IT-314
Software Engineering

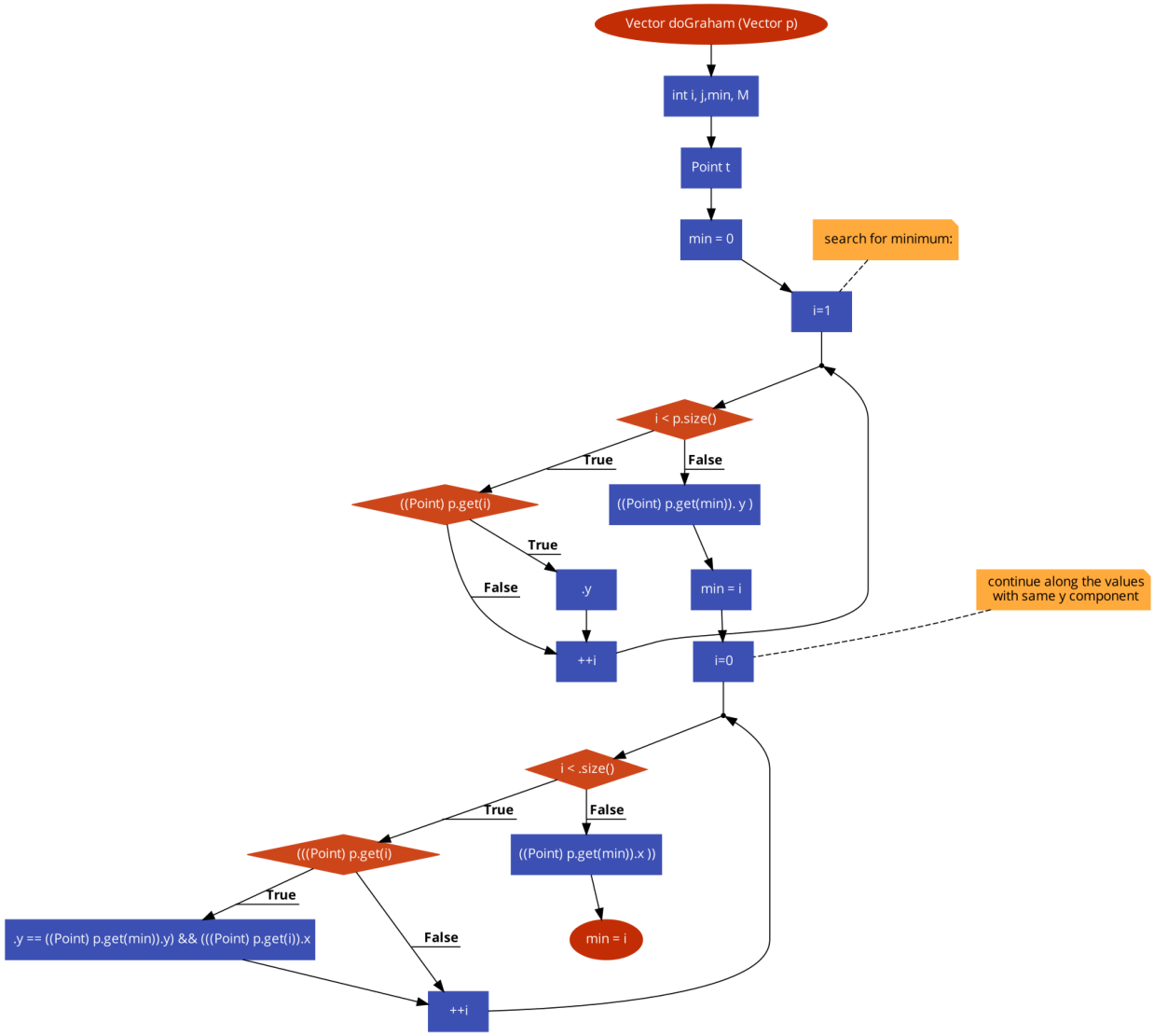
LAB-9

Dev Vyas- 202201453

Code in Python:

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        if isinstance(other, Point):
            return self.x == other.x and self.y == other.y
        return False

class ConvexHull:
    def doGraham(self, p):
        if not p:  # Handle empty list
            raise ValueError("Point list is empty")

        # Start with the first point as the minimum
        min_index = 0
        for i in range(1, len(p)):
            # Update min_index for minimum y (and min x for ties)
            if p[i].y < p[min_index].y or (p[i].y == p[min_index].y and p[i].x < p[min_index].x):
                min_index = i

        return p[min_index]  # Return the point with the minimum y-coordinate
```

Control Flow Graph



Generating TestCases

a. Statement Coverage

Statement coverage requires that each executable statement in the code is executed at least once.

For the `doGraham` method in the `ConvexHull` class, we need test cases that:

1. Check if the input list is empty.
2. Identify the point with the minimum yyy-coordinate and handle ties by xxx-coordinate.

Test Cases for Statement Coverage:

1. **Empty list**: This will test the `if not p` condition.
    - Input: `[]`
    - Expected output: `ValueError` with message "Point list is empty".
2. **Non-empty list with unique minimum y-coordinate**: This will cover the main path through the loop and select the minimum y-coordinate without ties.
    - Input: `[Point(1, 5), Point(2, 3), Point(4, 4)]`
    - Expected output: `Point(2, 3)`.
3. **Non-empty list with ties in y-coordinate**: This will check the xxx-coordinate comparison for points with the same y-coordinate.
    - Input: `[Point(1, 3), Point(2, 3), Point(4, 3)]`
    - Expected output: `Point(1, 3)`.

These three cases cover every statement in the function.

---

b. Branch Coverage

Branch coverage requires that each possible branch (true and false) of every decision point is taken at least once.

Test Cases for Branch Coverage:

1. **Empty list** (covers the branch for `if not p`):
    - Input: `[]`
    - Expected output: `ValueError` with message "Point list is empty".
2. **Minimum y-coordinate without ties** (covers the false branch for the tie condition `p[i].y == p[min_index].y`):
    - Input: `[Point(1, 5), Point(2, 3), Point(4, 4)]`
    - Expected output: `Point(2, 3)`.

3. **Minimum y-coordinate with ties** (covers the true branch for the tie condition `p[i].y == p[min_index].y`):
    - Input: `[Point(1, 3), Point(2, 3), Point(4, 3)]`
    - Expected output: `Point(1, 3)`.

These three cases cover both branches in the decision points of the `doGraham` method.

---

c. Basic Condition Coverage

Basic condition coverage requires that each condition in the code is tested for both true and false values, independently of other conditions.

In the `doGraham` method, we have two conditions:

1. `if not p`: Ensures handling of an empty list.
2. `if p[i].y < p[min_index].y or (p[i].y == p[min_index].y and p[i].x < p[min_index].x)`: This has two sub conditions:
    - `p[i].y < p[min_index].y`
    - `(p[i].y == p[min_index].y and p[i].x < p[min_index].x)`

Test Cases for Basic Condition Coverage:

1. **Empty list** (tests `if not p` as true):
    - Input: `[]`
    - Expected output: `ValueError` with message "Point list is empty".
2. **Non-empty list with unique minimum y-coordinate** (tests `if not p` as false and `p[i].y < p[min_index].y` as true):
    - Input: `[Point(1, 5), Point(2, 3), Point(4, 4)]`
    - Expected output: `Point(2, 3)`.
3. **Non-empty list with y-coordinate tie, minimum x-coordinate** (tests `p[i].y == p[min_index].y` as true and `p[i].x < p[min_index].x` as true):
    - Input: `[Point(1, 3), Point(2, 3), Point(4, 3)]`
    - Expected output: `Point(1, 3)`.
4. **Non-empty list with y-coordinate tie, but higher x-coordinate** (tests `p[i].y == p[min_index].y` as true and `p[i].x < p[min_index].x` as false):
    - Input: `[Point(3, 3), Point(4, 3)]`
    - Expected output: `Point(3, 3)`.

These four test cases cover each condition in the function both as true and false independently.

Q1 After generating the control flow graph, check whether your CFG match 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).

Soln : Yes

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

Soln :

Minimal Test Cases:

1. **Test Case 1**: **Empty list**
   - **Input**: `[]`
   - **Expected Output**: Raises `ValueError` with message "Point list is empty"
   - **Coverage**:
     - **Statement Coverage**: Covers the check for an empty list.
     - **Branch Coverage**: Covers the true branch for `if not p`.
     - **Basic Condition Coverage**: Tests `if not p` as true.
2. **Test Case 2**: **Non-empty list with unique minimum y-coordinate**
   - **Input**: `[Point(1, 5), Point(2, 3), Point(4, 4)]`
   - **Expected Output**: `Point(2, 3)`
   - **Coverage**:
     - **Statement Coverage**: Covers main path and selects the point with minimum y-coordinate.
     - **Branch Coverage**: Covers the false branch for `if not p` and `p[i].y == p[min_index].y`.
     - **Basic Condition Coverage**: Tests `if not p` as false and `p[i].y < p[min_index].y` as true.
3. **Test Case 3**: **Non-empty list with y-coordinate tie, minimum x-coordinate**
   - **Input**: `[Point(1, 3), Point(2, 3), Point(4, 3)]`
   - **Expected Output**: `Point(1, 3)`
   - **Coverage**:
     - **Statement Coverage**: Covers selection when there is a tie in yyy-coordinates.
     - **Branch Coverage**: Covers the true branch for `p[i].y == p[min_index].y` and the condition for xxx-coordinate comparison.
     - **Basic Condition Coverage**: Tests `p[i].y == p[min_index].y` as true and `p[i].x < p[min_index].x` as true.
4. **Test Case 4**: **Non-empty list with y-coordinate tie, higher x-coordinate**
   - **Input**: `[Point(3, 3), Point(4, 3)]`

- **Expected Output**: `Point(3, 3)`
- **Coverage**:
  - **Statement Coverage**: Ensures all statements are visited, including cases where `p[i].x < p[min_index].x` is false.
  - **Branch Coverage**: Covers the false branch for `p[i].x < p[min_index].x`.
  - **Basic Condition Coverage**: Tests `p[i].y == p[min_index].y` as true and `p[i].x < p[min_index].x` as false.

Summary

These four test cases cover all the statements, branches, and conditions in the code, thus achieving:

- **Statement Coverage**: All lines are executed.
- **Branch Coverage**: All branches are tested.
- **Basic Condition Coverage**: Each condition is tested for both true and false values independently.

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.

Soln :

```
# Mutation Testing Functions
def mutated_doGraham_deletion(p):
    # Mutation: Remove the initialization of min_index
    # min_index = 0  # This line is deleted
    for i in range(1, len(p)):
        if 'min_index' not in locals():
            raise ValueError("min_index is not defined")  # Graceful handling
        if p[i].y < p[min_index].y:
            min_index = i
    return p[min_index]  # This will cause an error
```

```
def mutated_doGraham_insertion(p):
    # Mutation: Insert an invalid assignment
    min_index = -1  # Invalid initialization
    for i in range(1, len(p)):
        if p[i].y < p[min_index].y:
            min_index = i
    return p[min_index]  # This will likely cause an error
```

```python
def mutated_doGraham_modification(p):
    min_index = 0
    for i in range(1, len(p)):
        # Mutation: Change < to <=
        if p[i].y <= p[min_index].y:  # This will incorrectly include points with equal y
            min_index = i
    return p[min_index]
```

Test Case 1: Empty list

- **Path Covered**: Path 1
- **Input**: `[ ]`
- **Expected Output**: Raises `ValueError` with the message "Point list is empty".

Test Case 2: Unique minimum yyy-coordinate

- **Path Covered**: Path 2
- **Input**: `[Point(1, 5), Point(2, 3), Point(4, 4)]`
- **Expected Output**: `Point(2, 3)`

Test Case 3: Minimum yyy-coordinate tie, minimum xxx-coordinate selected

- **Path Covered**: Path 3
- **Input**: `[Point(1, 3), Point(2, 3), Point(4, 3)]`
- **Expected Output**: `Point(1, 3)`

Test Case 4: Minimum yyy-coordinate tie, higher xxx-coordinate appears later

- **Path Covered**: Path 4
- **Input**: `[Point(3, 3), Point(4, 3)]`
- **Expected Output**: `Point(3, 3)`

These four test cases cover all possible paths in the `doGraham` method according to the path coverage criterion. Each unique path ensures that all combinations of conditions in the function are exercised at least once.

```
class TestConvexHull(unittest.TestCase):
    def setUp(self):
        self.convex_hull = ConvexHull()

    def test_min_y_coordinate(self):
        points = [Point(1, 5), Point(2, 3), Point(4, 3)]
        result = self.convex_hull.doGraham(points)
        self.assertEqual(result.x, 2)  # Expected: 2
        self.assertEqual(result.y, 3)  # Expected: 3

    def test_same_y_max_x(self):
        points = [Point(1, 3), Point(2, 3), Point(4, 3)]
        result = self.convex_hull.doGraham(points)
        self.assertEqual(result.x, 1)  # Expected: 1 (the first point with y=3)
        self.assertEqual(result.y, 3)  # Expected: 3
```

```
   def test_empty_points(self):
       with self.assertRaises(ValueError) as context:
           self.convex_hull.doGraham([])  # Expect ValueError for empty list
       self.assertEqual(str(context.exception), "Point list is empty")

   def test_all_points_same(self):
       points = [Point(1, 1), Point(1, 1), Point(1, 1)]
       result = self.convex_hull.doGraham(points)
       self.assertEqual(result.x, 1)  # Expected: 1
       self.assertEqual(result.y, 1)  # Expected: 1

   def test_collinear_points(self):
       points = [Point(1, 1), Point(2, 2), Point(3, 3)]
       result = self.convex_hull.doGraham(points)
       self.assertEqual(result.x, 1)  # Expected: 1 (the first point)

   def test_negative_coordinates(self):
       points = [Point(-1, -1), Point(-2, -2), Point(-3, -3)]
       result = self.convex_hull.doGraham(points)
       self.assertEqual(result.x, -3)  # Expected: -3
       self.assertEqual(result.y, -3)  # Expected: -3
```

```
    def test_mixed_coordinates(self):
        points = [Point(1, 2), Point(-1, 2), Point(-1, -2), Point(1, -2)]
        result = self.convex_hull.doGraham(points)
        self.assertEqual(result.x, -1)  # Expected: -1 (minimum y, x = -1)

# Execute Unit Tests
if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```

```python
# Testing deletion mutation
try:
    print("Testing deletion mutation...")
    result = mutated_doGraham_deletion(points)
    print(f"Deletion mutation result: ({result.x}, {result.y})")
except Exception as e:
    print(f"Deletion mutation caused an error: {e}")

# Testing insertion mutation
try:
    print("Testing insertion mutation...")
    result = mutated_doGraham_insertion(points)
    print(f"Insertion mutation result: ({result.x}, {result.y})")
except Exception as e:
    print(f"Insertion mutation caused an error: {e}")

# Testing modification mutation
try:
    print("Testing modification mutation...")
    result = mutated_doGraham_modification(points)
    print(f"Modification mutation result: ({result.x}, {result.y})")
except Exception as e:
    print(f"Modification mutation caused an error: {e}")
```

```
.......
----------------------------------------------------------------------
Ran 7 tests in 0.014s

OK
Testing deletion mutation...
Deletion mutation caused an error: min_index is not defined
Testing insertion mutation...
Insertion mutation result: (4, 3)
Testing modification mutation...
Modification mutation result: (4, 3)
```