Software Engineering (IT314)

[Mutation Testing]

Lab - 9

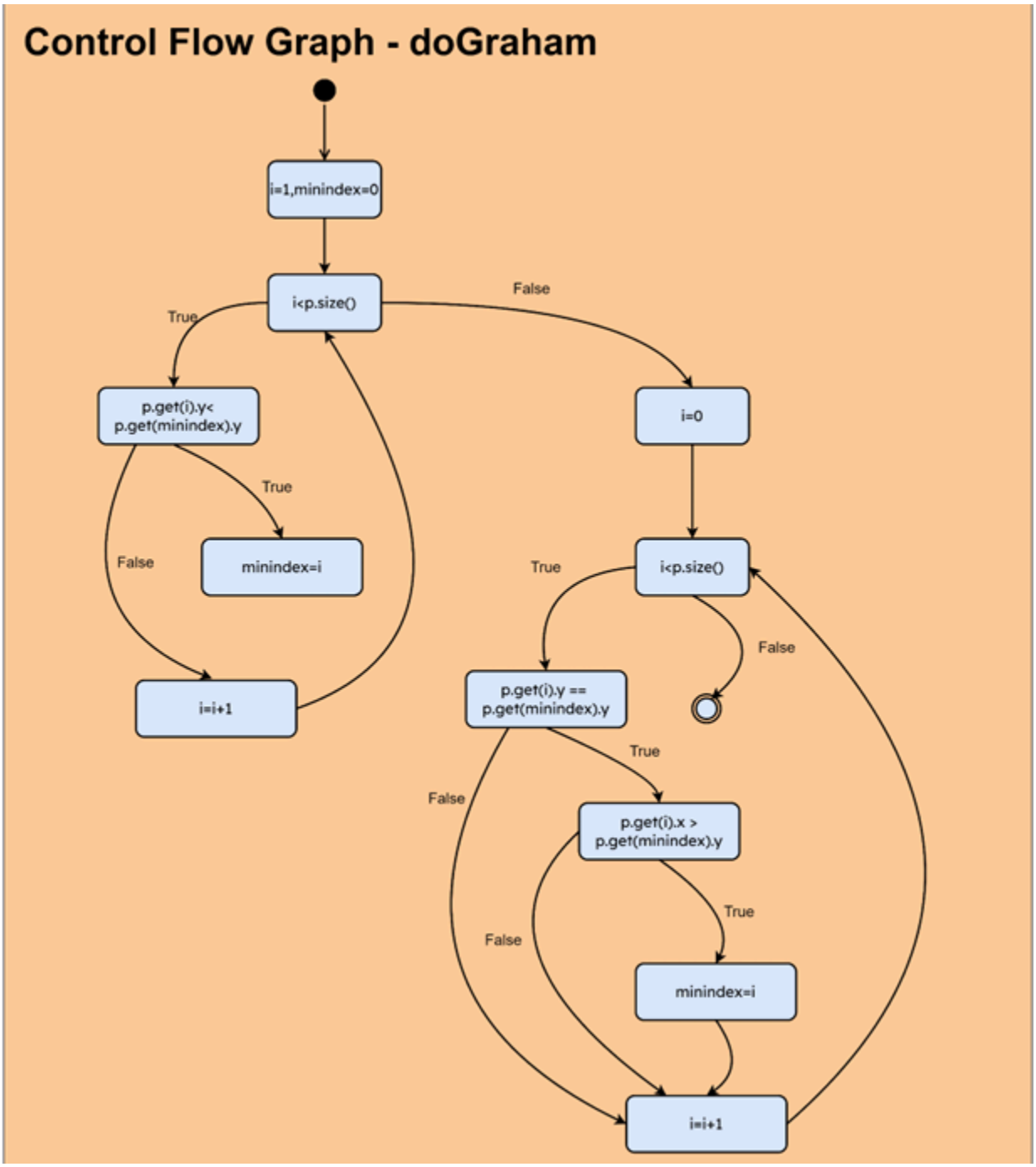
Kartavya Akabari - 202201213

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.

```
# 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
    ]

    # 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()
```

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.



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

Test Cases for Statement Coverage

- **Test Case 1:**
  - **Input Points:** `Point(2,3)`, `Point(1,2)`, `Point(3,1)`
  - **Objective:** This test case is designed to confirm that every statement in the function's control flow executes at least once, ensuring complete statement coverage.

Test Cases for Branch Coverage

- **Test Case 1:**
  - **Input Points:** `Point(2,3)`, `Point(1,2)`, `Point(3,1)`
  - **Objective:** This case is crafted to validate that both possible outcomes (True and False) of each conditional statement are exercised at least once.
- **Test Case 2:**
  - **Input Points:** `Point(3,3)`, `Point(4,3)`, `Point(5,3)`
  - **Objective:** This case ensures that conditions where both branches evaluate to False are also tested.

Test Cases for Basic Condition Coverage

- **Test Case 1:**
  - **Input Points:** `Point(2,3)`, `Point(1,2)`, `Point(3,1)`
  - **Objective:** This test checks that the condition `p[i].y < p[min_idx].y` is evaluated as True.
- **Test Case 2:**
  - **Input Points:** `Point(1,3)`, `Point(2,3)`, `Point(3,3)`
  - **Objective:** This case ensures that the condition `p[i].y < p[min_idx].y` is evaluated as False.
- **Test Case 3:**
  - **Input Points:** `Point(2,2)`, `Point(1,2)`, `Point(3,2)`

    - **Objective:** This case checks that both conditions `p[i].y == p[min_idx].y` and `p[i].x < p[min_idx].x` are evaluated as True.
- **Test Case 4:**
    - **Input Points:** `Point(3,2)`, `Point(4,2)`, `Point(2,2)`
    - **Objective:** This test validates that `p[i].y == p[min_idx].y` is True while `p[i].x < p[min_idx].x` is False.

Output :

```
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)
```

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

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

Mutated Code (Deletion Mutation):

```
min = i;
```

Statement Coverage Analysis:

- **Effect of Mutation:** The deletion mutation removes the conditional check, so `min = i` is executed for every iteration without

considering whether the current `y` value is smaller. This still achieves statement coverage, as the line is executed, but it may go undetected if test cases only focus on execution rather than validating that the smallest `y` value is correctly identified.

- **Potential Issue:** This mutation might not trigger a failure if tests do not verify that `min` truly points to the smallest `y` value, allowing erroneous results without detection.

b. Change Mutation

Original Code:

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

    min = i;

}
```

Mutated Code (Change Mutation):

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

    min = i;

}
```

Statement Coverage Analysis:

- **Effect of Mutation:** The condition changes to `<=`, allowing points with equal `y` values to also update `min`. This modification can affect the algorithm's behavior, especially when multiple points have identical `y` values.
- **Potential Issue:** With the updated condition, the algorithm could select points with equal `y` values as potential minimums, altering the

intended logic and potentially producing a different outcome if such cases are present.

c. Insertion Mutation

Original Code:

java

```
min = i;
```

Mutated Code (Insertion Mutation):

java

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

Basic Condition Coverage Analysis:

- **Effect of Mutation:**
  - **Altered Behavior:** This mutation modifies `min` to refer to the index immediately after `i`, rather than `i` itself, which results in an incorrect position being recorded.
  - **Risk of Out-of-Bounds Error:** If `i` is the last index, `i + 1` will exceed the list's bounds, potentially causing an out-of-range error or other unintended behavior.
  - **Impact on Algorithm Accuracy:** In algorithms for finding minimums, such incorrect indexing may lead to failing to identify the intended minimum, undermining the function's correctness.

- **Potential Undetected Outcome:** If tests focus only on the assignment of a value to `min` but do not check if the assigned index is correct, this error might not be detected, leading to unnoticed flaws in the algorithm's performance.

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: Loop Does Not Execute

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

Test Code:
java

```
Vector<Point> p = new Vector<>();
```

-
- **Expected Result:** Since `p` contains no elements (`p.size() == 0`), the loop will not run at all. The function should exit immediately without performing any operations. This scenario tests the condition where the loop is bypassed due to an empty input.

Test Case 2: Single Iteration of the Loop (No Processing)

- **Input:** A vector with a single point.

Test Code:
java

```
Vector<Point> p = new Vector<>();
p.add(new Point(0, 0));
```

- **Expected Result:** With only one element in the vector (`p.size() == 1`), the loop condition is technically checked but does not perform any iterations. The method should not make any changes, leaving the point in place. This case covers the situation where the loop condition is evaluated, but no loop iteration occurs due to having only one element.

Test Case 3: Loop Iterates Once with a Swap

- **Input:** A vector containing two points where the first point has a higher y-coordinate than the second.

Test Code:
java

```
Vector<Point> p = new Vector<>();
p.add(new Point(1, 1));
p.add(new Point(0, 0));
```

- **Expected Result:** The loop will run a single time, comparing the two points:
    - The first point `(1, 1)` has a higher y-coordinate than the second point `(0, 0)`, so a swap will occur.
    - The vector should end up as `[(0, 0), (1, 1)]`.
- **Purpose:** This case checks that the loop can execute once and perform a swap when the first point has a greater y-coordinate than the second.

Test Case 4: Multiple Iterations of the Loop (Swapping Occurs)

- **Input:** A vector with three points to test multiple loop iterations.

Test Code:
java

```
Vector<Point> 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 will iterate through all three points as follows:
    - **First iteration:** Compare the first point `(2, 2)` with `(1, 0)`. Since `(1, 0)` has a lower y-coordinate, `minY` is updated, and a swap occurs, placing `(1, 0)` at the start.
    - **Second iteration:** The updated first point `(1, 0)` is compared with `(0, 3)`. No swap is needed as `(1, 0)` has the lowest y-coordinate.
    - **Final Vector:** The resulting vector should be `[(1, 0), (2, 2), (0, 3)]`.

- **Purpose:** This case verifies that the loop iterates over multiple points, performs comparisons, and carries out swaps as needed.

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

Q3) and Q4) Same as Part I