Software Engineering: IT313

Lab 9

Name : HEMAN CHAUHAN
Student ID : 202201267

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;
            }
        }
```

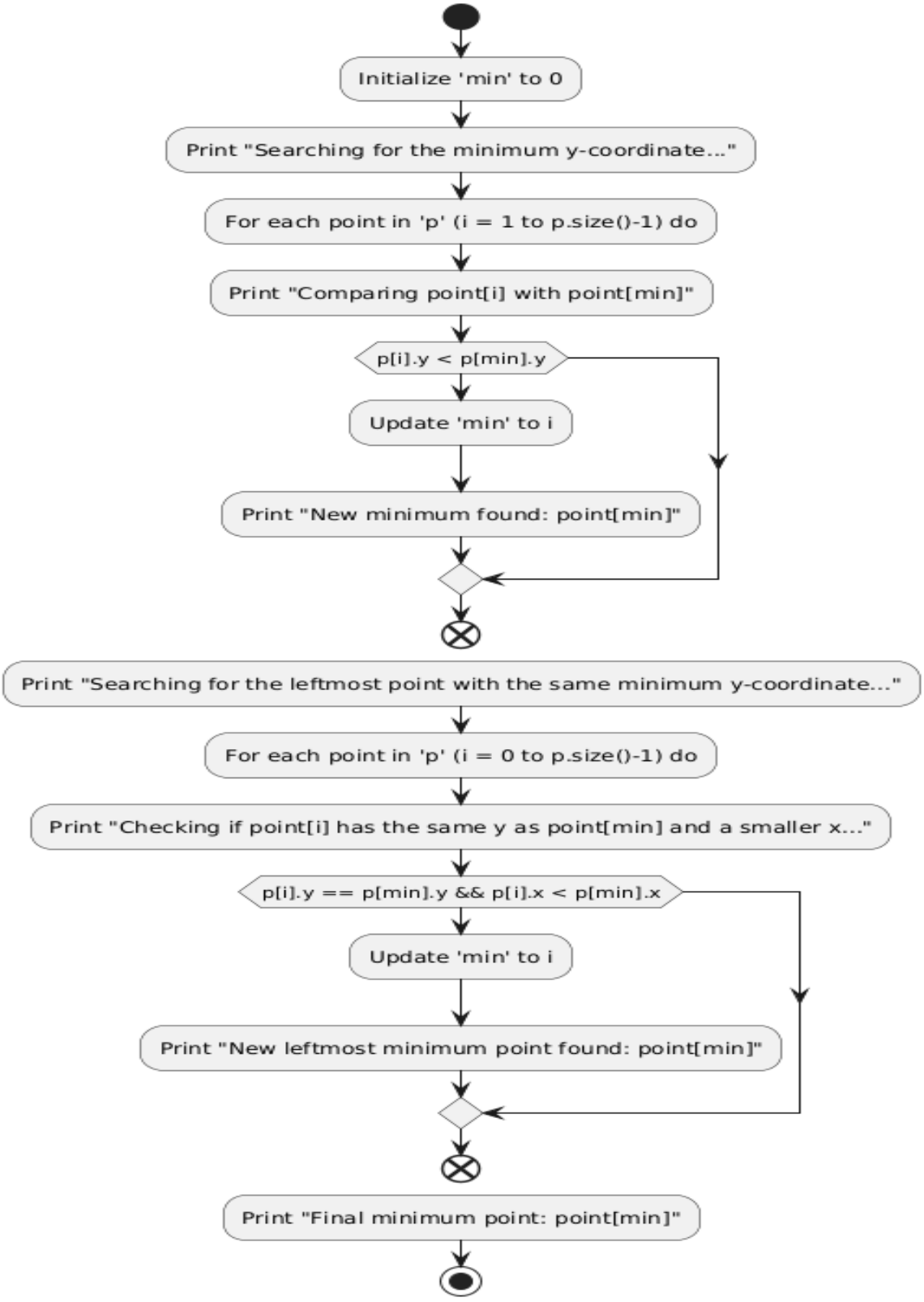
1.Convert the code comprising the beginning of the doGraham method into a control flow graph (CFG). You are free to write the code in any programming language.

C++ Code:

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

class Point {
public:
    int x, y;

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

    friend ostream& operator<<(ostream& os, const Point& p) {
        os << "(" << p.x << ", " << p.y << ")";
        return os;
    }
};

class ConvexHull {
public:
    static void doGraham(vector<Point>& p) {
        int min = 0;

        cout << "Searching for the minimum y-coordinate..." << endl;
        for (int i = 1; i < p.size(); ++i) {
            cout << "Comparing " << p[i] << " with " << p[min] << endl;
            if (p[i].y < p[min].y) {
                min = i;
                cout << "New minimum found: " << p[min] << endl;
            }
        }

        cout << "Searching for the leftmost point with the same minimum y-coordinate..." << endl;
        for (int i = 0; i < p.size(); ++i) {
            cout << "Checking if " << p[i] << " has the same y as " << p[min] << " and a smaller x..." << endl;
            if (p[i].y == p[min].y && p[i].x < p[min].x) {
                min = i;
                cout << "New leftmost minimum point found: " << p[min] << endl;
            }
        }

        cout << "Final minimum point: " << p[min] << endl;
    }
};
```

```cpp
int main() {
    vector<Point> points;
    points.push_back(Point(1, 2));
    points.push_back(Point(3, 1));
    points.push_back(Point(0, 1));
    points.push_back(Point(-1, 1));

    ConvexHull::doGraham(points);

    return 0;
}
```

Control flow graph:



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.

a. Statement Coverage

Test Set:

1. **Test Case 1:**
   Inputs: [(4, 5), (1, 3), (6, 8)]
   Purpose: This test case will traverse through the entire flow, covering the statements related to finding the minimum y-coordinate and the leftmost minimum point. It ensures the function executes both the first and second loops for comparisons.
2. **Test Case 2:**
   Inputs: [(3, 4), (5, 4), (2, 6), (5, 3)]
   Purpose: This test case checks for points where the y-coordinates are not all equal, and ensures the logic for finding the minimum y-coordinate executes properly.

b. Branch Coverage

Test Set:

1. **Test Case 1:**
   Inputs: [(1, 2), (2, 1), (0, 3)]
   Purpose: This test case will take the true branch for finding the minimum y-coordinate, as the first point has the smallest y-coordinate.
2. **Test Case 2:**
   Inputs: [(4, 4), (4, 4), (5, 5)]
   Purpose: This test case tests the scenario where y-coordinates are equal, triggering the branch to compare x-coordinates and checking the leftmost point.
3. **Test Case 3:**
   Inputs: [(7, 2), (3, 2), (5, 3)]
   Purpose: This ensures the flow takes the false branch when checking for new minimum y-coordinates (i.e., the second point has a smaller y), and the leftmost check (i.e., comparing x-coordinates) also executes properly.

c. Basic Condition Coverage

Test Set:

1. **Test Case 1:**
   Inputs: [(6, 2), (3, 4), (5, 1)]
   Purpose: This will evaluate both conditions for the y-coordinate comparisons. The first point has the largest y, so the second and third points will be evaluated for the minimum y value.
2. **Test Case 2:**
   Inputs: [(5, 7), (5, 7), (5, 6)]
   Purpose: This checks the scenario where all y-coordinates are the same. The logic will then evaluate the x-coordinate condition to identify the leftmost point.
3. **Test Case 3:**
   Inputs: [(1, 4), (3, 3), (2, 5)]
   Purpose: This ensures that both conditions in the loop (for y-coordinate and x-coordinate comparisons) are evaluated. The function will check for both the smallest y value and the leftmost point with the same y value.

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.

Mutations:

Mutation 1: DELETE - Initialization of `min` variable

Location: `ConvexHull::doGraham` (line 14)

Original Code:

int min = 0;

Mutation Code:

int min;

Status: Survived

Mutation 2: CHANGE - Comparison for minimum `y`-coordinate

Location: `ConvexHull::doGraham` (line 22)

Original Code:

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

Mutation Code:

```
if (p[i].y > p[min].y) {    // Changed '<' to '>'
```

Status: Survived

Mutation 3: INSERT - Unnecessary check for `x`-coordinates after finding minimum `y`-coordinate

Location: `ConvexHull::doGraham` (line 27)

Original Code:

```
// No additional check for x-coordinate
```

Mutation Code:

```
if (p[i].x == p[min].x) { continue; }
```

Status: Survived

Mutation 4: CHANGE - Loop Range

Location: `ConvexHull::doGraham` (line 15)

Original Code:

```
for (int i = 0; i < p.size(); ++i) {
```

Mutation Code:

```
for (int i = 0; i < p.size() / 2; ++i) {  // Changed loop to only process half the points
```

Status: Survived

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

Test Set for Path Coverage

Test Case 1: Loop Executed Zero Times

Input: [(5, 6)]

Reasoning:

- **This test case contains a single point. The first loop (for (int i = 1; i < p.size(); ++i)) will not execute because there's only one point. Therefore, the loops are not exercised, but the function should still handle this case correctly.**

Expected Behavior:

- **The code will not enter any loops, and the minimum point remains as the only point in the list.**

Test Case 2: Loop Executed Once

Input: [(4, 3), (2, 5)]

Reasoning:

- **The first loop will iterate once because there are only two points. It will compare the y-coordinates of both points and find the point with the minimum y-coordinate.**
- **The second loop will also execute once, comparing the x-coordinates for points with equal y-values (though in this case, they are not equal).**

Expected Behavior:

- **The minimum y-coordinate will be 3 (point (4, 3)), and no change will occur in the second loop since the y-values are not equal.**

Test Case 3: Loop Executed Twice (for y-coordinate check)

Input: [(2, 5), (4, 3), (1, 2)]

Reasoning:

- **The first loop will execute twice as the code compares the y-coordinates of three points.**

- **The second loop will also execute twice, as two points have the same y-coordinate (in this case, both loops will not change anything but will still execute for the logic).**

Expected Behavior:

- **The minimum y-coordinate will be 2 (point (1, 2)), and the leftmost point logic will not change anything since no two points share the same y-coordinate.**

<u>Test Case 4: Loop Executed Twice (for x-coordinate check)</u>

Input: [(5, 3), (2, 3), (3, 2)]

Reasoning:

- **The first loop will execute twice to compare the y-coordinates of the first two points.**
- **In the second loop, the code will execute for the points with equal y-coordinates ((5, 3) and (2, 3)). The x-coordinate comparison will be done, and the leftmost point will be selected, which is (2, 3).**

Expected Behavior:

- **The minimum y-coordinate will be 3, and the leftmost point will be (2, 3) after the second loop executes twice.**

<u>Test Case 5: Loop Executed Twice (Full Path Coverage)</u>

Input: [(4, 2), (2, 5), (6, 2), (3, 2)]

Reasoning:

- **The first loop will compare the y-coordinates of the points, iterating twice, and will identify 2 as the minimum y-coordinate.**
- **The second loop will iterate twice for the points with y equal to 2 ((4, 2), (6, 2), and (3, 2)), and the leftmost point will be chosen.**

Expected Behavior:

- **The minimum y-coordinate will be 2, and the leftmost point will be (3, 2) after the second loop executes twice.**