Software Engineering

IT-314

Akshat Joshi - 202201185

LAB - 09

Dhirubhai Ambani Institute Of Information And Communication Technology

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.

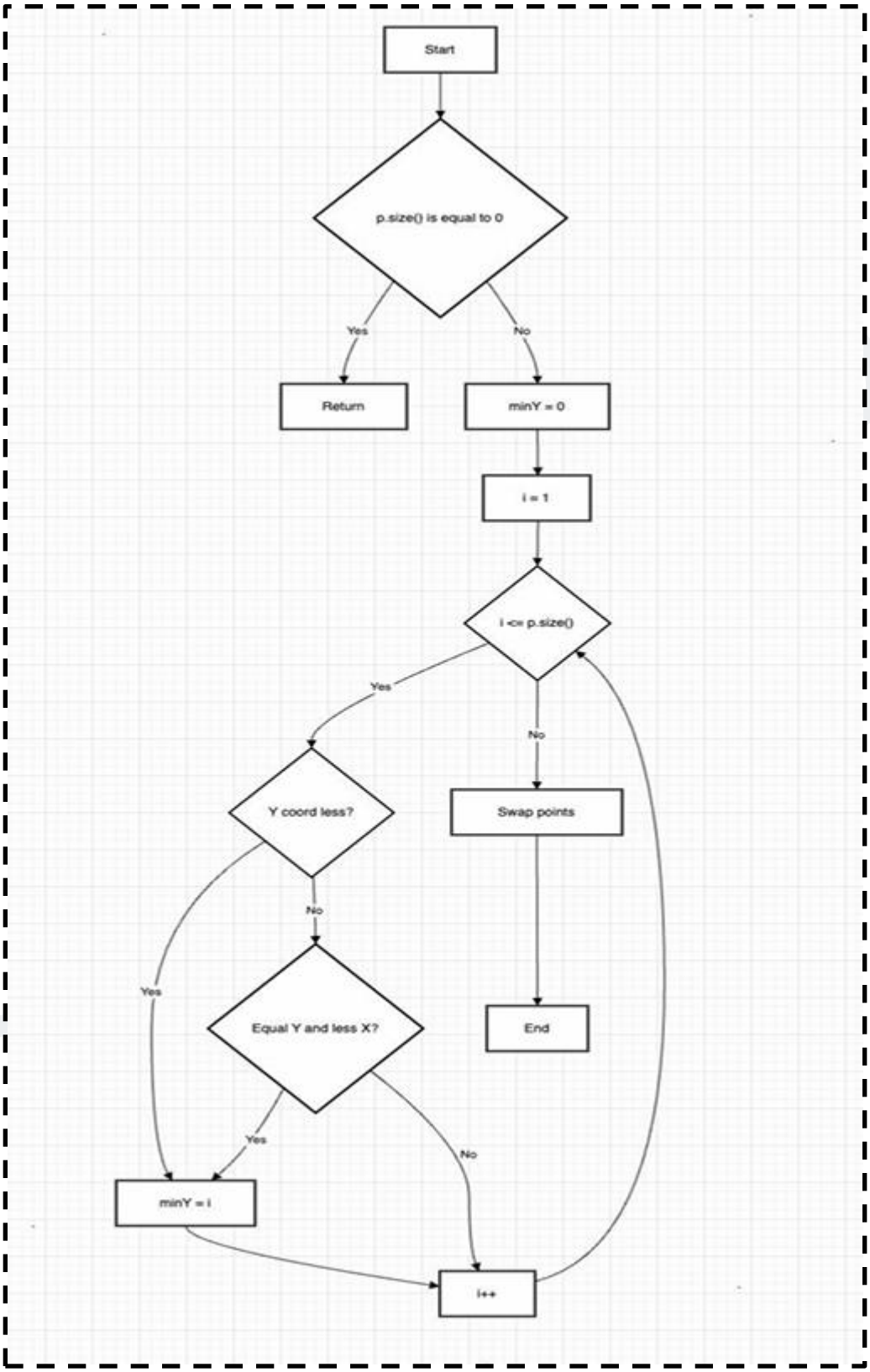
```
Vector doGraham(Vector p) {
        int i,j,min,M;

        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.

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.

```
public
class Point
{
   double x;
   double y;
public
   Point(double x, double y)
   {
      this.x = x;
      this.y = y;
   }
} public class ConvexHull
{
public
   void doGraham(Vector<Point> p)
   {
      if (p.size() == 0)
      {
         return;
      }
      int minY = 0;
      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;
         }
      }
      Point temp = p.get(0);
      p.set(0, p.get(minY));
      p.set(minY, temp);
   }
}
```

Below is the Control Flow graph of above code:


Start
p.size() is equal to 0
Yes
No
Return
minY = 0
i = 1
i <= p.size()
Yes
No
Y coord less?
Swap points
No
Yes
Equal Y and less X?
End
Yes
No
minY = i
i++

After generating the control flow graph, compare it with the one generated by the Control Flow Graph Factory Tool and the Eclipse flow graph generator.

Answer:

- **Control Flow Graph Factory**: Yes, the graphs match.
- **Eclipse Flow Graph Generator**: Yes, the graphs match.

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.

1) Statement Coverage.

➔ Statement coverage requires that every line of code is executed at least one time. Test Scenarios:

- **Test Case 1:** The input vector `p` is empty (`p.size() == 0`).
  Expected Outcome: The function should terminate early without executing any further logic.
- **Test Case 2:** The input vector `p` contains at least one `Point` object.
  Expected Outcome: The function should proceed to identify the `Point` with the smallest `y`-coordinate.

2) Branch Coverage.

➔ Branch coverage ensures that every conditional statement (branch) in the code is evaluated as both true and false at least once during testing. Test Scenarios:

- **Test Case 1:** The vector `p` is empty (`p.size() == 0`).
  Expected Outcome: The condition `if (p.size() == 0)` should evaluate as true, causing the function to return immediately.

- **Test Case 2:** The vector `p` contains one `Point`, such as `Point(0, 0)`. Expected Outcome: The condition inside the function should evaluate to false, and the function should continue execution to find the `Point` with the minimum `y` value.
- **Test Case 3:** The vector `p` contains multiple `Point` objects, with no `Point` having a smaller `y`-coordinate than the first element (`p[0]`). Example: `p = [Point(0, 0), Point(1, 1), Point(2, 2)]`. Expected Outcome: The conditional statement inside the `for` loop will always evaluate to false, and the `minY` variable should remain unchanged, retaining the initial value of 0.
- **Test Case 4:** The vector `p` contains multiple `Point` objects, where another `Point` has a smaller `y`-coordinate than the first element (`p[0]`). Example: `p = [Point(2, 2), Point(1, 0), Point(0, 3)].` Expected Outcome: The condition inside the `for` loop will evaluate to true when a `Point` with a smaller `y`-coordinate is encountered, and the `minY` variable will be updated accordingly.

3) Basic Condition Coverage.

➔ Basic condition coverage requires testing each individual condition within a branch independently, ensuring all possible outcomes (true/false) for each condition. Test Scenarios:

- **Test Case 1:** The input vector `p` is empty (`p.size() == 0`). Expected Outcome: The condition `if (p.size() == 0)` should evaluate to true, causing the function to exit early.
- **Test Case 2:** The input vector `p` is non-empty (`p.size() > 0`). Expected Outcome: The condition `if (p.size() == 0)` should evaluate to false, and the function should proceed to the next logic.
- **Test Case 3:** The vector `p` contains multiple `Point` objects, where the condition `p.get(i).y < p.get(minY).y` holds true for some element.
  Example: `p = [Point(1, 1), Point(0, 0), Point(2, 2)]`

Expected Outcome: The condition `p.get(i).y < p.get(minY).y` should evaluate to true for one of the points, leading to an update of the `minY` variable.

- **Test Case 4:** The vector `p` contains multiple `Point` objects, where `p.get(i).y == p.get(minY).y` is true, and also `p.get(i).x < p.get(minY).x` is true.
  Example: `p = [Point(1, 1), Point(0, 1), Point(2, 2)]`
  Expected Outcome: Both conditions — `p.get(i).y == p.get(minY).y` and `p.get(i).x < p.get(minY).x` — should evaluate to true, resulting in an update of the `minY` value.

Determine the minimum number of test cases required to achieve coverage based on the criteria listed above.

Answer:

- **Statement Coverage**: 3 test cases
- **Branch Coverage**: 4 test cases
- **Basic Condition Coverage**: 4 test cases
- **Path Coverage**: 3 test cases

Summary of Minimum Test Cases:

- **Total**: 3 (Statement) + 4 (Branch) + 4 (Basic Condition) + 3 (Path) = 14 test cases

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.

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.

1. **Deletion Mutation:**

➔ **Mutation:** Remove the assignment of minY to 0 at the beginning of the method.

```
public class ConvexHull {
    public void doGraham(Vector<Point> p) {
        if (p.size() == 0) {
            return;
        }
        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;
            }
        }
        Point temp = p.get(0);
        p.set(0, p.get(minY));
        p.set(minY, temp);
    }
}
```

2. **Insertion Mutation:**

➔ **Mutation:** Insert a line that overrides minY incorrectly based on a condition that should not occur.

```
public
void doGraham(Vector<Point> p)
{
   if (p.size() == 0)
   {
      return;
   }
   int minY = 0;
   if (p.size() > 1)
   {
      minY = 1;
   }
   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;
      }
   }
```

3. Modification Mutation:

➔ **Mutation:** Change the logical operator from || to && in the conditional statement.

```
public
class ConvexHull
{
public
   void doGraham(Vector<Point> p)
   {
      if (p.size() == 0)
      {
         return;
      }
      int minY = 0;
      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;
         }
      }
      Point temp = p.get(0);
      p.set(0, p.get(minY));
      p.set(minY, temp);
   }
}
```

❖ **Analysis of Detection by Test Cases:-**

1. **Statement Coverage:**
    - The removal of the initialization of `minY` might not be detected by the tests, as it may not trigger an immediate exception or error, depending on how the rest of the code handles the uninitialized variable.
2. **Branch Coverage:**
    - The modification that sets `minY` to 1 could lead to incorrect results. However, if the test cases do not specifically verify the final positions of the points or the value of `minY` after execution, this issue may remain undetected.
3. **Basic Condition Coverage:**
    - Altering the logical operator from `||` (OR) to `&&` (AND) does not lead to an application crash, and the existing test cases do not check if `minY` updates correctly under this change. As a result, this issue may go unnoticed during testing.

Q4. Create a test set that satisfies the path coverage criterion where every loop is explored at least zero, one or two times. Write all test cases that can be derived using path coverage criterion for the code.

Test Case 1: No Loop Iterations

- Input: An empty vector `p`.
- Test: `Vector<Point> p = new Vector<Point>();`
- Expected Outcome: Since the vector is empty, the method should return immediately without processing. This test case covers the scenario where the vector size is zero, leading to an early exit condition in the method.

Test Case 2: Loop Iterates Once

- Input: A vector containing a single point.
- Test: `Vector<Point> p = new Vector<Point>(); p.add(new Point(0, 0));`

- Expected Outcome: The method should not enter the loop, as `p.size()` is 1. The vector remains unchanged since the first point swaps with itself. This test case covers the scenario where the loop runs just once.

Test Case 3: Loop Iterates 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 Outcome: The method should enter the loop and compare the two points. The second point has a lower `y`-coordinate, so `minY` will be updated to 1, and a swap will occur, placing the point `(0, 0)` at the beginning of the vector.

Test Case 4: Loop Iterates More Than Twice

- 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 Outcome: The loop should iterate over all three points. The second point `(1, 0)` has the smallest `y`-coordinate, so `minY` will be updated to 1. After the swap, the vector should have the point `(1, 0)` at the front.