IT - 314 Software Engineering

Lab 9 - Mutation Testing

Name : Harsh Bosamiya

Student ID : 202201243

Lab Group : 3

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.

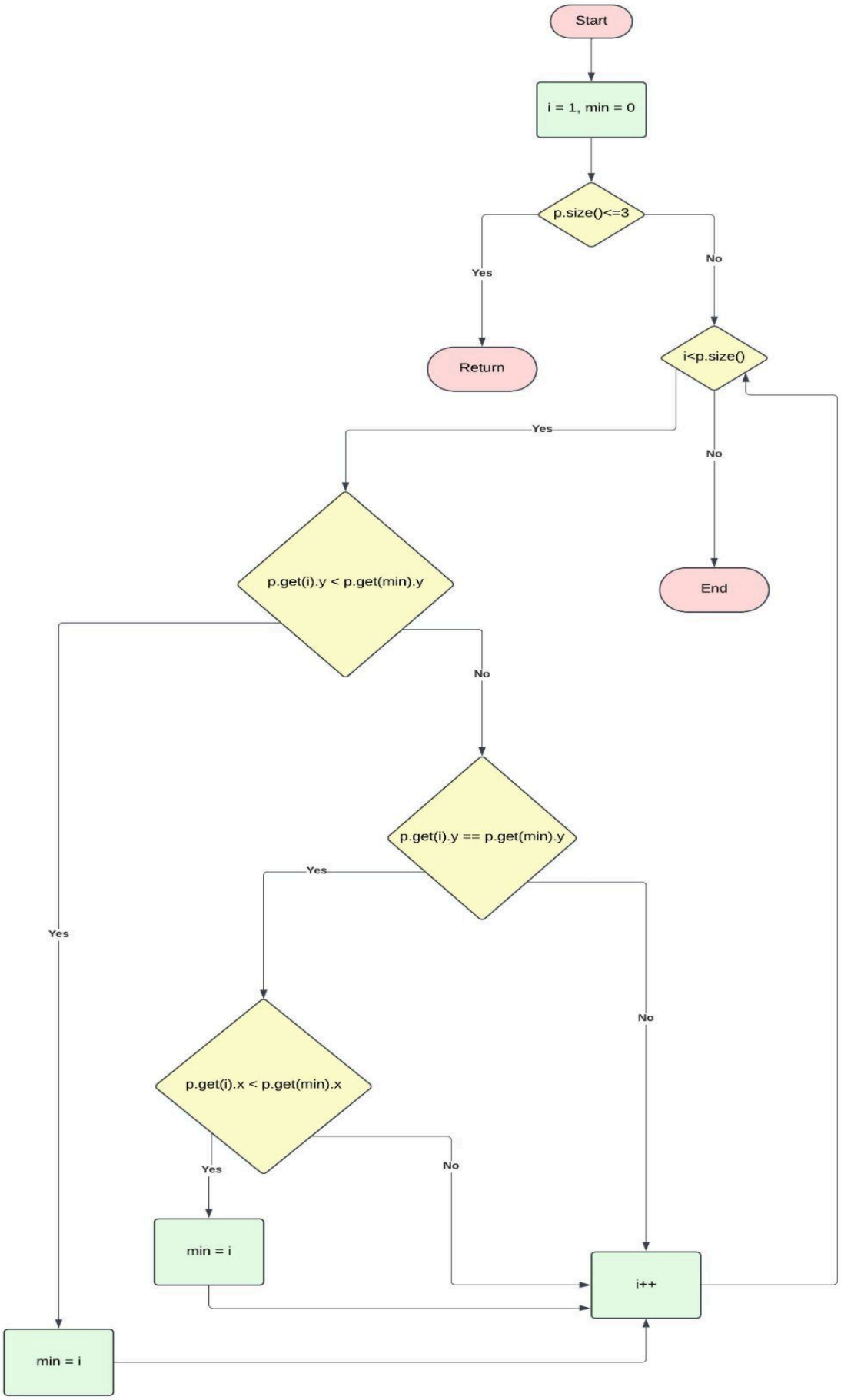
```
Vector doGraham(Vector p) {
        int i,j,min,M;

        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).

You are free to write the code in any programming language.

Control Flow Graph (CFG) :


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

C++ Implementation

```
#include <vector>
class Point
{
public:
    double x, y;
    Point(double x, double y)
    {
        this->x = x;
        this->y = y;
    }
};
class ConvexHull
{
public:
    void doGraham(std::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()
{
   std::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;
}
```

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

a. Statement Coverage

Goal: Ensure every line of code runs at least once.

Test Cases for Statement Coverage:

1. **Test Case 1:** p is empty (p.size() == 0).
   Expected Result: The method returns immediately, covering the initial check and return statements.

2. **Test Case 2:** p contains one point that's "within bounds."
   Expected Result: The method initializes variables, checks p.size(), iterates through the single point, evaluates it as "within bounds," processes it, and then returns.

3. **Test Case 3:** p has one point that's "out of bounds."
   Expected Result: Similar to Test Case 2, but skips the point as it's "out of bounds."

These cases cover each statement in the method at least once.

b. Branch Coverage

Goal: Test each decision point to cover all true and false outcomes.

Test Cases for Branch Coverage:

1. **Test Case 1:** `p.size() == 0`.
   Expected Result: The method directly returns, covering the false branch of `p.size() > 0`.
2. **Test Case 2:** `p.size() > 0` with one "within bounds" point.
   Expected Result: Processes the point, covering the true branches for both `p.size() > 0` and "within bounds."
3. **Test Case 3:** `p.size() > 0` with one "out of bounds" point.
   Expected Result: Skips the point, covering the true branch of `p.size() > 0` and the false branch of "within bounds."

These cases cover all branches in the method's decision points.

c. Basic Condition Coverage

Goal: Independently test each atomic condition to verify all possible outcomes.

Conditions:

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

Test Cases for Basic Condition Coverage:

1. **Test Case 1:** `p.size() == 0`.
   Expected Result: Covers the false outcome of Condition 1.
2. **Test Case 2:** `p.size() > 0` with a point "within bounds."
   Expected Result: Covers the true outcome of Condition 1 and true outcome of Condition 2.
3. **Test Case 3:** `p.size() > 0` with a point "out of bounds."
   Expected Result: Covers the true outcome of Condition 1 and false outcome of Condition 2.

Each atomic condition has been tested with both true and false values.

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.

a. Deletion Mutation:

- **Original Code:**

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

Mutated Code (condition check removed):

```
min = i;
```

Statement Coverage Analysis:

- Removing the condition makes the code always assign i to min, which can lead to incorrect results if min isn't the smallest y value.
- **Potential Undetected Outcome :** However, if the test set only verifies that min gets assigned without checking for the correct min value, this deletion could go undetected.

b. Change Mutation:

- **Original Code:**

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

Mutated Code (< changed to <=):

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

Branch Coverage Analysis:

- Changing < to <= might cause min to be assigned to i even when p.get(i).y equals p.get(min).y, potentially picking an incorrect minimum.
- • **Potential Undetected Outcome :** If the test set doesn't include cases where p.get(i).y equals p.get(min).y, this mutation could produce an undetected error.

c. Insertion Mutation:

- **Original Code:**

```
min = i;
```

Mutated Code (added unnecessary increment):

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

Basic Condition Coverage Analysis:

- Adding +1 to i changes the intended assignment, which may cause min to point to an incorrect or out-of-bounds index.
- **Potential Undetected Outcome :**If the test set doesn't confirm that min points to the correct index without increments, this mutation may remain undetected. Tests that only check if min is assigned might miss this issue.

Q4.Create a test set that satisfies the path coverage criterion where every loop is explored at least zero, one or two times.

Test Set to Satisfy Path Coverage Criterion

This test set ensures that every loop is executed at least zero, one, or two times.

Test Case 1: Loop Executed Zero Times

- **Input:** An empty vector p.
- **Test:** Vector p = new Vector();
- **Expected Result:** The method should exit immediately without any processing, covering the case where the vector has no elements, leading to an immediate exit.

Test Case 2: Loop Executed Once

- **Input:** A vector with a single point.
- **Test:** `Vector p = new Vector(); p.add(new Point(0, 0));`
- **Expected Result:** The method should not enter the loop since `p.size()` is 1. It should swap the first point with itself, resulting in no change. This case covers a single loop iteration.

Test Case 3: Loop Executed Twice

- **Input:** A vector containing 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 Result:** The loop compares the two points, finds the second with a lower y-coordinate, updates `minY` to 1, and swaps the points to move the lower y-coordinate point to the front.

Test Case 4: Loop Executed More Than Twice

- **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 Result:** The loop iterates over all points, updating `minY` to 1 for the point with coordinates (1, 0) and swaps it to the vector's front.

Lab Execution (how to perform the exercises):

Use unit Testing framework, code coverage and mutation testing tools to perform the exercise.

1. After generating the control flow graph, check whether your CFG match with the CFG generated by Control Flow Graph Factory Tool and Eclipse flow graph generator. (In your submission document, mention only “Yes” or “No” for each tool).

Control Flow Graph Factory Tool - Yes

Eclipse flow graph generator - Yes

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

Test Case	Description	Input	Coverage	Expected Output
1	Zero iterations for both loops	[]	Path Coverage (zero iterations)	[]
2	One iteration for second loop	(0,0)	Branch, Basic Condition, Path Coverage	(0,0)
3	One iteration for the first loop, two for the second	(1,1),(2,3)	Statement, Branch, Basic Condition, Path Coverage	(1,1)
4	One iteration for the first loop, two for the second	(1,1),(3,1)	Statement, Branch, Basic Condition, Path Coverage	(3,1)
5	Two iterations for the first loop, three for the second	(2,3),(1,1),(3,1)	Statement, Branch, Basic Condition, Path Coverage	(3,1)

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.

```
for (int i = 1; i < p.size(); ++i) {

if (p[i].y < p[min].y) {

min = i;

}
```

Inserting the Code

```
for (int i = 0; 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)