IT313: Software Engineering

# Lab - 9 Mutation Testing

Preet Dave - 202201072

5th November, 2024

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;
            }
        }
```

Q1. 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.

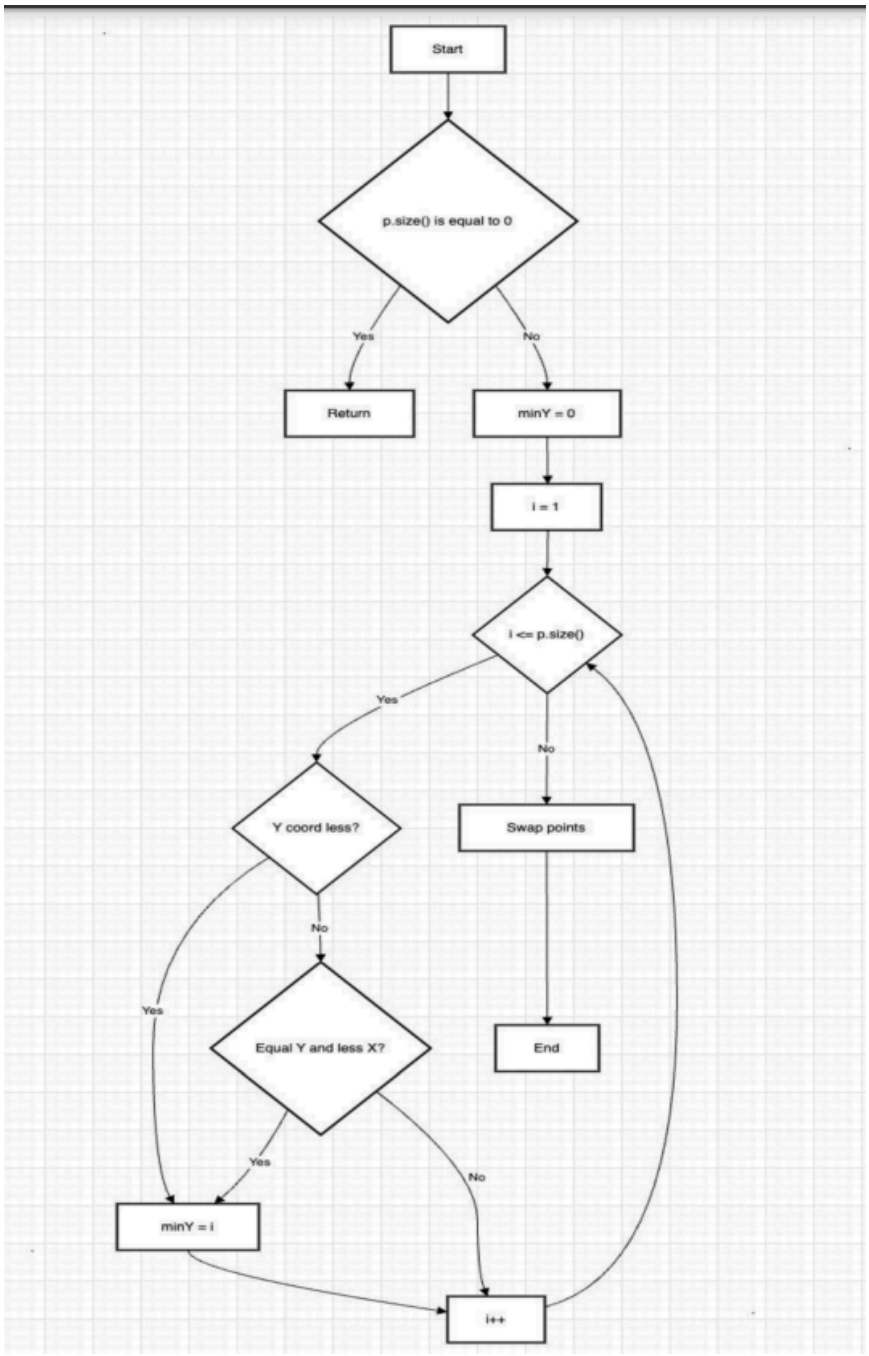
- Code for the doGraham method :-

```
#include <iostream>

#include <vector>

class Point {

public:

    int x, y;

    Point(int x, int y) : x(x), y(y) {}

    friend std::ostream& operator<<(std::ostream& os, const Point& point)
{

        os << "(" << point.x << ", " << point.y << ")";

        return os;

    }

};

void doGraham(std::vector<Point>& points) {

    int i, min = 0;

    std::cout << "Searching for the minimum y-coordinate..." << std::endl;

    for (i = 1; i < points.size(); ++i) {

        std::cout << "Comparing " << points[i] << " with " << points[min]
<< std::endl;

        if (points[i].y < points[min].y) {
```

```
            min = i;
            std::cout << "New minimum found: " << points[min] <<
std::endl;
        }
    }
    std::cout << "Searching for the leftmost point with the same minimum
y-coordinate..." << std::endl;
    for (i = 0; i < points.size(); ++i) {
        std::cout << "Checking if " << points[i] << " has the same y as "
<< points[min] << " and a smaller x..." << std::endl;
        if (points[i].y == points[min].y && points[i].x < points[min].x) {
            min = i;
            std::cout << "New leftmost minimum point found: " <<
points[min] << std::endl;
        }
    }
    std::cout << "Final minimum point: " << points[min] << std::endl;
}
int main() {
    std::vector<Point> points;
    points.emplace_back(1, 2);
    points.emplace_back(3, 1);
    points.emplace_back(0, 1);
    points.emplace_back(-1, 1);
    doGraham(points);
```

```
    return 0;

}
```

- Control Flow Diagram :-



Q2. Construct test sets for your flow graph that are adequate for the following criteria:

a. Statement Coverage.

➔ Test Case 1 :-

- Inputs: Any list p with more than one point (e.g., [(0, 1), (1, 2), (2, 0)])
- This will traverse through the entire flow, covering statements related to finding the minimum y-coordinate and leftmost minimum point.

➔ Test Case 2 :-

- Inputs: [(2, 2), (2, 2), (3, 3)]
- This checks for points with the same y-coordinate and ensures the leftmost point logic executes.

b. Branch Coverage.

➔ Test Case 1 :-

- Inputs: [(0, 1), (1, 2), (2, 0)]
- This will take the true branch for finding the minimum y-coordinate.

➔ Test Case 2 :-

- Inputs: [(2, 2), (2, 2), (3, 3)]

- This will test the scenario where y-coordinates are equal, triggering the branch for checking x-coordinates.

➔ Test Case 3 :-

- Inputs: [(1, 2), (1, 1), (2, 3)]
- This ensures the flow takes the false branch when checking for new minimum y-coordinates and the leftmost check.

c. Basic Condition Coverage.

➔ Test Case 1 :-

- Inputs: [(1, 1), (2, 2), (3, 3)]
- This will evaluate both conditions for the y-coordinate comparisons.

➔ Test Case 2 :-

- Inputs: [(1, 1), (1, 1), (1, 2)]
- This checks the scenario where the y-coordinates are the same but evaluates the x-coordinate condition.

➔ Test Case 3 :-

- Inputs: [(3, 1), (2, 2), (1, 3)]

- This ensures that both conditions in the loop are executed, confirming the function's logic is robust.

Q3. 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.

➔ Delete Mutation :-

```
if (points[i].y < points[min].y) {

    min = i;

}

min = i;
```

- Analysis for Statement Coverage:
  - If the condition check is deleted, the code always assigns i to min, which could lead to an incorrect outcome. However, this may not cause a detectable failure if no specific test validates the selection of the minimum y value.
  - **Potential Undetected Outcome** : If the test set only checks if min is assigned without verifying the correctness of the chosen min value, the deletion might go unnoticed.

➔ Change Mutation :-

```
if (points[i].y < points[min].y)
```

```
if (points[i].y <= points[min].y)
```

- Analysis for Branch Coverage:
  - Changing < to <= could cause the code to mistakenly assign min = i even if p.get(i).y equals p.get(min).y, potentially selecting an incorrect point as the minimum.
  - **Potential Undetected Outcome** : If the test set does not specifically validate cases where p.get(i).y equals p.get(min).y, the mutation could produce a subtle fault without detection.

➔ Insertion Mutation :-

```
min = i;
```

```
min = i + 1;
```

- Analysis for Basic Condition Coverage:
  - Adding an unnecessary increment (i + 1) changes the intended assignment, leading min to point to an incorrect index, potentially out of the array bounds.

- **Potential Undetected Outcome** : If the test set does not validate that min is correctly assigned to the expected index without additional increments, this mutation might not be detected. Tests only checking if min is assigned (rather than validating correctness) might miss this error.

Q4. 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 :-

◆ Loop Explored Zero Times

- Input : An empty vector ρ.
- Test : Vector<Point> ρ = new Vector<Point>();
- Expected Result : The method should return immediately without any processing. This covers the condition where the vector size is zero, leading to the exit condition of the method.

➔ Test Case 2 :-

◆ Loop Explored Once

- Input : A vector with one point.
- Test : Vector<Point> ρ = new Vector<Point>(); ρ.add(new Point(0, 0));
- Expected Result : The method should not enter the loop since ρ.size() is 1. It should swap the first point with itself, effectively leaving the vector

unchanged. This test case covers the scenario where the loop iterates once.

➔ Test Case 3 :-

◆ Loop Explored Twice

- Input : A vector with two points where the first point has a higher y-coordinate than the second.
- Test : Vector<Point> p = new Vector<Point>(); p.add(new Point(1, 1)); p.add(new Point(0, 0));
- Expected Result : The method should enter the loop and compare the two points, finding that the second point has a lower y-coordinate. Thus, minY should be updated to 1, and a swap should occur, moving the second point to the front of the vector.

➔ Test Case 4 :-

◆ Loop Explored More Than Twice

- Input : A vector with multiple points.
- Test : Vector<Point> p = new Vector<Point>(); p.add(new Point(2, 2)); p.add(new Point(1, 0)); p.add(new Point(0, 3));
- Expected Result : The loop should iterate through all three points. The second point will have the lowest y-coordinate, so minY will be updated to 1. The swap will place the point with coordinates (1, 0) at the front of the vector.

2. 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.

- Control Flow Graph Factory :- YES

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

- Statement Coverage: 3 test cases
    - Branch Coverage: 3 test cases
    - Basic Condition Coverage: 3 test cases
    - Path Coverage: 3 test cases
- Summary of Minimum Test Cases :
    - Sum:
        - 3 (Statement) + 3 (Branch) + 2 (Basic Condition) + 3 (Path) = 11 test cases

Q3 and Q4 are to be done in same way as part 1