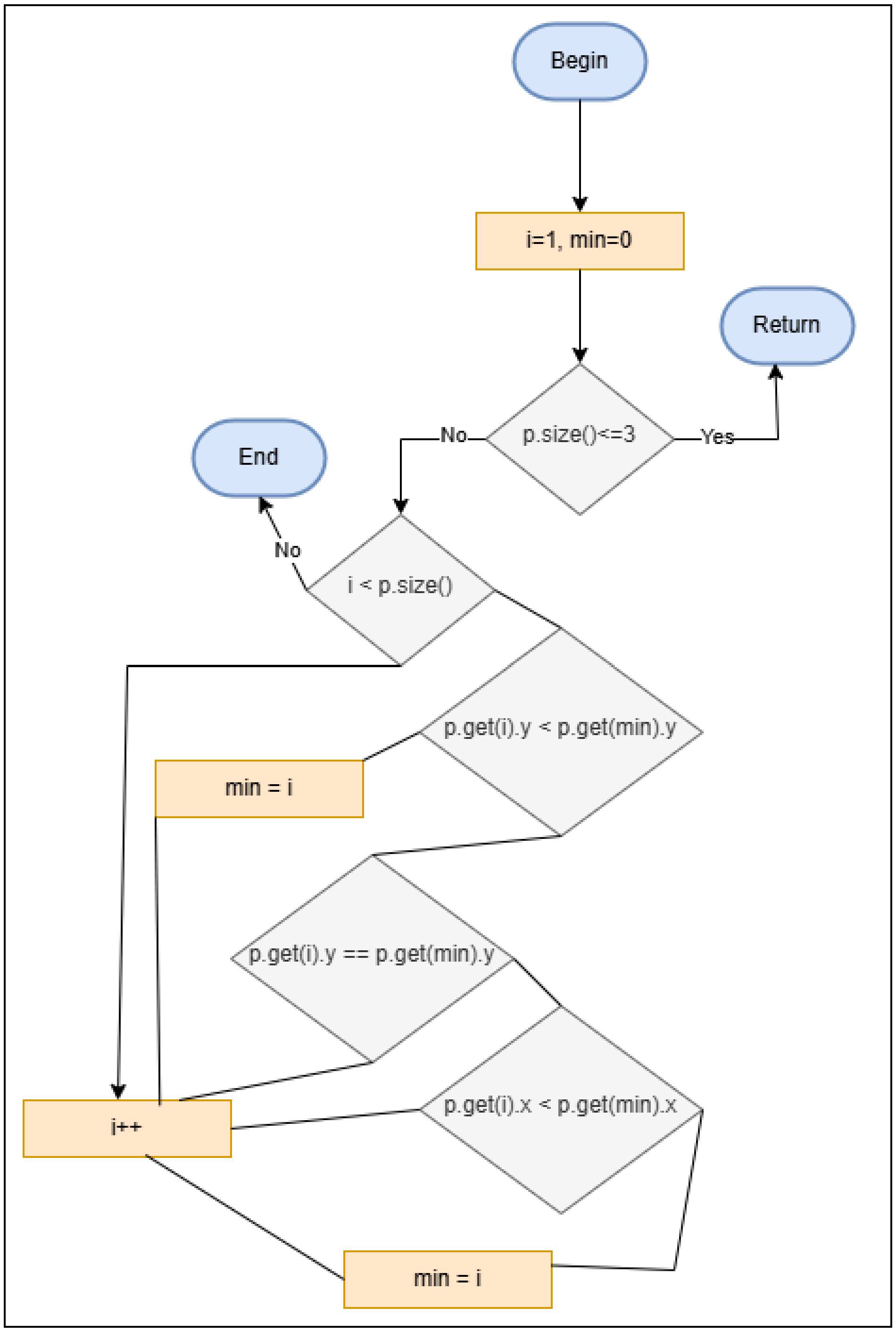
IT314 - software engineering
Lab- 9
Mutation Testing

ID: 202201199
Name: Vrund Leuva

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.

Task-1:


Begin
i=1, min=0
Return
p.size()<=3
No
Yes
End
No
i < p.size()
p.get(i).y < p.get(min).y
min = i
p.get(i).y == p.get(min).y
p.get(i).x < p.get(min).x
i++
min = i

C++ Code:

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

typedef long long ll;
typedef long double ld;

class Point {
public:
    double x, y;
    Point(double xCoord, double yCoord) {
        x = xCoord;
        y = yCoord;
    }
};

class ConvexHullAlgorithm {
public:
    void computeGrahamScan(vector<Point> &points) {
        int n = points.size();
        if (n <= 3) return;
        int lowestIndex = 0;
        for (int i = 1; i < n; i++) {
            if (points[i].y < points[lowestIndex].y || (points[i].y == points[lowestIndex].y
&& points[i].x < points[lowestIndex].x)) {
                lowestIndex = i;
            }
        }
    }
};

int main() {
    vector<Point> points;
    points.push_back(Point(0, 0));
    points.push_back(Point(1, 1));
    points.push_back(Point(2, 2));

    ConvexHullAlgorithm hull;
    hull.computeGrahamScan(points);
}
```

Task - 2: Construct test sets for your flow graph that are adequate for the following criteria:

- **Statement Coverage:**

Objective: Ensure that every line within the `DoGraham` method is executed at least once during testing.

Test Case 1: Small Input Size (`p.size() <= 3`)
Input: A vector `p` containing 3 or fewer points, such as `(0, 0)`, `(1, 1)`, and `(2, 2)`.
Expected Outcome: The method should terminate immediately upon encountering the `return` statement, which handles the case where the number of points is less than or equal to 3.

Test Case 2: Diverse Points (`p.size() > 3` with distinct y and x values)
Input: A vector `p` containing points like `(0, 0)`, `(1, 2)`, `(2, 3)`, and `(3, 4)`.
Expected Outcome: The loop processes each point and correctly identifies the one with the smallest `y` value (and the smallest `x` value in case of a tie), triggering the `if` and `else` conditionals inside the loop.

Test Case 3: Same y-coordinate, different x-values (`p.size() > 3`)
Input: A vector `p` with points such as `(1, 1)`, `(2, 1)`, `(0, 1)`, and `(3, 2)`.
Expected Outcome: The method identifies the point with the smallest `x` value among those

with the same `y` value, ensuring that both the `y` and `x` comparison logic within the loop is executed.

Test Case 4: Identical Points (`p.size() > 3` with all points being the same)
Input: A vector `p` containing identical points like `(1, 1)`, `(1, 1)`, `(1, 1)`, and `(1, 1)`.
Expected Outcome: As all points are identical, the loop will iterate over all points but will not modify the `min` (or `lowestIndex`), confirming that the loop concludes without making any updates to the index of the minimum point.

- ## Branch Coverage:

Objective: Ensure that each decision point (branch) in the `DoGraham` method is exercised at least once during testing. This involves covering both the `if` and `else` branches, ensuring every conditional path is executed.

Test Case 1: Small Number of Points (`p.size() <= 3`)
Input: A vector `p` containing three or fewer points, such as `(0, 0)`, `(1, 1)`, `(2, 2)`.
Expected Outcome: Since there are not enough points to form a convex hull, the method should immediately exit at the `return` statement, confirming that this path (the initial return condition) is covered.

Test Case 2: Multiple Points with Distinct Coordinates (`p.size() > 3`)
Input: A vector `p` with points that have both distinct `y` and `x` values, for example `(0, 0)`, `(1, 2)`, `(2, 3)`, and `(3, 4)`.
Expected Outcome: The method will loop through each point and correctly identify the one with the smallest `y` value, exercising the decision-making logic inside the loop. Both the comparison conditions (for `y` and possibly `x` when `y` is equal) will be triggered.

Test Case 3: Points with Identical `y` Values but Different `x` Values (`p.size() > 3`)
Input: A vector `p` where the points have the same `y` value but different `x` values, such as `(1, 1)`, `(2, 1)`, `(0, 1)`, and `(3, 2)`.
Expected Outcome: The loop will evaluate the points and find the one with the smallest `x` value among those with the same `y` value. This will ensure that both the `y` comparison and the tie-breaking `x` comparison are exercised.

Test Case 4: All Points Identical (`p.size() > 3`)
Input: A vector `p` where all points are identical, like `(1, 1)`, `(1, 1)`, `(1, 1)`, and `(1, 1)`.
Expected Outcome: The loop will iterate over all the points, but since they are all the same, the `min` (or `lowestIndex`) will not change. This verifies that the loop will correctly complete without modifying `min`, ensuring this branch is also covered.

- **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.

Deletion Mutation: Removal of Specific Conditions or Code Lines

Mutation Example: Remove the condition `if (p.size() <= 3)` at the start of the method.

```
void DoGraham(vector<pt> &p)
{
    int i = 1;
    int min = 0;
    // Removed check for small input size (p.size() <= 3)

    // Continue with the loop regardless of the number of points
    while (i < p.size())
    {
        // Find the point with the minimum y value
        if (p[i].y < p[min].y)
        {
            min = i;
        }
        else if (p[i].y == p[min].y)
        {
            // In case of a tie, compare the x values
            if (p[i].x < p[min].x)
            {
                min = i;
            }
        }
        i++;
    }
}
```

Expected Outcome:

- Removing the check `if (p.size() <= 3)` at the beginning of the method results in the method running even when the input size is 3 or smaller.
- For input vectors with `p.size() <= 3`, the loop would still iterate and perform unnecessary computations that aren't required for convex hull calculations on a minimal number of points.
- Our existing tests, which only check when `p.size() > 3`, will fail to detect this issue because they assume the input will be large enough to bypass this check.

Test Case Needed:

To identify this issue, a test case where `p.size()` is exactly 3 would be critical. For instance, using points like `(0, 0)`, `(1, 1)`, and `(2, 2)` will trigger the unnecessary loop behavior when the input size is too small to proceed with the convex hull calculation.

Change Mutation: Modification of Conditions, Variables, or Operators

Mutation Example: Modify the condition by changing the `<` operator to `<=` in the comparison `if (p[i].y < p[min].y)`.

```
void DoGraham(vector<pt> &p)
{
    int i = 1;
    int min = 0;

        if (p.size() <= 3)
    {
        return;
    }

        while (i < p.size())
    {
        // Changed the comparison operator from < to <=
        if (p[i].y <= p[min].y)
        {
            min = i;  // Assign the new minimum point
        }
        else if (p[i].y == p[min].y)
        {
            // In case of tie in y-values, compare by x-coordinate
            if (p[i].x < p[min].x)
            {
                min = i;  // Update min if a smaller x is found
            }
        }
        i++;
    }
}
```

Expected Outcome:

With the condition `if (p[i].y <= p[min].y)`, the code incorrectly selects points with equal `y` values, potentially choosing the last occurrence rather than the correct minimum. This happens because the `<=` condition allows points with equal `y` values to replace the current `min`, while the original logic with `<` would only replace `min` if the `y` value is strictly smaller.

Test Case Needed:

To identify this issue, a test case with points having the same `y` value but different `x` values is crucial. The test should verify that the point with the smallest `x` value is selected as the minimum.

Insertion Mutation: Adding Extra Statements or Conditions

Mutation Example: Introduce a line that resets the `min` index after each iteration of the loop.

```
void DoGraham(vector<pt> &p)
{
    int i = 1;
    int 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; // This effectively resets the progress made in
finding the minimum point
    }
}
```

Expected Outcome:

Introducing the line `min = 0;` at the end of each iteration resets the `min` index back to 0, undoing any progress made in identifying the point with the minimum `y` (and `x` in case of ties). This causes the loop to lose track of the minimum point across iterations.

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 (No iterations)

- **Input:** `p = []` (empty vector).
- **Expected Behavior:** Since `p.size()` is 0, the method will immediately return, bypassing the loop entirely.
- **Path Covered:** This test case ensures that the method handles an empty input correctly by not entering the loop when there are no points to process.

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

- **Input**: `p = [(0, 0)]` (a single point).
- **Expected Behavior**: With `p.size() <= 3`, the method will exit early without entering the loop, as the condition is met.
- **Path Covered**: This test case validates that the method exits prematurely when there is only one point, avoiding unnecessary loop execution.

Test Case 3: p.size() = 4 (Loop executes once)

- **Input**: `p = [(0, 0), (1, 1), (2, 2), (3, 3)]`.
- **Expected Behavior**: Since `p.size()` is greater than 3, the loop will run once. In this iteration, the method checks the second point `(1, 1)` and updates `min` to point to the smallest point based on the `y` value (or `x` in case of ties).

- **Path Covered**: This case demonstrates the behavior of the method when the loop runs a single iteration before it terminates.

Test Case 4: p.size() = 4 (Loop runs twice)

- **Input**: `p = [(0, 0), (1, 2), (2, 1), (3, 3)].`
- **Expected Behavior**: The loop runs twice, first evaluating `(1, 2)` and then `(2, 1)`. The method updates the `min` index as it finds the point with the smallest `y` value and, if necessary, updates it further based on `x`.
- **Path Covered**: This case verifies the method's behavior when the loop needs to run multiple times to correctly determine the smallest point.

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.

Ans: YES

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

Ans.

- 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**