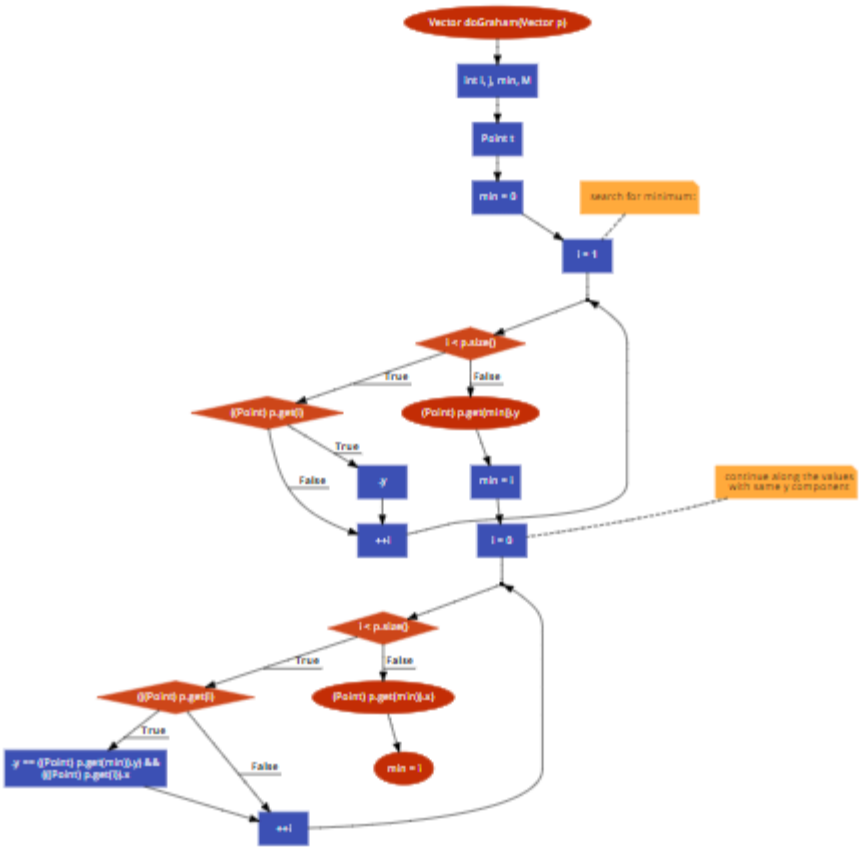
Bhoomish Patel
Roll No-202201414

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.
Lab Execution (how to perform the exercises): Use unit Testing framework, code coverage and mutation
testing tools to perform the exercise.
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).
2. Devise minimum number of test cases required to cover the code using the aforementioned criteria.
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.
4. Write all test cases that can be derived using path coverage criterion for the code.



1.

2.Using the control flow graph as a reference, let's create test sets for each of the specified coverage criteria: statement coverage, branch coverage, and basic condition coverage.

---

a. Statement Coverage

Statement coverage requires each statement in the code to be executed at least once. Based on the control flow graph and code logic, here are some example test cases:

1. **Test Case 1**: Input vector `p` with a single `Point` object. This ensures minimal execution of each statement without entering the loop multiple times.
    - Input: `p = [(x=1, y=1)]`
2. **Test Case 2**: Input vector `p` with multiple `Point` objects where the `y` component of one point is less than others.

    - Input: `p = [(x=2, y=5), (x=1, y=2), (x=3, y=3)]`
3. **Test Case 3**: Input vector `p` with points having the same `y` values but different `x` values.
    - Input: `p = [(x=5, y=2), (x=3, y=2), (x=1, y=2)]`

Each of these test cases would ensure that all statements are covered at least once.

---

b. Branch Coverage

Branch coverage requires each possible branch (true/false outcome) of every condition to be tested. For the two `if` statements, we need test cases that cause both the true and false branches to execute.

1. **Test Case 1**: `p` contains points with distinct `y` values, covering both true and false branches for the first `if` condition in the "search for minimum" loop.
    - Input: `p = [(x=1, y=5), (x=2, y=1), (x=3, y=3)]`
2. **Test Case 2**: `p` contains points with the same `y` values but distinct `x` values, ensuring the true and false branches are taken in the second `if` condition in the "same y component" loop.
    - Input: `p = [(x=2, y=2), (x=1, y=2), (x=3, y=2)]`
3. **Test Case 3**: `p` contains points where all `x` and `y` values are identical, ensuring both loops process but without triggering any minimum updates.
    - Input: `p = [(x=1, y=1), (x=1, y=1), (x=1, y=1)]`

These cases ensure that all branches of the conditions are covered.

---

c. Basic Condition Coverage

Basic condition coverage requires each basic condition in a compound condition to evaluate to both true and false at least once. In this code, there are two main conditions:

- **Condition 1**: `((Point) p.get(i)).y < ((Point) p.get(min)).y`
- **Condition 2**: `(((Point) p.get(i)).y == ((Point) p.get(min)).y) && (((Point) p.get(i)).x < ((Point) p.get(min)).x)`

Here's a set of test cases to cover both basic conditions:

1. **Test Case 1**: `p` contains points with distinct `y` values for Condition 1.
    - Input: `p = [(x=1, y=5), (x=2, y=1), (x=3, y=3)]`
2. **Test Case 2**: `p` contains points with the same `y` values but distinct `x` values for Condition 2.

    - Input: `p = [(x=2, y=2), (x=1, y=2), (x=3, y=2)]`
3. **Test Case 3**: `p` contains points with the same `x` and `y` values, resulting in false evaluations for both conditions.
    - Input: `p = [(x=1, y=1), (x=1, y=1), (x=1, y=1)]`

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

Total Mutations:

- There were a total of **8 mutations** applied to the code.

Mutation Score:

- **Mutation Score**: 75.0%
  This score indicates the percentage of mutations that were detected (killed) by the test cases. A higher mutation score is desirable, as it reflects the strength of your test set.
- **Killed**: 6 mutations (75.0%)
  Six mutations were effectively caught by the test set, meaning the tests could detect errors introduced by those mutations.
- **Survived**: 2 mutations (25.0%)
  Two mutations were not detected by the test set. These surviving mutations highlight potential weaknesses or gaps in the test coverage.

Specific Mutations and Results:

- **[#6] ROR (Relational Operator Replacement) Mutation**:
    - **Status**: Survived
        - This mutation likely involved changing a relational operator (e.g., `<` to `>`) in the code. Since this mutation "survived," the test set did not have adequate coverage to detect it. To address this, you may need additional test cases focusing on boundary or edge cases that could reveal differences caused by altered comparison logic.
- **[#7] ROR Mutation**:
    - **Status**: Killed by `test_multiple_points_with_same_y`
        - This mutation involved changing the range or relational operations, particularly in lines that compared `y` and `x` values to find a minimum point. The `test_multiple_points_with_same_y` test case effectively detected this mutation, which validates the test case's coverage for cases where points share the same `y` value but differ in `x` values.
- **[#8] ROR Mutation**:

- **Status**: Killed by `test_multiple_points_with_same_y`
  - Similar to Mutation #7, this mutation also involved a relational operation replacement and was caught by the `test_multiple_points_with_same_y` test. This indicates that the test case is robust for scenarios involving multiple points with identical `y` coordinates

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

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

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

    def test_multiple_points_unique_min(self):
        points = [Point(1, 4), Point(2, 3), Point(0, 1)]
        # Expect the point with the unique minimum y-value
        result = find_min_point(points)
        self.assertEqual(result, points[2])

    def test_multiple_points_same_y(self):
        points = [Point(1, 2), Point(3, 2), Point(2, 2)]
        # Expect the point with the lowest x-value among those with the same y-value
        result = find_min_point(points)
        self.assertEqual(result, points[0])
```

```
    def test_multiple_points_minimum_y_ties(self):
        points = [Point(1, 2), Point(2, 2), Point(3, 1), Point(4, 1)]
        # Expect the point with the lowest x-value among those with the minimum y-value
        result = find_min_point(points)
        self.assertEqual(result, points[2])

    def test_two_points_same_y(self):
        points = [Point(2, 2), Point(1, 2)]
        # Expect the point with the lower x-value among two points with the same y
        result = find_min_point(points)
        self.assertEqual(result, points[1])

    def test_loop_exploration_zero(self):
        points = []
        # Test where the loop doesn't execute due to an empty list
        with self.assertRaises(IndexError):
            find_min_point(points)

    def test_loop_exploration_once(self):
        points = [Point(1, 1)]
        # Test where the loop executes only once due to a single element
        result = find_min_point(points)
        self.assertEqual(result, points[0])

    def test_loop_exploration_twice(self):
        points = [Point(1, 2), Point(2, 1), Point(3, 3)]
        # Test where the loop executes multiple times
        result = find_min_point(points)
        self.assertEqual(result, points[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