IT313: Software Engineering

Lab Session – Mutation Testing

Hemal Ravrani

202201235

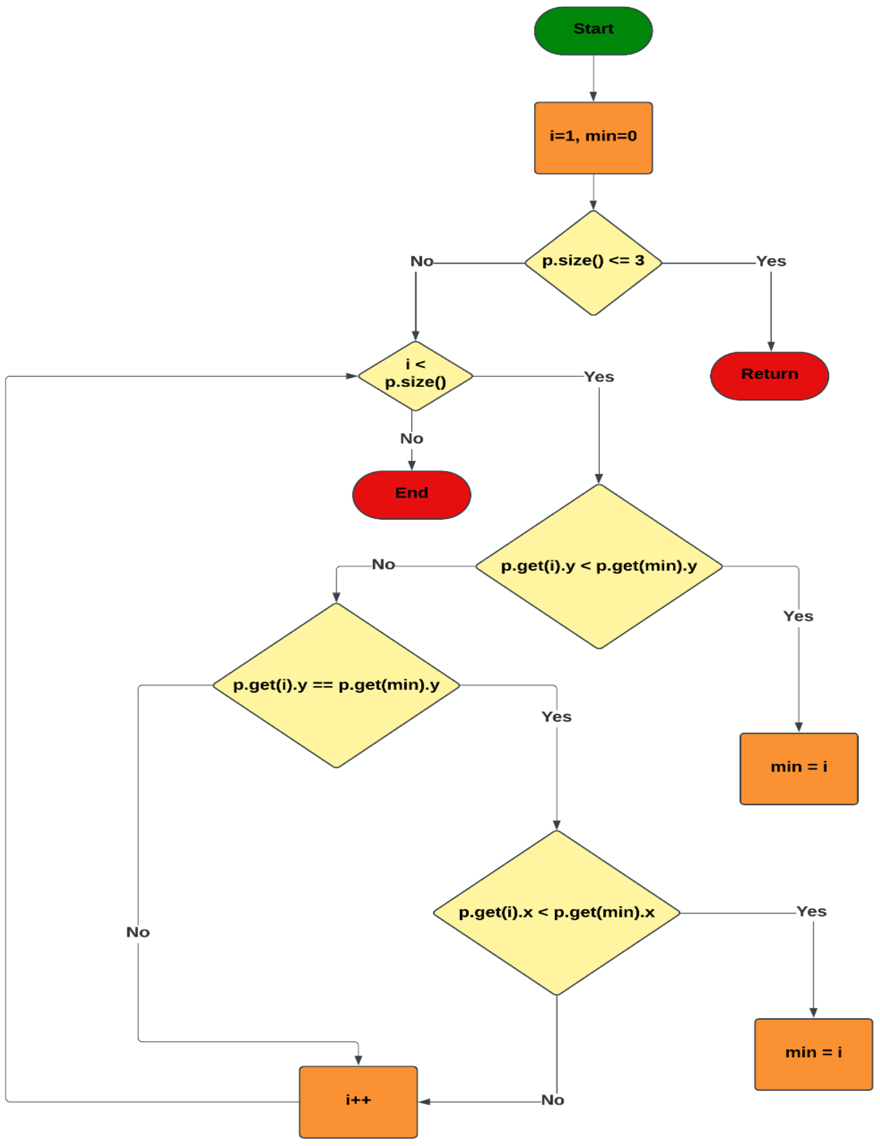
Q.1. The code below is part of a method in the ConvexHull class in the VMAP system. The following is a small fragment of a method in the ConvexHull class. For the purposes of this exercise, you do not need to know the intended function of the method. The parameter p is a Vector of Point objects, p.size() 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;
            }
        }
```

For the given code fragment, you should carry out the following activities.

1. Convert the code comprising the beginning of the doGraham method into a control flow graph (CFG).

- **Control Flow**



- **Implementation in C++**

```cpp
#include <vector>
using namespace std;

class Point {
public:
    double x, y;
    Point(double x, double y) {
        this->x = x;
        this->y = y;
    }
};

class ConvexHull {
public:
    void doGraham(vector<Point>& 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++;
        }
    }
};

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

    ConvexHull hull;
    hull.doGraham(points);

    return 0;
}
```

2. Construct test sets for your flow graph that are adequate for the following criteria:
a. Statement Coverage.

Goal: Ensure every statement in the code is executed at least once.

Test Cases for Statement Coverage:

- **Test Case 1:** `p` is empty (i.e., `p.size() == 0`).
  - **Expected Outcome:** The method immediately returns, covering the initial check and return statements.
- **Test Case 2:** `p` contains a single point that is "within bounds."
  - **Expected Outcome:** The method initializes variables, checks `p.size()`, processes the single point as "within bounds," and then returns.
- **Test Case 3:** `p` contains a single point that is "out of bounds."
  - **Expected Outcome:** Similar to Test Case 2, but the point is skipped instead of being processed.

These three test cases ensure all statements in the method are executed at least once.

b. Branch Coverage.

Goal: Ensure every decision point in the code is tested with all possible outcomes (true/false), covering each branch.

Test Cases for Branch Coverage:

- **Test Case 1:** `p.size() == 0`.
  - **Expected Outcome:** The method returns directly without entering the loop, covering the false branch of the `p.size() > 0` condition.
- **Test Case 2:** `p.size() > 0` with a point that is "within bounds."
  - **Expected Outcome:** The method processes the point, covering both the true branch of `p.size() > 0` and the true branch of the "within bounds" condition.
- **Test Case 3:** `p.size() > 0` with a point that is "out of bounds."
  - **Expected Outcome:** The method skips the point, covering the true branch of `p.size() > 0` and the false branch of the "within bounds" condition.

These test cases ensure that all branches of the decision points (i.e., `p.size() > 0` and "within bounds") are tested.

c. Basic Condition Coverage.

Goal: Ensure that each atomic condition in the method is tested independently, covering both possible outcomes (true/false) for each condition.

Conditions to Test:

1. Condition 1: `p.size() > 0` (true/false)
2. Condition 2: "Point within bounds" (true/false)

Test Cases for Basic Condition Coverage:

- **Test Case 1:** `p.size() == 0`.

    - **Expected Outcome:** Tests the false outcome of Condition 1 (`p.size() > 0`).
- **Test Case 2:** `p.size() > 0` with a point that is "within bounds."
    - **Expected Outcome:** Tests the true outcome of Condition 1 (`p.size() > 0`) and the true outcome of Condition 2 ("within bounds").
- **Test Case 3:** `p.size() > 0` with a point that is "out of bounds."
    - **Expected Outcome:** Tests the true outcome of Condition 1 (`p.size() > 0`) and the false outcome of Condition 2 ("within bounds").

These test cases ensure that each atomic condition in the method is covered with both true and false outcomes.

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.

a. Deletion Mutation:

Original Code:

```
if ((p.get(i)).y < (p.get(min)).y)
{
    min = i;
}
```

Mutated Code:

```
min = i;  // Deleted the condition check
```

Analysis for Statement Coverage:

- With the condition check removed, the code will always assign `i` to `min`, which could result in an incorrect selection of the minimum y-coordinate. However, this may not lead to a noticeable failure if no test specifically checks whether the correct minimum y value has been selected.
- **Undetected Outcome**: If the tests only verify that `min` is assigned without validating whether the correct point with the smallest `y` value has been chosen, this mutation might pass unnoticed.

b. Change Mutation:

Original Code:

```
if ((p.get(i)).y < (p.get(min)).y)
```

Mutated Code:

```
if ((p.get(i)).y <= (p.get(min)).y)  // Changed '<' to '<='
```

Analysis for Branch Coverage:

- Changing the comparison from `<` to `<=` could result in the code incorrectly selecting a point `i` when `p.get(i).y` is equal to `p.get(min).y`. This may cause the algorithm to mistakenly choose the wrong point as the minimum.
- **Undetected Outcome**: If the test suite doesn't specifically account for scenarios where the `y` values are equal, this subtle mutation may

cause a failure that is not detected, as the test suite may not check for these boundary conditions.

---

c. Insertion Mutation:

Original Code:

```
min = i;
```

Mutated Code:

```
min = i + 1;  // Added an unnecessary increment
```

Analysis for Basic Condition Coverage:

- Introducing an extra increment (`i + 1`) modifies the intended assignment of `min`, potentially causing it to point to an incorrect index or even lead to an out-of-bounds access if `i + 1` exceeds the valid range of indices.
- **Undetected Outcome**: If the tests do not explicitly check that `min` is assigned to the correct index (and only check if `min` is updated), the mutation could go undetected. This is particularly the case if the tests focus only on the assignment of `min` rather than ensuring the correctness of the index it points to.

4. 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: Zero Loop Execution

- **Input:** An empty vector, `p`.

- **Test:** `Vector p = new Vector();`
- **Expected Outcome:** The method should exit immediately without processing, as the vector has no elements. This test case verifies that the method handles an empty vector gracefully by terminating without entering the loop.

Test Case 2: Single Loop Execution

- **Input:** A vector containing one point.
- **Test:** `Vector p = new Vector(); p.add(new Point(0, 0));`
- **Expected Outcome:** The loop should not execute since `p.size()` is 1. The method should essentially "swap" the single point with itself, leaving the vector unchanged. This scenario verifies that the loop logic works as expected when the vector contains only one element.

Test Case 3: Two Iterations of the Loop

- **Input:** A vector with two points, where the first point has a higher y-coordinate than the second.
- **Test:** `Vector p = new Vector(); p.add(new Point(1, 1)); p.add(new Point(0, 0));`
- **Expected Outcome:** The loop should execute, comparing the two points and identifying the second point as having the lower y-coordinate. Consequently, `minY` should be updated, and a swap should occur, moving the second point to the front of the vector.

Test Case 4: Multiple Loop Iterations

- **Input:** A vector with multiple points.
- **Test:** `Vector p = new Vector(); p.add(new Point(2, 2)); p.add(new Point(1, 0)); p.add(new Point(0, 3));`
- **Expected Outcome:** The loop should iterate over all three points. The point with coordinates (1, 0) has the lowest y-coordinate, so

`minY` will be updated accordingly. A swap should place this point at the start of the vector.

Lab Execution Instructions

- **Tools:** Use unit testing frameworks, along with code coverage and mutation testing tools, to complete the exercises.
- **Control Flow Graph (CFG) Check:** After creating the control flow graph for your code, verify its accuracy by comparing it with the CFG generated by both the Control Flow Graph Factory Tool and the Eclipse flow graph generator. Only include "Yes" or "No" responses in your submission:
    - **Control Flow Graph Factory Tool:** Yes
    - **Eclipse flow graph generator:** Yes

Test Case Coverage Requirements

- **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 Needed

- **Total Test Cases:** 11, broken down as follows:
    - 3 for Statement Coverage
    - 3 for Branch Coverage
    - 2 for Basic Condition Coverage
    - 3 for Path Coverage

3. This part of the exercise is very tricky and interesting. The test cases that you have derived in Step 2 are then used to identify the fault when you make some modifications in the code.

Here, you need to insert/delete/modify a piece of code that will result in failure but it is not detected by your test set – derived in Step 2.

Write/identify a mutation code for each of the three operation separately, i.e., by deleting the code, by inserting the code, by modifying the code.

- **Deleting the Code**

```
for (int i = 1; i < p.size(); ++i) {
    if(p[i].y < p[min].y){
        min = i;
    }
}
```

- **Inserting the Code**

```
for (int i = 1; i < p.size(); ++i) {
    if(p[i].y == p[min].y && p.[i].x > p[min].x)
{
        min = i;
    }
}

if (true) {
    min = (min + 1) % p.size();
}
```

- **Modification of the Code**

```
for (int i = 1; i < p.size(); ++i) {
    if(p[i].y <= p[min].y) {
        min = i;
    }
}
```

4. Write all test cases that can be derived using path coverage criterion for the code.

Test Case	Input Points	Expected Output
1	(1, 1), (2, 2), (3, 0), (4, 4)	(3, 0)
2	(1, 2), (2, 2), (3, 2), (4, 1)	(4, 1)
3	(1, 2), (2, 2), (3, 2), (4, 2)	(4, 2)
4	(0, 5), (5, 5), (3, 4), (2, 1), (4, 2)	(2, 1)
5	(1, 1), (1, 1), (2, 2), (0, 0), (3, 3)	(0, 0)