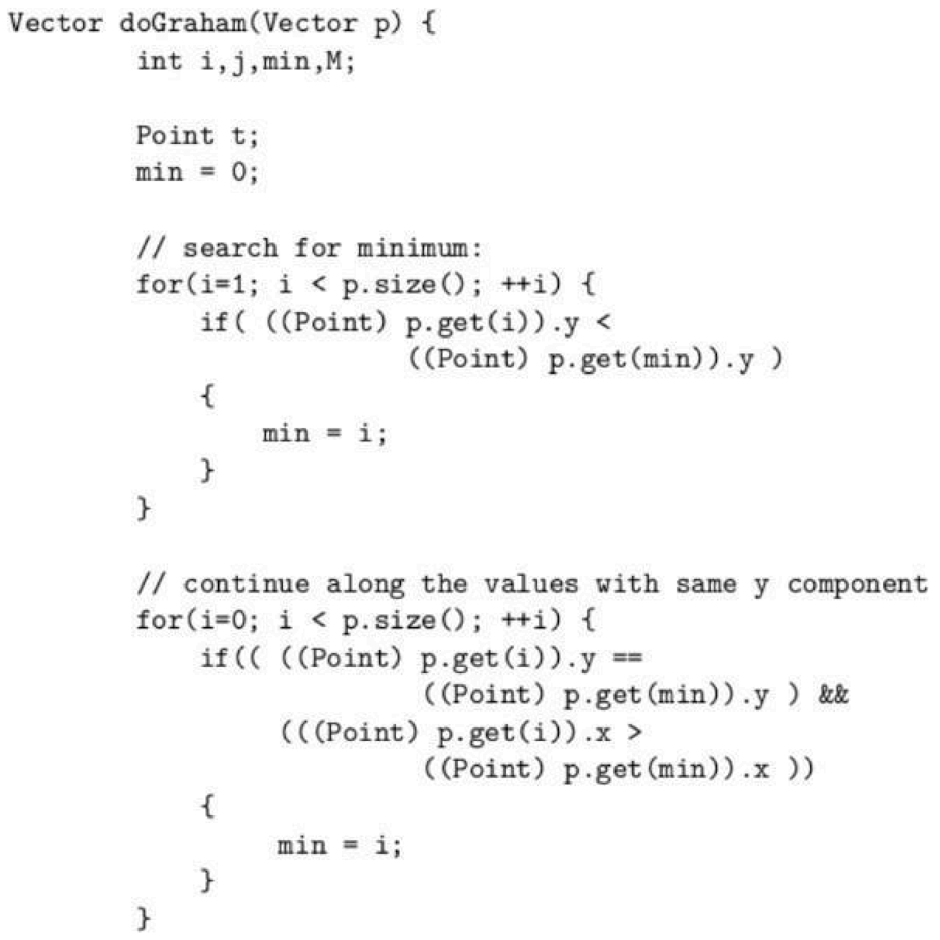
IT314 : Software Engineering

Name : Het Gandhi
Student ID : 202201167

Lab 9 – Mutation Testing

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.



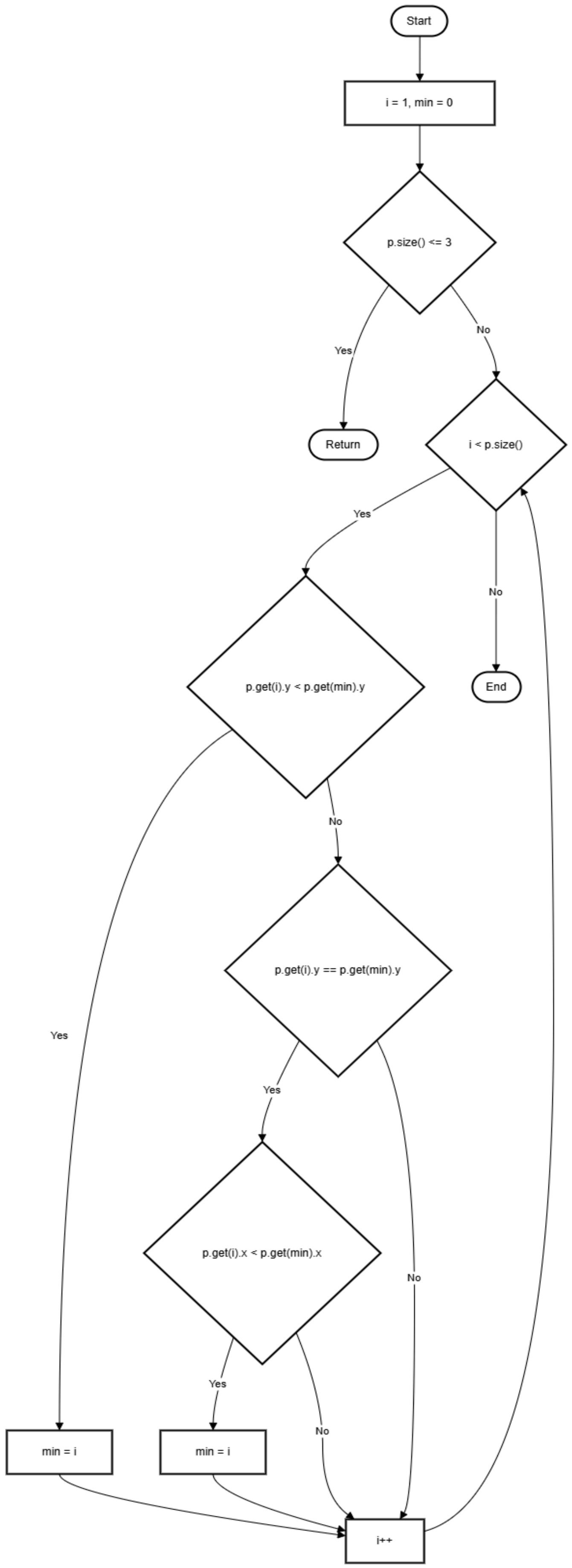
```
Vector doGraham(Vector p) {
        int i,j,min,M;

        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.

❖ CFG (Control Flow Graph) : -


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++

❖ Implementation in C++ : -

```
#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;

}
```

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

a. Statement Coverage.
b. Branch Coverage.
c. Basic Condition Coverage.

Solution :

a. Statement Coverage

Objective : Ensure that 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 directly returns, covering the initial check and return statements.

Test Case 2 : p contains one point that is "within bounds."

- Expected Outcome : The method initializes variables, checks p.size(), iterates through the single point, evaluates it as "within bounds," processes it, and then returns.

Test Case 3 : p contains one point that is "out of bounds."

- Expected Outcome: Similar to Test Case 2, except the point is skipped instead of processed.

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

b. Branch Coverage

Objective : Test each decision point with all possible outcomes (true/false) to cover every branch.

Test Cases for Branch Coverage :

Test Case 1 : p.size() == 0.

- Expected Outcome : The method directly returns without entering the loop, covering the false branch of the p.size() > 0 check.

Test Case 2 : p.size() > 0 with one point that is "within bounds."

- Expected Outcome : The method processes the point, covering the true branch of both p.size() > 0 and "within bounds."

Test Case 3 : p.size() > 0 with one 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 "within bounds."

These test cases ensure that all branches in the method's decision points (p.size() > 0 and "within bounds") are covered.

c. Basic Condition Coverage

Objective : Test each atomic condition within the method independently to cover all possible outcomes of each condition.

Conditions :

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 : Covers the false outcome of Condition 1.

Test Case 2 : p.size() > 0 with a point that is "within bounds."

- Expected Outcome : Covers the true outcome of Condition 1 and true outcome of Condition 2.

Test Case 3 : p.size() > 0 with a point that is "out of bounds."

- Expected Outcome : Covers the true outcome of Condition 1 and false outcome of Condition 2.

Each atomic condition has been covered with both true and false values in these test cases.

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

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

{

        min = i;

}

// Mutated - deleted the condition check

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.

b. Change Mutation :

```
// Original

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

// Mutated - changed < to <=

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

<u>c. Insertion Mutation :</u>

// Original

min = i;

// Mutated - added unnecessary increment

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.

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

Solution :

Test Case 1 : Loop Explored Zero Times

- Input: An empty vector p.
- Test: Vector p = new Vector();
- 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 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, 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 p = new Vector(); 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 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 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.

<u>Lab Execution (how to perform the exercises)</u>

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 matches 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).

1. Control Flow Graph Factory Tool - Yes
2. Eclipse flow graph generator - Yes

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

Statement Coverage : 3 test cases

1. Branch Coverage: 3 test cases
2. Basic Condition Coverage: 3 test cases
3. Path Coverage: 3 test cases

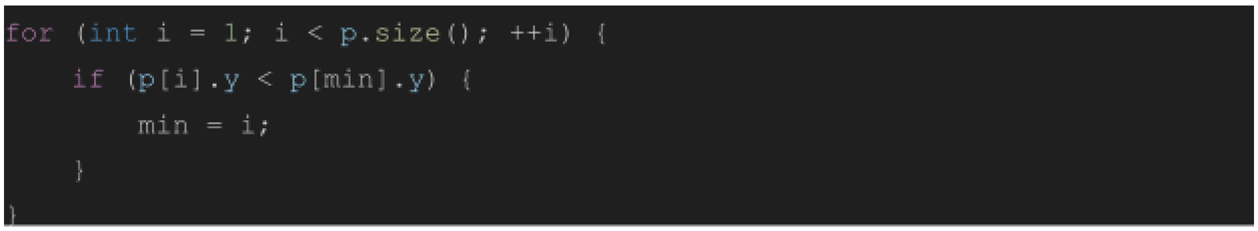
Summary of Minimum Test Cases:

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

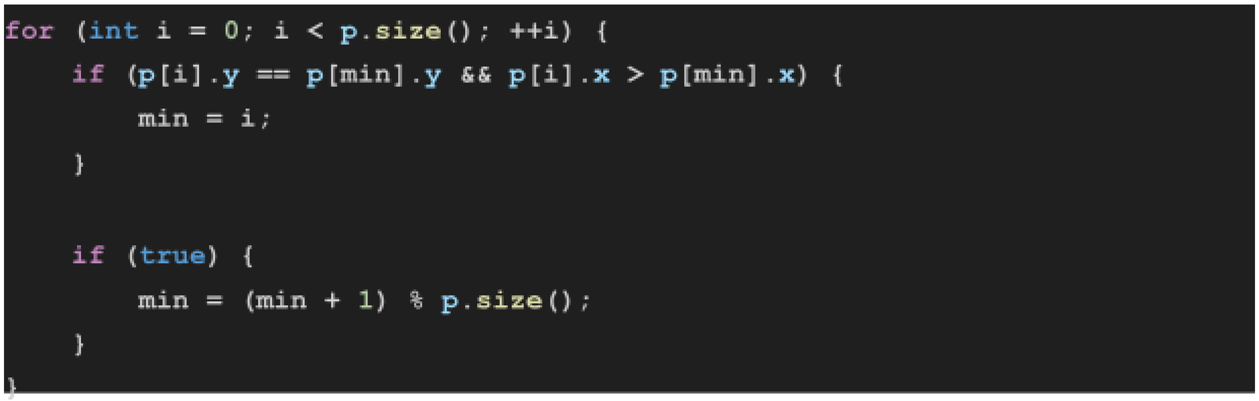
Test Case	Description	Input	Coverage	Expected Output
1	Zero iterations for both loops	[ ]	Path Coverage (zero iterations)	[ ]
2	One iteration for second loop	(0,0)	Branch, Basic Condition, Path Coverage	(0,0)
3	One iteration for the first loop, two for the second	(1,1),(2,3)	Statement, Branch, Basic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

<u>Deleting the Code</u>

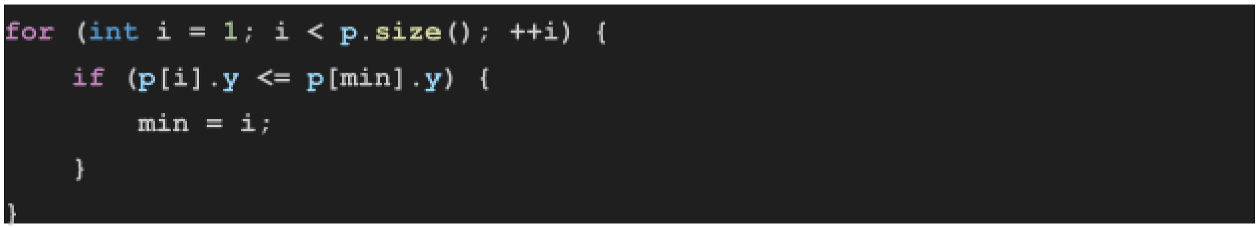

```
for (int i = 1; i < p.size(); ++i) {
    if (p[i].y < p[min].y) {
        min = i;
    }
}
```

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)