
# IT314-SOFTWARE ENGINEERING

## Lab9- Mutation Testing

Name : Dhriti Goenka

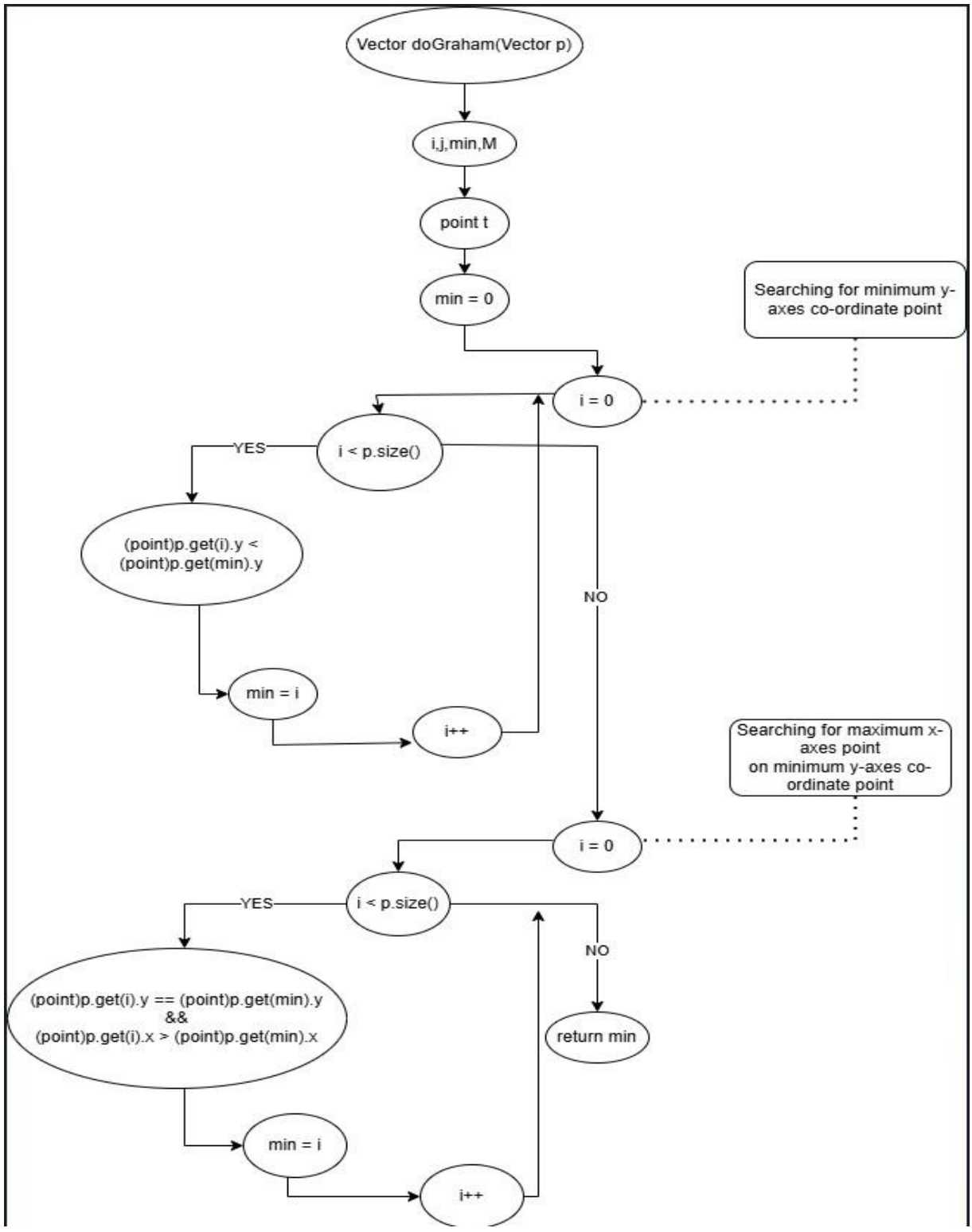
Student ID : 202201041

Lab Group :  1

**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:

a. Statement Coverage.

b. Branch Coverage.

c. Basic Condition Coverage.

### Test Cases for Statement Coverage

**Test Case 1:**

- **Input Points**:
  Point(2,3),Point(1,2),Point(3,1)
- **Purpose**:
  This test case is designed to ensure that all statements in the flow graph are executed at least once.

### Test Cases for Branch Coverage

**Test Case 1:**

- **Input Points**:
  Point(2,3),Point(1,2),Point(3,1)
- **Purpose**:
  This test case ensures that both branches (True/False) of the conditional statements are tested at least once.

**Test Case 2:**

- **Input Points**:
  Point(3,3),Point(4,3),Point(5,3)
- **Purpose**:
  This test case ensures that both branches (False/False) of the conditional statements are tested.

### Test Cases for Basic Condition Coverage

**Test Case 1:**

- **Input Points**:

Point(2,3),Point(1,2),Point(3,1)

**Purpose**:
This test case ensures 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)
- **Purpose**:
  This test 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)
- **Purpose**:
  This test case ensures 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)
- **Purpose**:
  This test case ensures that the condition `p[i].y == p[min_idx].y` is **True** and the condition `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:**

**Original Code:**

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

    min = i;

}
```

**Mutated Code (Deletion Mutation):**

```
min = i;
```

## Analysis for Statement Coverage

→ The deletion mutation removes the condition check, causing `min = i` to execute unconditionally. While **statement coverage** is still satisfied (since `min = i` is executed), the mutation may go **undetected** if the test cases do not verify the correctness of the minimum value selection, potentially leading to incorrect results without triggering a failure.

**b. Change Mutation:**

Original Code:

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

Mutated Code:

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

## Analysis for Branch Coverage

→ In the original condition, the comparison checks if the `y` value of `p.get(i)` is strictly less than the `y` value of `p.get(min)`. After mutating, the condition will now check if the `y` value of `p.get(i)` is less than or equal to the `y` value of `p.get(min)`.

This change could impact how the algorithm behaves, particularly if `y` values are equal for multiple points. The updated condition could now include cases where `y` values are equal, potentially resulting in a different behavior (such as including equal `y` values in the logic).

### c. Insertion Mutation:

#### Original Code

```
min = i;
```

### Mutated Code:

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

## Analysis for Basic Condition Coverage:

### → Effect of Mutation:

- **Altered Behavior**: The mutation changes the assignment of `min`, making it point to the index immediately after `i`, instead of `i` itself. This leads to an incorrect index being tracked.
- **Risk of Index Out-of-Bounds**: If `i` is the last index of the array or list, `i + 1` will exceed the valid index range, causing an out-of-bounds error or unexpected behavior.
- **Incorrect Algorithm Functionality**: In sorting or searching algorithms, this incorrect assignment of `min` may result in missing the correct element or failing to find the intended value, breaking the algorithm's correctness.

### Potential Undetected Outcome:

- **Failure to Detect the Issue**: If tests focus only on whether `min` is assigned a value and don't verify the correctness of the index, they may fail to catch the problem. Tests that don't check the correctness of the assignment may overlook this error.

## 3. 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 Explored Zero Times

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

**Test**:

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

- **Expected Result**:
  Since the vector has zero elements (`p.size() == 0`), the loop will not execute. The method should return immediately without any processing. This covers the scenario where the loop is not executed at all.

### Test Case 2: Loop Explored Once

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

**Test**:

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

p.add(new Point(0, 0));
```

- **Expected Result**:
  The vector has only one point (`p.size() == 1`), so the loop will be skipped. The method should perform no swapping. The point will remain in the same position, effectively leaving the vector unchanged. This test case covers the scenario where the loop iterates zero times but checks the condition for a single element.

### 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 loop will iterate once, comparing the first and second points:
  - The first point (`(1,1)`) has a higher `y`-coordinate than the second point (`(0,0)`), so the loop will perform a swap.
  - The vector will now be `[ (0, 0), (1, 1)]`. This test case checks the loop's behavior when it iterates once, performing a swap.

**Test Case 4: Loop Explored More Than Twice (Loop Iterates Over Multiple Points)**

- **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 will iterate through all three points:
  1. **First iteration**: The first point `(2, 2)` is compared with `(1, 0)`:
     - Since `(1, 0)` has a lower `y`-coordinate, the `minY` will be updated, and a swap will occur, moving `(1, 0)` to the front.
  2. **Second iteration**: The newly swapped first point `(1, 0)` will be compared with `(0, 3)`.
     - Since `(1, 0)` has a lower `y`-coordinate than `(0, 3)`, no further swap will be needed, as `minY` remains at index 1.
- The final vector will be `[ (1, 0), (2, 2), (0, 3)]`. This case ensures the loop iterates through multiple points and performs at least one swap.

# 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**