Dhirubhai Ambani
Institute of Information and Communication Technology

Lab 9: Mutation Testing

Jay Rathod - 202201255

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.

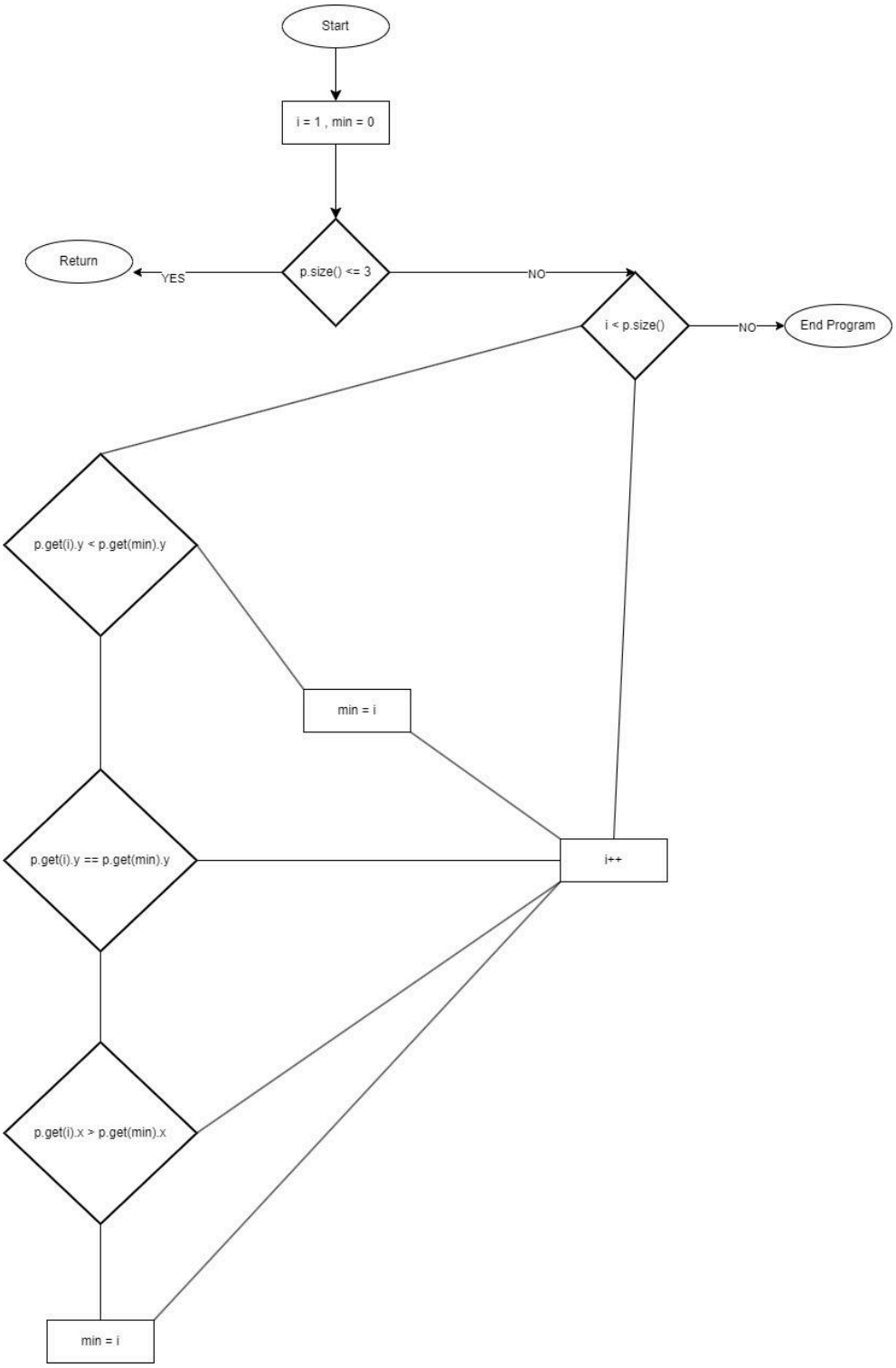
```
Vector doGraham(Vector p) {
        int i,j,min,M;

        Point t;
        min = 0;

        // search for minimum:
        for(i=1; i < p.size(); ++i) {
            if( ((Point) p.get(i)).y <
                        ((Point) p.get(min)).y )
            {
                min = i;
            }
        }

        // continue along the values with same y component
        for(i=0; i < p.size(); ++i) {
            if(( ((Point) p.get(i)).y ==
                        ((Point) p.get(min)).y ) &&
                 (((Point) p.get(i)).x >
                        ((Point) p.get(min)).x ))
            {
                 min = i;
            }
        }
```

Task - 1: Control flow graph:



C++ code:

z_last > hello.cpp > pt > pt(double, double)

```cpp
#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++;
        }
    }
};
```

```
46  int main{
47      vector<pt> jay;
48      jay.pb(pt(0, 0));
49      jay.pb(pt(1, 1));
50      jay.pb(pt(2, 2));
51
52      ConvexHull hull;
53      hull.DoGraham(jay);
54  }
```

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

a) Statement Coverage:

Goal: Ensure that every line in the `DoGraham` method runs at least once.

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

- **Input**: A vector `p` with three or fewer points, such as (0, 0), (1, 1), and (2, 2).
- **Expected Result**: The method exits immediately, covering the initial return condition.

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

- **Input**: A vector `p` with points like (0, 0), (1, 2), (2, 3), (3, 4).
- **Expected Result**: The loop processes each point, identifying the point with the smallest y value and covering the loop body and related conditions.

Test Case 3: `p.size() > 3` with identical y values but distinct x values

- **Input**: A vector `p` containing (1, 1), (2, 1), (0, 1), (3, 2).
- **Expected Result**: The loop finds the point with the lowest x value among those with the same y, covering the comparison logic for y

and x values.

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

- **Input**: A vector `p` with points like (1, 1), (1, 1), (1, 1), (1, 1).
- **Expected Result**: The loop runs without any updates to `min`, ensuring completion without changes to `min`.

b) Branch Coverage:

Goal: Verify that each decision point is evaluated for both true and false outcomes.

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

- **Input**: A vector `p` with points like (0, 0), (1, 1), (2, 2).
- **Expected Result**: The method returns immediately, covering the false branch of the loop entry condition.

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

- **Input**: A vector `p` with points such as (0, 0), (1, 1), (2, 2), (3, 3).
- **Expected Result**: The true branch of the main condition (`p[i].y < p[min].y`) is tested as each point has a unique y value.

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

- **Input**: Points such as (1, 1), (2, 1), (0, 1), (3, 2).
- **Expected Result**: Covers both true for the main condition (when y is lower) and true/false for the secondary condition (same y, different x).

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

- **Input**: Points like (1, 1), (1, 1), (1, 1), (1, 1).
- **Expected Result**: The loop completes without updates to `min`, covering the false branch for all comparisons.

c) Basic Condition Coverage:

Goal: Evaluate each atomic condition separately to confirm all possible outcomes are achieved.

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 three or fewer points, such as (0, 0), (1, 1), (2, 2).
- **Expected Result**: Verifies the true outcome of `p.size() <= 3`.

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

- **Input**: Points like (0, 0), (1, 1), (2, 2), (3, 3).
- **Expected Result**: Verifies the true condition for `p[i].y < p[min].y`.

Test Case 3: `p.size() > 3` with identical y values but distinct x values

- **Input**: Points such as (0, 1), (2, 1), (3, 1), (1, 0).
- **Expected Result**: Verifies the true condition for `p[i].y == p[min].y` and tests both true and false outcomes for `p[i].x < p[min].x`.

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

- **Input**: Points like (1, 1), (1, 1), (1, 1), (1, 1).
- **Expected Result**: Confirms false outcomes for `p[i].y < p[min].y`, `p[i].y == p[min].y`, and `p[i].x < p[min].x`, as no changes occur to `min`.

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 Result: Without this condition, the method might still execute even when `p.size() <= 3`, potentially leading to unnecessary or incorrect operations due to insufficient input. The current tests that focus on cases where `p.size() > 3` could overlook this and fail to catch the mutation.

Required Test Case: A test where `p.size()` equals exactly 3 (e.g., points (0, 0), (1, 1), and (2, 2)) would identify the issue, as it would lead to the method performing unwarranted calculations.

Mutation Change

Mutation Type: Modify a condition, variable, or operator in the code.

Example of Mutation: Replace the `<` operator with `<=` 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 values could 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 y values (e.g., (2, 1), (1, 1), (3, 1), (0, 1)) to ensure the smallest x is selected correctly. 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 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 causes the code to lose track of the actual minimum point found so far, as `min` is always set back to 0. This leads to incorrect results when processing sequences of points where the smallest y or x value does not appear at the start of the vector.

Required Test Case: A vector `p` containing points with varying positions for minimum values, such as [(5, 5), (1, 0), (2, 3), (4, 2)], would help detect this mutation. The correct `min` should be maintained across iterations, and the test would fail if it is not.

Task 4: Path Coverage Test Set

Objective: Create a test set that achieves path coverage, ensuring that each loop runs zero, one, or two times.

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

- Input: `p = []` (an empty vector)
- Expected Outcome: The method exits immediately as `p.size() == 0`, without entering the loop.
- Path Covered: Covers the path where the loop condition fails initially, resulting in no loop execution.

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

- Input: `p = [(0, 0)]` (a single point)
- Expected Outcome: The method returns immediately due to `p.size() <= 3`, ensuring the loop is skipped.
- Path Covered: Confirms that the method exits early when `p.size() <= 3`.

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

- Input: `p = [(0, 0), (1, 1), (2, 2), (3, 3)]`
- Expected Outcome: The method checks `p.size() > 3`, enters the loop, evaluates the first point (1, 1), updates `min` to reflect the current

lowest y or x, and then exits.
- Path Covered: Ensures that the loop executes once before completion.

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

- Input: `p = [(0, 0), (1, 2), (2, 1), (3, 3)]`
- Expected Outcome: The method processes the first two points in the loop to identify the minimum value. After two iterations, `min` is updated, and the loop prepares to continue or exit.
- Path Covered: Confirms the path where the loop runs exactly twice.

Lab Execution:

Q1) Compare your control flow graph (CFG) with those generated by the Control Flow Graph Factory Tool and Eclipse flow graph generator.

- Result: YES, the CFGs match.

Q2). Determine the minimum number of test cases needed to achieve full coverage using the given criteria.

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

Summary of Minimum Test Cases:

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