IT-314

202201236

Fenil Amin

Lab-9

Mutation Testing

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.

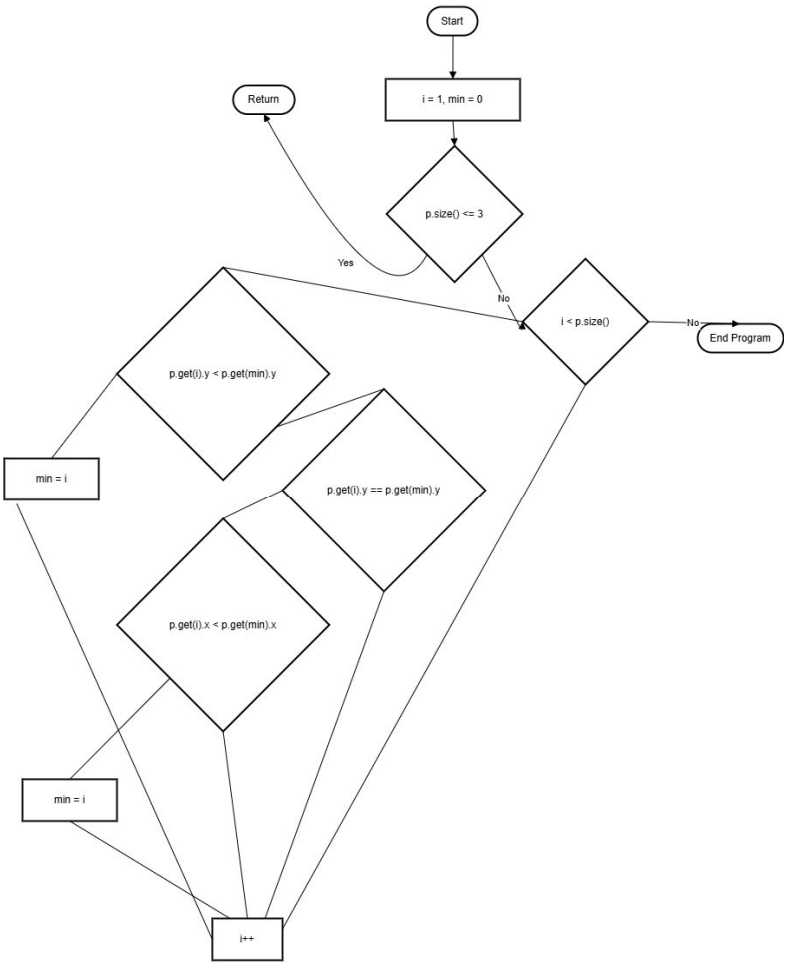
```
Vector doGraham(Vector p) {
        int i,j,min,M;

        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;
            }
        }
```

→Executable code in c++:-

```cpp
#include <bits/stdc++.h>
using namespace std;
#define ll long long
#define ld long double
#define pb push_back
class pt {
public:
double x, y;
pt(double x, double y) {
this->x = x;
this->y = y;
}
};
class ConvexHull {
public:
void DoGraham(vector<pt>& p) {
ll i = 1;
ll min = 0;
if (p.size() <= 3) {
return;
}
while (i < p.size()) {
if (p[i].y < p[min].y) {
min = i;
}
else if (p[i].y == p[min].y) {
if (p[i].x < p[min].x) {
min = i;
}
```

```
}
i++;
}
}
};
int32_t main() {
vector<pt> polls;
polls.pb(pt(0, 0));
polls.pb(pt(1, 1));
polls.pb(pt(2, 2));
ConvexHull hull;
hull.DoGraham(polls);
}
```

1. 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.



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.

   a) Statement Coverage:

   Objective: Ensure every line in the DoGraham method is executed at least once.

Test Case 1: p.size() <= 3

- **Input**: A vector p containing 3 or fewer points, such as (0, 0), (1, 1), (2, 2).
- **Expected Outcome**: The method exits immediately, covering the initial return condition.

Test Case 2: p.size() > 3, with points having unique y and x values

- **Input**: A vector p with points like (0, 0), (1, 2), (2, 3), (3, 4).
- **Expected Outcome**: The loop iterates through each point to find the point with the smallest y-value, covering the loop body and any associated if-else conditions.

Test Case 3: p.size() > 3, with points sharing the same y-values but different x-values

- **Input**: Points such as (1, 1), (2, 1), (0, 1), (3, 2).
- **Expected Outcome**: The loop identifies the point with the smallest x-value among those with the same y-value, testing both y and x comparisons.

Test Case 4: p.size() > 3, with all points having identical y and x values

- **Input**: Points such as (1, 1), (1, 1), (1, 1), (1, 1).
- **Expected Outcome**: The loop iterates without any updates to the min variable, as all points are identical, ensuring the loop completes without altering min.

b) Branch Coverage:

Objective: Ensure that both true and false outcomes for each decision point in the DoGraham method are tested.

Test Case 1: p.size() <= 3

- **Input**: A vector p with 3 or fewer points, such as (0, 0), (1, 1), (2, 2).
- **Expected Outcome**: The method exits immediately, covering the false branch for the loop entry condition.

Test Case 2: p.size() > 3, with all y-values distinct

- **Input**: Points like (0, 0), (1, 1), (2, 2), (3, 3).
- **Expected Outcome**: Covers the true branch of the main if condition (p[i].y < p[min].y) for each unique y-value.

Test Case 3: p.size() > 3, with some points having the same y-values but different x-values

- **Input**: Points such as (1, 1), (2, 1), (0, 1), (3, 2).
- **Expected Outcome**: Covers the true branch of the if condition (when y is lower) and tests both true and false outcomes of the else-if condition (when y is the same but x is different).

Test Case 4: p.size() > 3, with multiple points having identical y and x values

- **Input**: Points such as (1, 1), (1, 1), (1, 1), (1, 1).
- **Expected Outcome**: The loop iterates, covering the false branch for all comparisons since no updates to min occur.

c) Basic Condition Coverage:

Objective: Test each atomic condition in the method independently to cover all possible outcomes.

Conditions:

1. p.size() <= 3
2. p[i].y < p[min].y
3. p[i].y == p[min].y
4. p[i].x < p[min].x

Test Case 1: p.size() <= 3

- **Input**: A vector p with 3 or fewer points, such as (0, 0), (1, 1), (2, 2).
- **Expected Outcome**: Tests the true condition for p.size() <= 3.

Test Case 2: p.size() > 3, with points having distinct y-values

- **Input**: Points like (0, 0), (1, 1), (2, 2), (3, 3).
- **Expected Outcome**: Tests the true outcome for p[i].y < p[min].y, as each point has a higher y-value than the initial min.

Test Case 3: p.size() > 3, with points having the same y-value but different x-values

- **Input**: Points like (0, 1), (2, 1), (3, 1), (1, 0).
- **Expected Outcome**: Tests the true outcome for p[i].y == p[min].y and both true and false outcomes for p[i].x < p[min].x.

Test Case 4: p.size() > 3, with identical y and x values

- **Input**: Points like (1, 1), (1, 1), (1, 1), (1, 1).
- **Expected Outcome**: Tests the false outcomes for p[i].y < p[min].y, p[i].y == p[min].y, and p[i].x < p[min].x, as no changes occur to min.

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: Modify the code by removing the specific condition or line, such as the check if (p.size() <= 3) at the beginning of the method.

   Code:

```
void DoGraham(vector<pt>& p) {
    ll i = 1;
    ll min = 0;
    // Removed condition check for p.size() <= 3
    while (i < p.size()) {
        if (p[i].y < p[min].y) {
            min = i;
        } else if (p[i].y == p[min].y) {
            if (p[i].x < p[min].x) {
                min = i;
            }
        }
        i++;
    }
}
```

Expected Outcome: Removing this condition will allow the method to execute even when p.size() <= 3, which could lead to unnecessary or incorrect

calculations when the input size is too small. Current tests that only check when p.size() > 3 might miss this, allowing the mutation to go undetected.

Test Case Needed: A test case where p.size() is exactly 3 (for example, points (0,0), (1,1), and (2,2)) would help identify this issue, as the method would now perform unnecessary calculations.

2)Change Mutation: Modify a condition, variable, or operator within the code

Mutation Example: Change the < operator to <= in the condition if(p[i].y < p[min].y).

Code:

```
void DoGraham(vector<pt>& p) {
    ll i = 1;
    ll min = 0;
    if (p.size() <= 3) {
        return;
    }
    while (i < p.size()) {
        if (p[i].y <= p[min].y) {  // Changed < to <=
            min = i;
        }
        else if (p[i].y == p[min].y) {
            if (p[i].x < p[min].x) {
                min = i;
            }
        }
        i++;
    }
}
```

Expected Outcome: With the <= condition, points with equal values may incorrectly replace the minimum, potentially leading to the incorrect selection of the minimal point. Since the current test cases may not fully test scenarios with multiple points having identical values, this issue could go unnoticed.

Test Case Needed: Include a test case with multiple points having identical values (e.g., (2,1), (1,1), (3,1), (0,1)) to ensure that the smallest x-value is selected correctly. This will help identify the issue of selecting the last instance instead of the actual minimum.

3) **Insertion Mutation:** Add additional statements or conditions to the code.

Mutation Example: Insert a line that resets the minimum index at the end of each loop iteration.

Code:

```
void DoGraham(vector<pt>& p) {
    ll i = 1;
    ll min = 0;
    if (p.size() <= 3) {
        return;
    }
    while (i < p.size()) {
        if (p[i].y < p[min].y) {
            min = i;
        }
        else if (p[i].y == p[min].y) {
            if (p[i].x < p[min].x) {
                min = i;
            }
        }
        i++;
        min = 0; // Added mutation: resetting min after each iteration
    }
}
```

Expected Outcome: Resetting the minimum after each iteration prevents the code from properly tracking the actual minimum point found so far, as the minimum is always reset to 0. This can lead to incorrect results, especially if the sequence of points includes cases where the smallest y or x value is not at the beginning.

Test Case Needed: A vector of points with different positions for the minimum values, such as [(5, 5), (1, 0), (2, 3), (4, 2)], will help identify this issue. The correct minimum should persist across iterations, but this mutation will cause it to fail.

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

Test Case 1: p.size() = 0 (Zero iterations)

- **Input:** p = [] (empty vector)
- **Expected Outcome:** Since p.size() == 0, the method will exit immediately without entering the loop.
- **Path Covered:** This tests the path where the loop condition fails initially, and as a result, the loop is not executed.

Test Case 2: p.size() = 1 (Zero iterations)

- **Input:** p = [(0, 0)] (a single point)
- **Expected Outcome:** Since p.size() <= 3, the method will return immediately, skipping the loop.
- **Path Covered:** This tests the case where the method exits early due to the condition p.size() <= 3.

Test Case 3: p.size() = 4 with the loop executing once

- **Input:** p = [(0, 0), (1, 1), (2, 2), (3, 3)]
- **Expected Outcome:** The method checks if p.size() > 3 and enters the loop. On the first iteration, it evaluates (1, 1), updates the minimum to the index with the lowest y or x value, and exits.
- **Path Covered:** This tests the case where the loop executes once before completing.

Test Case 4: p.size() = 4 with the loop executing twice

- **Input:** p = [(0, 0), (1, 2), (2, 1), (3, 3)]
- **Expected Outcome:** The method evaluates the first two points in the loop to find the minimum. After two iterations, it updates the minimum and prepares to either continue or exit.
- **Path Covered:** This tests the case where the loop executes exactly twice.

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).

   Ans : YES

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

   →Branch Coverage = 3

→Basic Condition Coverage = 3
→Path Coverage = 3