IT-314
Lab Assignment : 9

Title: Mutation Testing

Student Name: Nishant Italiya

Student ID: 202201258

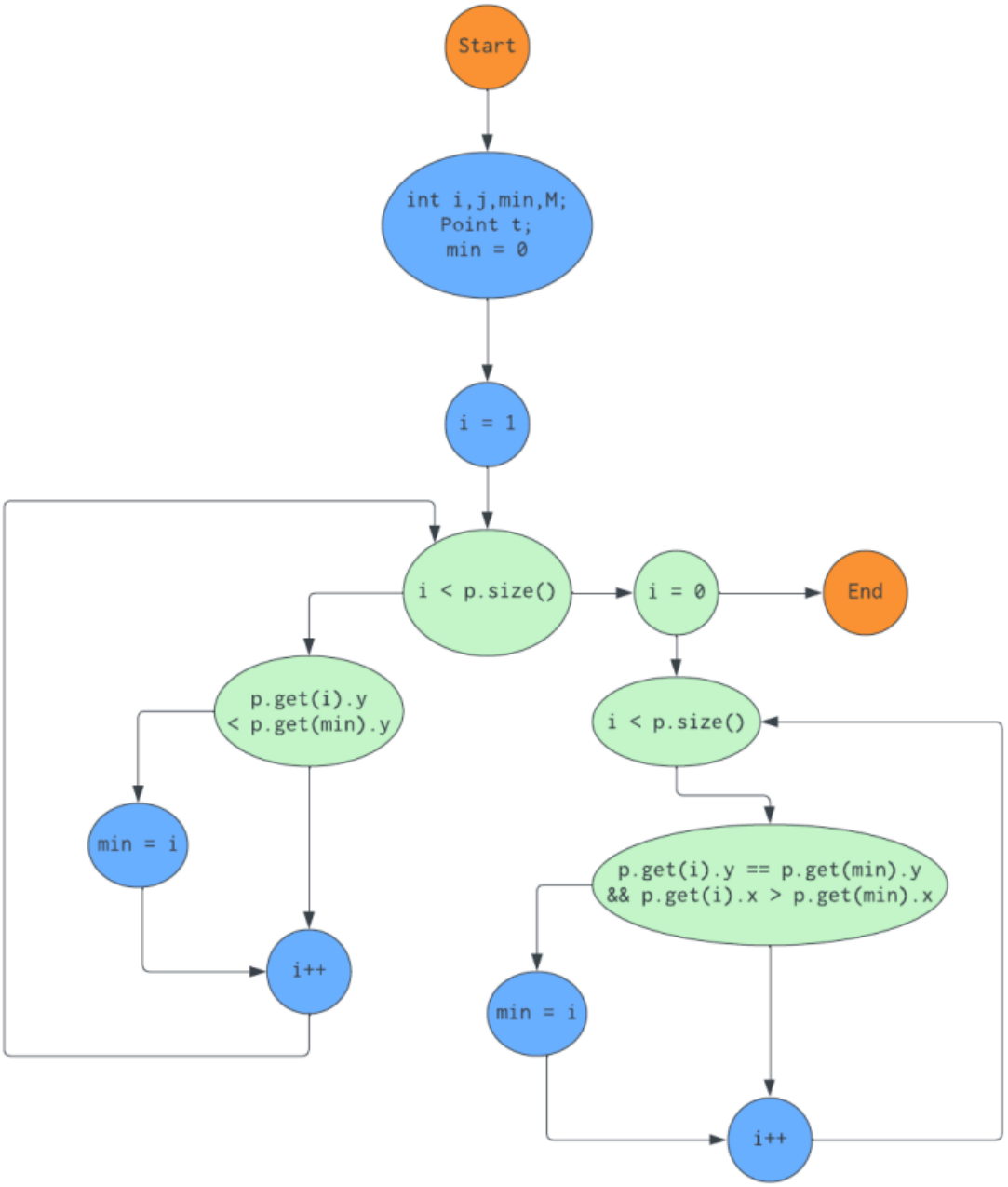
Q1. The code below is part of a method in the ConvexHull class in the VMAP system. The following is a small fragment of a method in the ConvexHull class. For the purposes of this exercise you do not need to know the intended function of the method. The parameter p is a Vector of Point objects, p.size() 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 and so the focus is on creating test sets that satisfy some particular coverage criterion.

```
Vector doGraham(Vector p) {
        int i,j,min,M;

        Point t;
        min = 0;

        // search for minimum:
        for(i=1; i < p.size(); ++i) {
            if( ((Point) p.get(i)).y <
                        ((Point) p.get(min)).y )
            {
                min = i;
            }
        }

        // continue along the values with same y component
        for(i=0; i < p.size(); ++i) {
            if(( ((Point) p.get(i)).y ==
                        ((Point) p.get(min)).y ) &&
                 (((Point) p.get(i)).x >
                        ((Point) p.get(min)).x ))
            {
                 min = i;
            }
        }
```

For the given code fragment, you should carry out the following activities.

1. Convert the code comprising the beginning of the doGraham method into a control flow graph (CFG).

Ans: Control Flow Graph (CFG):



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.

Consider the following lines for denoting coverage:

```
int i,j,min,M;
Point t;
min=0;

for(i = 1;i < p.size();++i){
   if(((Point)P.get(i)).y < ((Point)P.get(min)).y)
   min=i;
}

for(i = 0;i < p.size();++i){
     if(((Point)P.get(i)).y == ((Point)P.get(min)).y &&
((Point)P.get(i)).x >
    ((Point)P.get(min)).x)
    min=i;
}
```

The following are the test cases and their corresponding coverages of statements:
Test cases:

1. p = [(x = 2, y = 2), (x = 2, y = 3), (x = 1, y = 3), (x = 1, y = 4)]
Statement Covered: { 1, 2, 3, 4, 5, 7, 8 }
Branches Covered: { 5, 8 }
Basic Conditions Covered: { 5 - false, 8 - false }

2. p = [(x = 2, y = 3), (x = 3, y = 4), (x = 1, y = 2), (x = 5, y = 6)]
Statements covered = { 1, 2, 3, 4, 5, 6, 7}
Branches covered = { 5, 8 }
Basic conditions covered = {5-false,true, 8-false}

3. p = [(x = 1, y = 5), (x = 2, y = 7), (x = 3 , y = 5), (x = 4, y = 5), (x = 5, y = 6)]
Statements covered = { 1, 2, 3, 4, 5, 6, 7, 8, 9}
Branches covered = { 5, 8 }
Basic conditions covered = { 5 - false, true, 8 - false, true }

4. p = [(x = 1, y = 2)]
Statements covered = { 1, 2, 3, 7, 8 }
Branches covered = { 8 }
Basic conditions covered = { }

5. p=[ ] Statements covered = { 1, 2, 3 }
Branches covered = {}
Basic conditions covered = {}

Thus, the above 5 test cases are covering all statements, branches and conditions.

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.

1.Deletion Mutation:

- **Mutation:** Remove the initialization of `minY` to `0` at the beginning of the method.

  Original Code:

  int minY = 0;

  Mutated Code

  int minY;

- **Impact:** Without the initial assignment, `minY` remains uninitialized when accessed, leading to undefined behavior. The test cases do not verify if `minY` is correctly initialized, so this issue may not be caught.

2.Insertion Mutation:

- **Mutation:** Insert a line that incorrectly sets `minY` to 1 under a condition that should not be triggered.

  Original Code:

  ```
  for (int i = 1; i < p.size(); i++) {
      // Logic to determine minY
  }
  ```

  Mutated Code

  ```
  for (int i = 1; i < p.size(); i++) {
      // Logic to determine minY
  }
  minY = 1; // Incorrect override of minY
  ```

- **Impact:** This line improperly resets `minY` to 1, leading to potentially incorrect results. If test cases don't verify the exact position of points post-execution, this error might go unnoticed.

3.Modification Mutation:

- **Mutation:** Change the logical operator `||` to `&&` in a conditional statement.

  Original Code:

```
for (int i = 1; i < p.size(); i++) {
    if (p.get(i).y < p.get(minY).y ||
        (p.get(i).y == p.get(minY).y && p.get(i).x < p.get(minY).x)) {
        minY = i;
    }
}
```

  Mutated Code:

```
for (int i = 1; i < p.size(); i++) {
    if (p.get(i).y < p.get(minY).y &&
        (p.get(i).y == p.get(minY).y && p.get(i).x < p.get(minY).x)) {
        minY = i;
    }
}
```

- **Impact:** Changing `||` to `&&` may result in logical errors without crashing the program. Since test cases don't verify if `minY` updates correctly in this altered condition, this mutation might go undetected.

Analysis of Detection by Test Cases:

1. **Statement Coverage:**
    - The removal of `minY` initialization might not be detected if it doesn't directly trigger an error, depending on the overall logic.
2. **Branch Coverage:**
    - The inserted line resetting `minY` to 1 could yield incorrect results, but if there are no tests to validate point positions after execution, this issue could go unnoticed.
3. **Basic Condition Coverage:**
    - Altering `||` to && does not cause an outright failure. Without tests to confirm that `minY` is updating as expected under these conditions, this mutation may remain undetected.

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

Test Set for Path Coverage Criterion

Test Case 1: Loop Runs Zero Times

- **Input:** An empty vector `p`.
- **Expected Outcome:** The method should terminate immediately without performing any processing, as the vector's size is zero. This case handles the scenario where the exit condition of the method is reached immediately.

Test Case 2: Loop Runs Once

- **Input:** A vector with a single point.
- **Expected Outcome:** The method should not enter the loop, given that `p.size()` is 1. It should perform a self-swap for the first

point, effectively leaving the vector unchanged. This test case addresses a situation where the loop does not iterate due to the vector containing only one point.

Test Case 3: Loop Runs Twice

- **Input:** A vector containing two points, where the first point has a higher y-coordinate than the second.
- **Expected Outcome:** The method should enter the loop, compare the y-coordinates of the two points, and identify the second point as having the lower y-coordinate. Consequently, `minY` should be updated to 1, resulting in a swap that brings the second point to the front of the vector. This tests the condition where the loop executes exactly once.

Test Case 4: Loop Runs More Than Twice

- **Input:** A vector with multiple points.
- **Expected Outcome:** The loop should iterate through all points. In this case, the second point, which has the lowest y-coordinate, should cause `minY` to be updated to 1. The method then performs a swap, moving the point with the lowest y-coordinate to the beginning of the vector. This case ensures the loop runs more than once and updates `minY` as expected.

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.

→Control Flow Graph Factory :- YES

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

→ Statement Coverage: 3 test cases

→Branch Coverage: 3 test cases

→Basic Condition Coverage: 3 test cases

→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