IT 314

Software Engineering

Lab 09-Mutation Testing

Supervised by:
Prof. Saurabh Tiwari

Submitted by:
Kavan Kansodariya: 202201223

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.**
    a. **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.**

Ans:

Code:

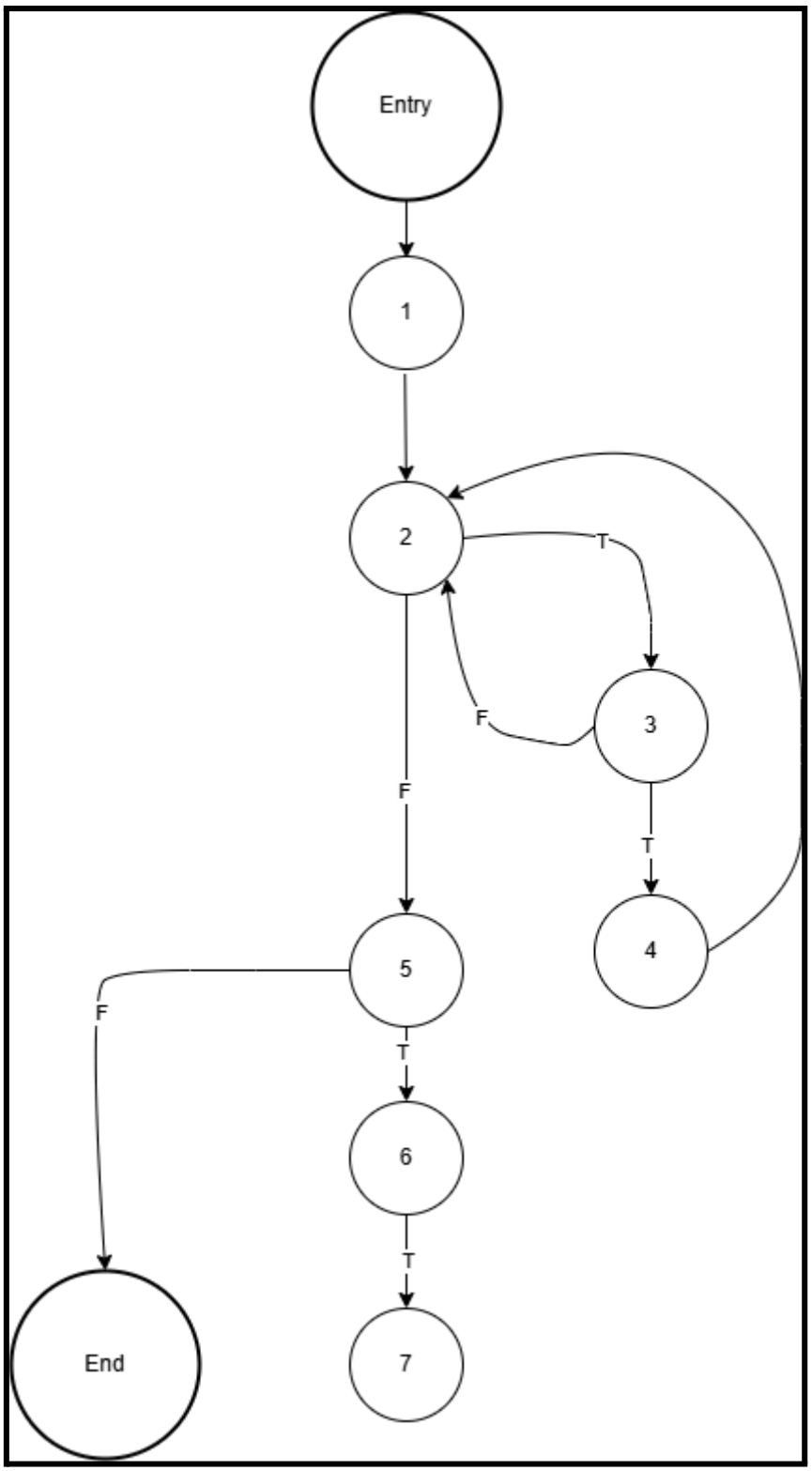
```
#include <bits/stdc++.h>
using namespace std;

class Point
{
public:
    int x, y;
    Point(int x, int y) : x(x), y(y) {}
};

vector<Point> doGraham(vector<Point> &p)
{
    int minInd = 0;

    for (int i = 0; i < p.size(); ++i)
    {
        if (p[i].y < p[minInd].y)
        {
            minInd = i;
        }
    }

    for (int i = 0; i < p.size(); ++i)
    {
        if (p[i].y == p[minInd].y && p[i].x > p[minInd].x)
        {
```

```
            minInd = i;
        }
    }

    cout << "Bottom-rightmost point: (" << p[minInd].x << ", " <<
p[minInd].y << ")"
         << endl;

    return p;
}

int main()
{
    vector<Point> points = {Point(1, 2), Point(3, 4), Point(5, 2),
Point(2, 2), Point(0, 5)};
    doGraham(points);
    return 0;
}
```

Nodes:

1. minInd=0
2. for i in range(1, len(p)):
    3. if p[i].y <p[minInd].y:
    4. minInd=i;
5. for i in range(len(p))
6. If p[i].y ==p[minInd].y and p[i].x >p[minInd].x:
    7. minInd=i;

Control flow diagram:



2. Construct test sets for your flow graph that are adequate for the following criteria:

Ans:

a. **Statement Coverage:** To achieve statement coverage, we need to ensure that every statement in the code is executed at least once. Here are some test cases to ensure that:
Test case 1: Single point case
Input : p(0,0)
Expected result: minInd=0;
Path will be : Entry → 1 → 2 False → 5 → 6 False → Exit

Test case 2 : Multiple points
Input: p(1,3) , p(2,2), p(0,1)
Expected Result: minInd=2
Path will be : Entry → 1→ 2 True → 3 True → 4 →2 False → 5 → 6 False → Exit

b. **Branch coverage**
To achieve branch coverage, we need to ensure that all branches (true/false paths) in the code are executed. Here are the conditions we need to cover:
   - if (p[i].y < p[min].y) (both true and false cases)
   - if (p[i].y == p[min].y && p[i].x > p[min].x) (both true and false cases)

Test Cases for Branch Coverage:

- **Test case 1 :** Input points: [(1, 2), (3, 4), (5, 2), (2, 2), (0, 5)]
  - Covers cases where points with the same y value but different x values exist.
- **Test case 2 :** Input points: [(0, 1), (0, 0), (0, -1)]
  - This test ensures the code handles cases where all x values are the same, testing the condition p[i].x > p[min].x.

Expected outcomes:

- Test Case 1 should output the point (5, 2).
- Test Case 2 should output the point (0, -1) as the bottom-rightmost point.

c. **Basic Condition Coverage:** For basic condition coverage, we need to test each individual condition within compound statements to evaluate both true and false outcomes independently.

Test Cases for Basic Condition Coverage:

- **Test case 1:** Input points: [(1, 2), (3, 4), (5, 2), (2, 2), (0, 5)]
- **Test case 2:** Input points: [(5, 2), (3, 2), (1, 2)]
  - Ensures that the condition p[i].y == p[min].y holds, but p[i].x > p[min].x will be evaluated multiple times.

Expected outcomes:

- Test Case 1 should output the point (5, 2).
- Test Case 2 should output the point (5, 2) as well, verifying the maximum x for the same minimum y.

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

- 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: 14 Test case

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

Code:

```
public class ConvexHull {

    public void doGraham(Vector<Point> p) {
        if (p.size() < 3) 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);
    }
}
```

2. **Insertion mutation:**
   Insert a line that overrides minY incorrectly based on a condition that should not occur.

Code:

```
void p(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:**
   Change the logical operator from || to && in the conditional statement.

Code:

```
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.

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

- **Test case 1(loop 0 times) :**input an empty list of points [].

```
[] // empty list of points
```

  - **Expected outcomes:** Should handle this ideally by returning an error message or a special indication (such as "No points provided") without causing runtime errors.
  - **Path Coverage:** This path skips the for loops entirely, covering the scenario where there are no elements to process.

- **Test case 2 (loop 1 time:** Input a single point [(0, 0)].

```
[(0, 0)] // Single point
```

  - **Expected Outcome**: The output should be: Bottom-rightmost point: (0, 0).
  - **Path Covered:** The first loop will iterate only once, identifying (0, 0) as the minimum y and maximum x point by default. The second loop will also iterate only once, confirming that (0, 0) is the bottom-rightmost point.

- **Test case 3(loop 2 times):** Input two points with the same y but different x values, such as [(1, 0), (2, 0)].

```
[(1, 0), (2, 0)] // Two points with the same y-coordinate but
different x-coordinates
```

  - **Expected Outcome:** The output should be: Bottom-rightmost point: (2, 0).
  - **Path Covered**:
    - The first loop will iterate twice. The code will first select (1, 0) as the minimum y point and then replace it with (2, 0) due to the larger x value.
    - The second loop confirms (2, 0) as the bottom-rightmost point since it has the same y but a larger x than (1, 0).

- **Test case 4(loop multiple times):** Use a standard test case like [(1, 2), (3, 4), (5, 2), (2, 2), (0, 5)].

```
[(1, 2), (3, 4), (5, 2), (2, 2), (0, 5)] // Multiple points with distinct y and x values
```

  - **Expected Outcome:** The output should be: Bottom-rightmost point: (5, 2).
  - **Path Covered:**
    - The first loop iterates over all points to find the minimum y point. It identifies (1, 2) initially but later replaces it with (5, 2) due to the same y and larger x.
    - The second loop iterates over all points again to confirm that (5, 2) is indeed the bottom-rightmost point.

- **Test case 5(loop multiple times):** Loop Multiple times with multiple points having the same minimum y-value.

```
[(2, 1), (1, 1), (3, 1), (0, 1)] // Multiple points with identical y values but different x-values
```

  - **Expected Outcome:** The output should be: Bottom-rightmost point: (3, 1).
  - **Path Covered:**
    - The first loop iterates over all points and identifies (2, 1) initially but eventually selects (3, 1) as it has the same y but a larger x than others.
    - The second loop confirms (3, 1) as the bottom-rightmost point.

Lab Execution:

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: Control Flow Graph Factory Tool: YES