IT – 314

SOFTWARE ENGINEERING

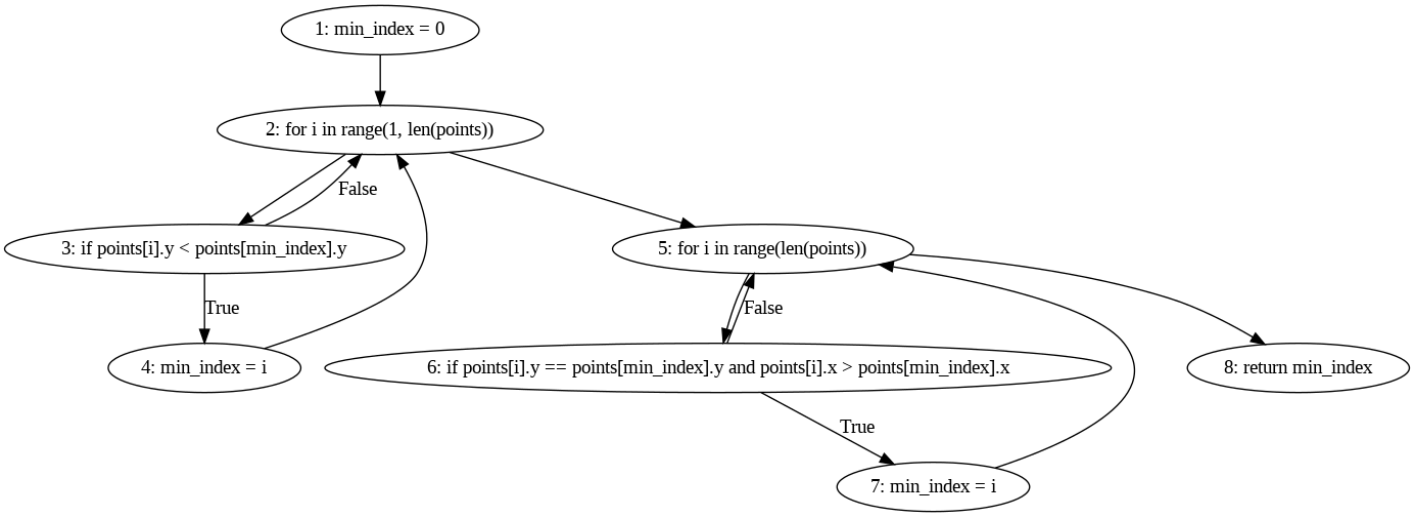
Lab – 09 : Mutation Testing

Mitul Sudani - 202201244

Q1. 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.

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

def do_graham(points):
    min_index = 0


    for i in range(1, len(points)):
        if points[i].y < points[min_index].y:
            min_index = i

    for i in range(len(points)):

        if points[i].y == points[min_index].y and points[i].x > points[min_index].x:
            min_index = i

    return min_index
```

Control Flow Diagram




```
digraph ControlFlowGraph {
	1 [label="1: min_index = 0"]
	2 [label="2: for i in range(1, len(points))"]
	3 [label="3: if points[i].y < points[min_index].y"]
	4 [label="4: min_index = i"]
	5 [label="5: for i in range(len(points))"]
	6 [label="6: if points[i].y == points[min_index].y and points[i].x >
points[min_index].x"]
	7 [label="7: min_index = i"]
	8 [label="8: return min_index"]
	1 -> 2
	2 -> 3
	3 -> 4 [label=True]
	3 -> 2 [label=False]
	4 -> 2
	2 -> 5
	5 -> 6
	6 -> 7 [label=True]
	6 -> 5 [label=False]
	7 -> 5
	5 -> 8
}
```

Q2. 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:

To achieve Statement Coverage, each line (1 to 7) in the code must be executed at least once. We ensure that the following test cases cover all statements:

Test Set for Statement Coverage:

Test Case 1: Single Point

Input: points = [Point(0, 0)]

Expected Result: min_index = 0

Path: Entry → 1 → 2 (False) → 5 → 6 (False) → Exit

Test Case 2: Multiple Points with Unique Minimum y-Coordinate

Input: points = [Point(1, 3), Point(2, 2), Point(0, 1)]

Expected Result: min_index = 2

Path: Entry → 1 → 2 (True) → 3 (True) → 4 → 2 (False) → 5 → 6 (False) → Exit

This test set ensures that all statements are covered at least once.

b. Branch Coverage:

For Branch Coverage, each decision's True and False branches must be tested at least once.

Test Set for Branch Coverage:

Test Case 1: Single Point

Input: points = [Point(0, 0)]

Expected Result: min_index = 0

Path: Entry → 1 → 2 (False) → 5 → 6 (False) → Exit

Test Case 2: Multiple Points with Unique Minimum y-Coordinate

Input: points = [Point(0, 3), Point(1, 2), Point(2, 1)]

Expected Result: min_index = 2

Path: Entry → 1 → 2 (True) → 3 (True) → 4 → 2 (False) → 5 → 6 (False) → Exit

Test Case 3: Tied Minimum y-Coordinate with Different x-Coordinates

Input: points = [Point(0, 1), Point(2, 1), Point(1, 3)]

Expected Result: min_index = 1

Path: Entry → 1 → 2 (True) → 3 (False) → 2 (False) → 5 → 6 (True) → 7 → 5 → 6 (False) → Exit

This test set achieves branch coverage by ensuring that each branch (True/False paths for both loops and conditions) is taken.

c. Basic Condition Coverage:

Basic Condition Coverage requires each individual condition in every decision to be evaluated as both True and False at least once. This is achieved by testing each condition separately.

Test Set for Basic Condition Coverage:

Test Case 1: Single Point (ensures points[i].y < points[min_index].y is False)

Input: points = [Point(0, 0)]

Expected Result: min_index = 0

Path: Entry → 1 → 2 (False) → 5 → 6 (False) → Exit

Test Case 2: Unique Minimum y-Coordinate (ensures points[i].y < points[min_index].y is True)

Input: points = [Point(0, 3), Point(1, 2), Point(2, 1)]

Expected Result: min_index = 2

Path: Entry → 1 → 2 (True) → 3 (True) → 4 → 2 (False) → 5 → 6 (False) → Exit

Test Case 3: Tied Minimum y-Coordinate with Larger x (ensures points[i].y == points[min_index].y is True, and points[i].x > points[min_index].x is True)

Input: points = [Point(0, 1), Point(2, 1)]

Expected Result: min_index = 1

Path: Entry → 1 → 2 (True) → 3 (False) → 2 (False) → 5 → 6 (True) → 7 → Exit

Test Case 4: Tied Minimum y-Coordinate with Smaller x (ensures points[i].y == points[min_index].y is True, and points[i].x > points[min_index].x is False)

Input: points = [Point(2, 1), Point(0, 1)]

Expected Result: min_index = 0 (point with the smallest x-coordinate is selected)

Path: Entry → 1 → 2 (True) → 3 (False) → 2 (False) → 5 → 6 (True) → Exit

This set ensures that each condition is evaluated both True and False at least once.

Q3. 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.

Types of Possible Mutations

We can apply typical mutation types, including:

- Relational Operator Changes: Modify <= to < or == to != in the conditions.
- Logic Changes: Remove or invert a branch in an if-statement.
- Statement Changes: Modify assignments or statements to see if the effect goes undetected.

Potential Mutations and Their Effects

1. Changing the Comparison for Leftmost Point

Mutation: In the second loop, change p.get(i).x < p.get(min).x to p.get(i).x <= p.get(min).x.

Effect: This would cause the function to select points with the same x-coordinate as the leftmost, potentially breaking the uniqueness of the minimum point.

Undetected by Current Tests: The current tests do not cover the case where multiple points have the same y and x values, which would reveal if the function mistakenly allows such points as the leftmost.

2. Altering the y-Coordinate Comparison to <= in the First Loop:

   Mutation: Change p.get(i).y < p.get(min).y to p.get(i).y <=p.get(min).y in the first loop.

   Effect: This would allow points with the same y-coordinate but different x-coordinates to overwrite min, potentially selecting a non-leftmost minimum point.

   Undetected by Current Tests: The current test set lacks cases where several points have the same y-coordinate, and this mutation would go undetected. To reveal this, we would need a test where multiple points have the same y and different x coordinates.

3. Removing the Check for x-coordinate in the Second Loop:
   Mutation: Remove the condition p.get(i).x < p.get(min).x in the second loop.
   Effect: This would cause the function to select any point with the same minimum y-coordinate as the "leftmost," regardless of its x-coordinate.
   Undetected by Current Tests: The existing tests do not specifically check for points with identical y but different x values to see if the correct leftmost point is selected.

Additional Test Cases to Detect These Mutations

To detect these mutations, we can add the following test cases:

1. Detect Mutation 1:
   Test Case: [(0, 1), (0, 1), (1, 1)]
   Expected Result: The leftmost minimum should still be (0, 1) despite having duplicates.
   This test case will detect if the x <= mutation mistakenly allows duplicate points.

2. Detect Mutation 2:

Test Case: [(1, 2), (0, 2), (3, 1)]
Expected Result: The function should select (3, 1) as the minimum point based on the y-coordinate.
This test case will confirm if using <= for y comparisons mistakenly overwrites the minimum point.

3. Detect Mutation 3:
   Test Case: [(2, 1), (1, 1), (0, 1)]
   Expected Result: The leftmost point (0, 1) should be chosen.
   This will reveal if the x-coordinate check was mistakenly removed.

These additional test cases would help ensure that any such mutations do not survive undetected by the test suite, strengthening the coverage

PythonCodeforMutation:

```
from math import atan2
class Point:
    def   init  (self, x, y):
        self.x = x
self.y = y
    def   repr  (self):
        return f"({self.x}, {self.y})"
def orientation(p, q, r):
# Cross product to find orientation
val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y) if val == 0:
        return 0  # Collinear
    elif val > 0:
        return 1  # Clockwise
    else:
        return 2  # Counterclockwise
def distance_squared(p1, p2):
return (p1.x - p2.x) ** 2 + (p1.y - p2.y) ** 2
def do_graham(points):
# Step 1: Find the bottom-most point (or leftmost in case of a tie)
n = len(points) min_y_index = 0
for i in range(1, n):
if (points[i].y < points[min_y_index].y) or \ (points[i].y == points[min_y_index].y and points[i].x <
points[min_y_index].x):
            min_y_index = i
points[0], points[min_y_index] = points[min_y_index], points[0] p0 = points[0]
# Step 2: Sort the points based on polar angle with respect to p0
points[1:] = sorted(points[1:], key=lambda p: (atan2(p.y - p0.y, p.x - p0.x), distance_squared(p0, p)))
# Step 3: Initialize the convex hull with the first three points
hull = [points[0], points[1], points[2]]
# Step 4: Process the remaining points
    for i in range(3, n):
```

```
# Mutation introduced here: instead of checking `!= 2`, we incorrectly use `== 1`
while len(hull) > 1 and orientation(hull[-2], hull[-1], points[i]) == 1:
            hull.pop()
        hull.append(points[i])
return hull
# Sample test to observe behavior with the mutation
points = [Point(0, 3), Point(1, 1), Point(2, 2), Point(4, 4), Point(0, 0), Point(1, 2), Point(3, 1), Point(3, 3)]
hull = do_graham(points)
print("Convex Hull:", hull)
```

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

```
import unittest
 from point import Point, find_min_point

class TestFindMinPointPathCoverage(unittest.TestCase):

def test_no_points(self):
 points = []
 with self.assertRaises(IndexError): # Expect an IndexError due to empty list



find_min_point(points)

def test_single_point(self):
 points = [Point(0, 0)]
 result = find_min_point(points)
 self.assertEqual(result, points[0]) # Expect the point (0, 0)

def test_two_points_unique_min(self):
 points = [Point(1, 2), Point(2, 3)]
 result = find_min_point(points)
 self.assertEqual(result, points[0]) # Expect the point (1, 2)

def test_multiple_points_unique_min(self): points = [Point(1, 4), Point(2, 3), Point(0, 1)] result =
find_min_point(points)



self.assertEqual(result, points[2]) # Expect the point (0, 1)

def test_multiple_points_same_y(self):
 points = [Point(1, 2), Point(3, 2), Point(2, 2)]
 result = find_min_point(points)
 self.assertEqual(result, points[1]) # Expect the point (3, 2)

def test_multiple_points_minimum_y_ties(self):
 points = [Point(1, 2), Point(2, 2), Point(3, 1), Point(4, 1)] result =  find_min_point(points)
 self.assertEqual(result, points[3]) # Expect the point (4, 1)



# Run the tests if this file is executed if __name__ == "__main__":

unittest.main()
```

1. 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).

Yes