Software Engineering Lab - 9

Sahil Vaghasiya

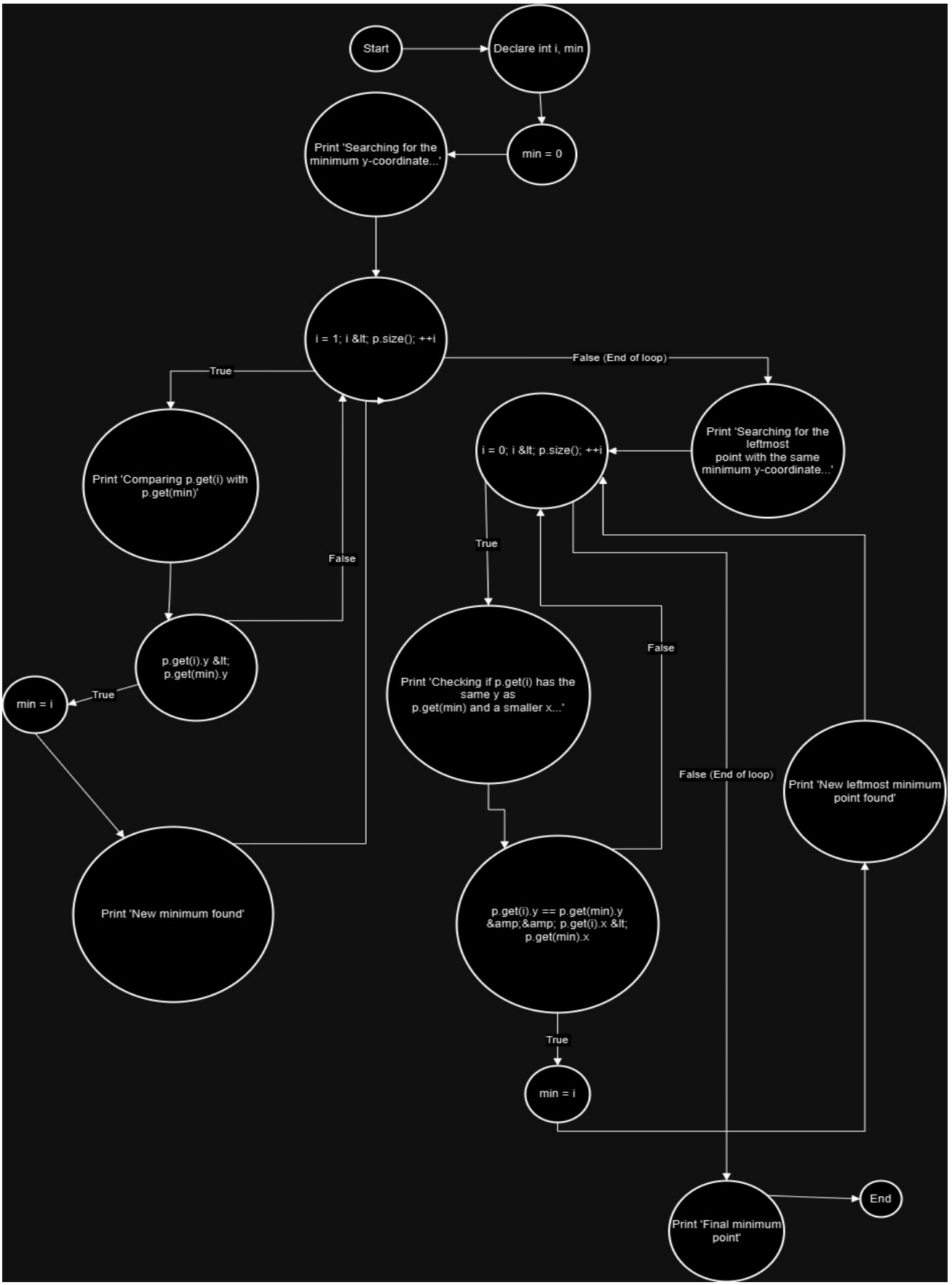
202201083

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.

Code:

```
#include <vector>
#include <algorithm>

struct Point {
    int x, y;
};

vector<Point> doGraham(std::vector<Point> p) {
    int min = 0;
    // Search for the point with the minimum y-coordinate
    for (int i = 1; i < p.size(); ++i) { if (p[i].y < p[min].y) {
            min = i;
        }
    }

    // If there are multiple points with the same y-coordinate, select the one with the
largest x-coordinate
    for (int i = 0; i < p.size(); ++i) { if (p[i].y ==
        p[min].y && p[i].x > p[min].x) {
            min = i;
        }
    }

    // Move the minimum point to the beginning
    swap(p[0], p[min]);

    return p;
}
```

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



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.

```
# Define the Point class
 class Point:
def __init__(self, x, y): # Constructor fixed to double underscore __init__ self.x
= x       self.y = y
          def __repr__(self): # Fixed the representation method to double underscore
__repr__          return f"Point(x={self.x}, y={self.y})"

# Define the do_graham
function def do_graham(p):
min_idx = 0

# Find the point with the minimum y-coordinate     for
i in range(1, len(p)):    if p[i].y < p[min_idx].y:
          min_idx = i

# If there are points with the same y-coordinate, choose the one with the
minimum x-coordinate for i in range(len(p)): if p[i].y == p[min_idx].y and p[i].x >
p[min_idx].x: min_idx = i

# Returning the identified minimum point for clarity
return p[min_idx]

# Define the test cases def
run_tests():
test_cases = [
          # Test case 1 - Statement Coverage
          [Point(2, 3), Point(1, 2), Point(3, 1)],

          # Test cases for Branch Coverage
          [Point(2, 3), Point(1, 2), Point(3, 1)], # Branch True in both conditions
```

```
        [Point(3, 3), Point(4, 3), Point(5, 3)], # Branch False in both conditions

        # Test cases for Basic Condition Coverage
        [Point(2, 3), Point(1, 2), Point(3, 1)], # p[i].y < p[min_idx].y is True
        [Point(1, 3), Point(2, 3), Point(3, 3)], # p[i].y < p[min_idx].y is False
        [Point(2, 2), Point(1, 2), Point(3, 2)], # p[i].y == p[min_idx].y is True, p[i].x <
p[min_idx].x is True
        [Point(3, 2), Point(4, 2), Point(2, 2)], # p[i].y == p[min_idx].y is
True, p[i].x < p[min_idx].x is False
]

# Run each test case for i, points in
enumerate(test_cases, start=1):
      min_point = do_graham(points)          print(f"Test Case {i}: Input Points =
{points}, Minimum Point = {min_point}")

# Run the tests if
__name__ == "__main__":
run_tests()
```

Output :

```
PS D:\python> python -u "d:\python\test_do_graham.py"
Test Case 1: Input Points = [Point(x=2, y=3), Point(x=1, y=2), Point(x=3, y=1)], Minimum Point = Point(x=3, y=1)
Test Case 2: Input Points = [Point(x=2, y=3), Point(x=1, y=2), Point(x=3, y=1)], Minimum Point = Point(x=3, y=1)
Test Case 3: Input Points = [Point(x=3, y=3), Point(x=4, y=3), Point(x=5, y=3)], Minimum Point = Point(x=5, y=3)
Test Case 4: Input Points = [Point(x=2, y=3), Point(x=1, y=2), Point(x=3, y=1)], Minimum Point = Point(x=3, y=1)
Test Case 5: Input Points = [Point(x=1, y=3), Point(x=2, y=3), Point(x=3, y=3)], Minimum Point = Point(x=3, y=3)
Test Case 6: Input Points = [Point(x=2, y=2), Point(x=1, y=2), Point(x=3, y=2)], Minimum Point = Point(x=3, y=2)
Test Case 7: Input Points = [Point(x=3, y=2), Point(x=4, y=2), Point(x=2, y=2)], Minimum Point = Point(x=4, y=2)
PS D:\python>
```

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[i].y < p[min_idx].y:
min_idx = i
```

// Mutated - deleted the condition check

```
min_idx = i
```

Analysis for Statement Coverage:

- **Impact of Removing Condition Check:** If the condition check is removed, the code will always assign `i` to `min`, potentially leading to an incorrect result. However, this issue might not be detected if the test cases only verify that `min` is assigned without specifically validating the correct selection of the minimum `y` value.
- **Risk of Undetected Errors:** If the test set only checks that `min` has been assigned, without confirming that the assigned value is indeed the correct minimum, this error may go unnoticed.

a. Change Mutation:

// Original

```
if p[i].y < p[min_idx].y:
```

// Mutated - changed < to <=

```
if p[i].y <= p[min_idx].y:
```

Analysis for Branch Coverage:

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

b. Insertion Mutation:

// Original

```
min_idx = i
```

// Mutated - added unnecessary increment

```
min_idx = 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.

Ans.

Test Case 1:

Loop Explored Zero Times

- Input: An empty vector p.

- Test: Vector<Point> p = 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> p = new Vector<Point>(); 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<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.

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.

Ans. Control Flow Graph Factory: Yes

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

Ans. 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