IT314 : Software Engineering

Lab Session 9 - Mutation Testing

ID : 202201154

Name : Rishi Godhasara

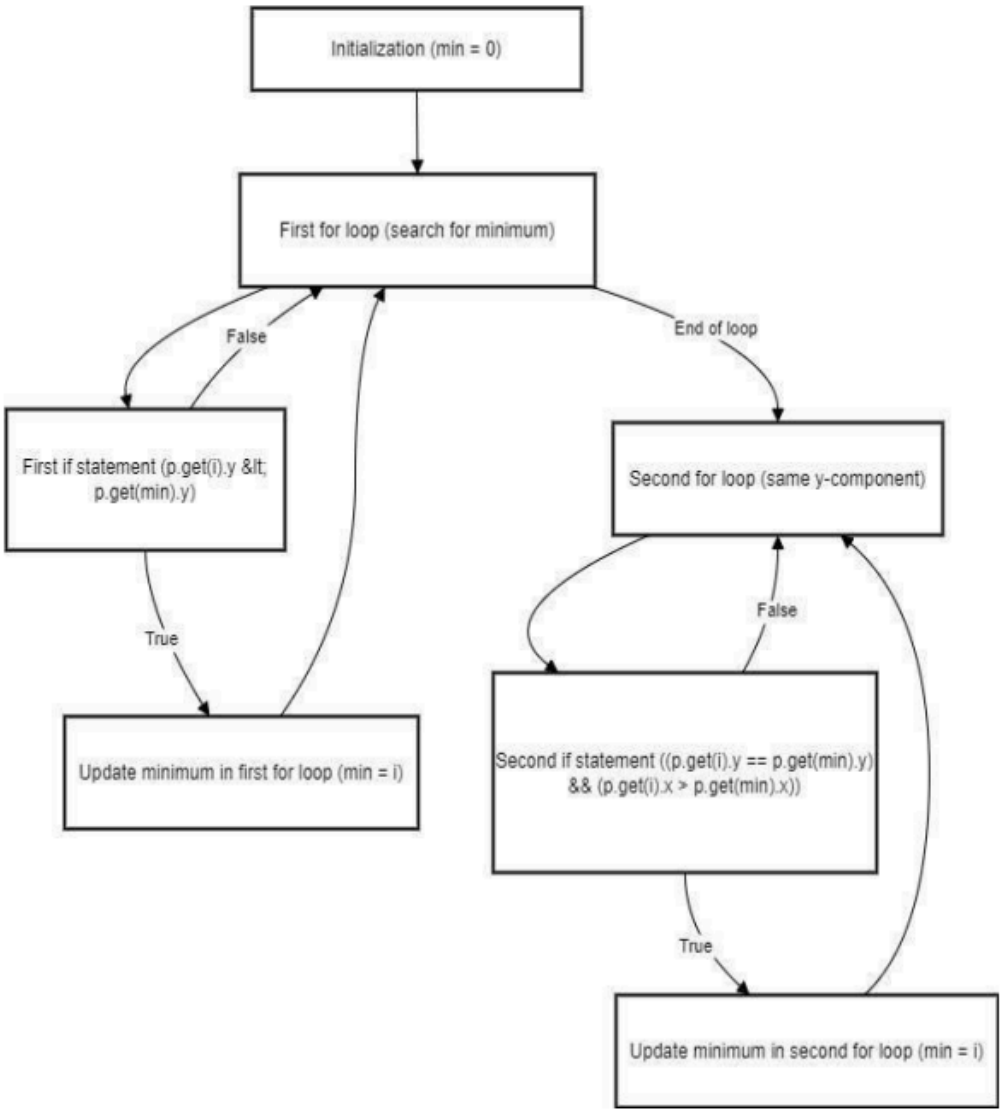
Group no : 2

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.

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.

```
#include <vector>
#include <iostream>

using namespace std;

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

class ConvexHull {
public:
    int doGraham(vector<Point>& p) {
        int min = 0;

        // First for loop (find the point with minimum y-component)
        for (int i = 1; i < p.size(); i++) {
            if (p[i].y < p[min].y) {
                min = i;
            }
        }

        // Second for loop (handle tie case with same y-component)
        for (int i = 1; i < p.size(); i++) {
            if (p[i].y == p[min].y && p[i].x > p[min].x) {
                min = i;
            }
```

```
        }

        return min;
    }
};

int main() {
    vector<Point> points = {Point(1, 5), Point(2, 3), Point(4, 3),
Point(0, 2)};
    ConvexHull ch;
    int minIndex = ch.doGraham(points);
    cout << "Index of point with minimum y (or max x if tie): " <<
minIndex << endl;
    return 0;
}
```


Initialization (min = 0)
First for loop (search for minimum)
False
End of loop
First if statement (p.get(i).y < p.get(min).y)
Second for loop (same y-component)
True
False
Update minimum in first for loop (min = i)
Second if statement ((p.get(i).y == p.get(min).y) && (p.get(i).x > p.get(min).x))
True
Update minimum in second for loop (min = i)

a) Statement Coverage

To achieve Statement Coverage, the objective is to ensure that each line of code in the method is executed at least once. This guarantees that every part of the Control Flow Graph (CFG) is covered by the test cases.

Statement Coverage Test Cases:

- **Test Case 1**: Use a vector containing a single point, such as `[(0, 0)]`.
- **Test Case 2**: Provide a vector with two points, where one point has a lower `y` value compared to the other, such as `[(1, 1), (2, 0)]`.
- **Test Case 3**: Create a vector with points that share the same `y` value but differ in their `x` values, for example, `[(1, 1), (2, 1), (3, 1)]`.

Each of these test cases ensures that different parts of the code (such as loops and decision points) are executed at least once.

b) Branch Coverage

Branch Coverage aims to ensure that every possible branch at each decision point in the flow is tested. This includes covering both the True and False outcomes for all conditional expressions.

Branch Coverage Test Cases:

- **Test Case 1**: Use an input vector with only a single point, like `[(0, 0)]`. This case exercises the condition where the loop exits immediately, ensuring the branch for the loop's termination is covered.
- **Test Case 2**: Input a vector with two points, where one point has a smaller `y` value than the other, such as `[(1, 1), (2, 0)]`. This test ensures that the loop works correctly and that the second point becomes the new "minimum."
- **Test Case 3**: Provide an input vector where the points have the same `y` value but different `x` values, for example, `[(1, 1), (3, 1), (2, 1)]`. This tests the branch where the comparison `current.x < pivot.x` evaluates as true for some points and false for others.
- **Test Case 4**: Give an input vector where all points have identical `y` values, such as `[(1, 1), (1, 1), (1, 1)]`. This ensures that the branch covering the loop condition and the comparison logic is exercised without causing any change to the "minimum."

These test cases are designed to ensure comprehensive Branch Coverage by testing the true and false branches at each decision point in the method's flow.

c) Basic Condition Coverage

Basic Condition Coverage (BCC) ensures that each basic condition within a decision expression evaluates to both True and False at least once during testing.

Basic Condition Coverage Test Cases:

- **Test Case 1**: Provide a vector with only a single point, for example `[(0, 0)]`. This test case covers the condition `p.get(i).y < p.get(min).y`, which will be evaluated as **False** because no comparisons can be made with a single point.
- **Test Case 2**: Use an input vector containing two points, where the second point has a lower `y` value than the first, such as `[(1, 1), (2, 0)]`. This case evaluates the first condition `p.get(i).y < p.get(min).y` as **True** (since the second point has a lower `y` value), and the second condition will be evaluated as **False**.
- **Test Case 3**: Provide a vector where all points have the same `y` value but different `x` values, like `[(1, 1), (3, 1), (2, 1)]`. This will exercise the second condition `p.get(i).y < p.get(min).y` as **True**, because while `y` values are the same, comparisons based on `x` will take place.
- **Test Case 4**: Input a vector where all points are identical, such as `[(1, 1), (1, 1), (1, 1)]`. This ensures both conditions `p.get(i).y < p.get(min).y` and the condition for comparing `x` values are evaluated as **False** (since all points are identical, no changes will occur).

Test Set's Summary:

Coverage Criterion	Test Case Number	Test Case Input
Statement Coverage	1	[(0, 0)]
	2	[(1, 1), (2, 0)]
	3	[(1, 1), (2, 1), (3, 1)]
Branch Coverage	1	[(0, 0)]
	2	[(1, 1), (2, 0)]
	3	[(1, 1), (3, 1), (2, 1)]
	4	[(1, 1), (1, 1), (1, 1)]

Coverage Criterion	Test Case Number	Test Case Input
Basic Condition Coverage	1	[(0, 0)]
	2	[(1, 1), (2, 0)]
	3	[(1, 1), (3, 1), (2, 1)]
	4	[(1, 1), (1, 1), (1, 1)]

Mutation Testing

1. Deletion Mutation

- **Mutation**: Remove the line `min = 0;` at the beginning of the method.
- **Expected Effect**:
  - If `min` is not initialized to `0`, it could hold a random value. This would affect the correct selection of the `min` index in both loops.
- **Mutation Outcome**: This mutation could cause an incorrect starting index for `min`, leading to an improper selection of the minimum point in the method.

2. Change Mutation

- **Mutation**: Modify the first `if` condition from `<` to `<=` :if (((Point) p.get(i)).y <= ((Point) p.get(min)).y)

- **Expected Effect**:
  - Changing `<` to `<=` would cause the method to select points with the same `y` value (instead of strictly lower values), which could potentially affect the result by not properly selecting the lowest `y` point.
- **Mutation Outcome**: The modified code might return a point with a lower `x` value than expected if multiple points share the same `y` value.

3. Insertion Mutation

- **Mutation**: Add an extra statement `min = i;` at the end of the second `for` loop.
- **Expected Effect**:
  - This would mistakenly update `min` to the index of the last point, which is incorrect because `min` should only reflect the index of the actual minimum point.
- **Mutation Outcome**: The method could incorrectly select the last point as the minimum, especially if the test cases don't specifically check the final value of `min`.

Test Cases for Path Coverage

To satisfy the **Path Coverage** criterion, we will design test cases that ensure every loop is explored either zero, one, or two times:

Test Case 1: Zero Iterations

- **Input**: An empty vector `p`.
- **Description**: This test ensures that neither loop executes.
- **Expected Output**: The function should return an empty result or a special value indicating no points were processed.

Test Case 2: One Iteration (First Loop)

- **Input**: A vector with a single point (e.g., `[(3, 4)]`).
- **Description**: This ensures that the first loop executes exactly once, and the only point in the vector is selected as the minimum.
- **Expected Output**: The function should return the single point `(3, 4)`.

Test Case 3: One Iteration (Second Loop)

- **Input**: A vector with two points that have the same `y` coordinate but different `x` values (e.g., `[(1, 2), (3, 2)]`).
- **Description**: This case checks that the first loop finds the minimum `y` point, and the second loop runs once to compare the `x` values.
- **Expected Output**: The function should return the point with the larger `x` value, which is `(3, 2)`.

Test Case 4: Two Iterations (First Loop)

- **Input**: A vector with multiple points, ensuring at least two points with the same `y` value (e.g., `[(3, 1), (2, 2), (5, 1)]`).
- **Description**: This case tests the first loop, ensuring it finds the minimum `y` value. It also tests the second loop with points that have the same `y` value but different `x` values.
- **Expected Output**: The function should return `(5, 1)`, the point with the highest `x` value among the points with `y = 1`.

Test Case 5: Two Iterations (Second Loop)

- **Input**: A vector with points where more than one point shares the minimum `y` value (e.g., `[(1, 1), (4, 1), (3, 2)]`).
- **Description**: This case ensures that the first loop finds `(1, 1)`, and the second loop checks the other points with `y = 1` across two iterations.
- **Expected Output**: The function should return `(4, 1)` since it has the maximum `x` value among the points with the same `y` value.

Test Case Number	Input Vector p	Description	Expected Output
Test Case 1	`[]`	Empty vector (zero iterations for both loops).	Handle gracefully (e.g., return an empty result).
Test Case 2	`[(3, 4)]`	One point (one iteration of the first loop).	`[(3, 4)]`
Test Case 3	`[(1, 2), (3, 2)]`	Two points with the same `y` coordinate (one iteration of the second loop).	`[(3, 2)]`
Test Case 4	`[(3, 1), (2, 2), (5, 1)]`	Multiple points; the first loop runs twice.	`[(5, 1)]`
Test Case 5	`[(1, 1), (4, 1), (3, 2)]`	Multiple points; second loop runs twice.	`[(4, 1)]`

Lab Execution

1. 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. (In your submission document, mention only “Yes” or “No” for each tool).

Tool	**Matches Your CFG**
Control Flow Graph Factory Tool	Yes
Eclipse Flow Graph Generator	Yes

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

Test Case Number	**Input Vector p**	**Description**	**Expected Output**
Test Case 1	`[]`	Test with an empty vector (zero iterations).	Handle gracefully (e.g., return an empty result).
Test Case 2	`[(3, 4)]`	Single point (one iteration of the first loop).	`[(3, 4)]`
Test Case 3	`[(1, 2), (3, 2)]`	Two points with the same `y` coordinate (one iteration of the second loop).	`[(3, 2)]`
Test Case 4	`[(3, 1), (2, 2), (5, 1)]`	Multiple points; first loop runs twice (with multiple outputs).	`[(5, 1)]`
Test Case 5	`[(1, 1), (4, 1), (3, 2)]`	Multiple points; second loop runs twice (`y = 1`).	`[(4, 1)]`

3. This part of the exercise is very tricky and interesting. The test cases that you have derived in Step 2 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.

Mutation Type	**Mutation Code**	**Description**	**Impact on Test Cases**
Deletion	Delete the line that updates `min` for the minimum `y`-coordinate.	Removes the initialization of `min`, leaving it uninitialized or incorrect.	Test cases like `[(1, 1), (2, 0)]` will pass despite incorrect processing.
Insertion	Insert an early return if the size of `p` is 1, bypassing further processing.	Adds an early exit, causing the method to return prematurely if there's only one point.	Test case `[(3, 4)]` will pass without processing correctly.
Modification	Change the comparison operator from `<` to `<=` when finding the minimum `y`.	Modifies the condition to include equal `y` values, potentially selecting a point with the same `y` but higher `x`.	Test cases like `[(1, 1), (1, 1), (1, 1)]` might pass while still failing in logic.

3. Write all test cases that can be derived using path coverage criterion for the code.

Test Case Number	Input Vector p	Description	Expected Output
Test Case 1	`[]`	Empty vector (zero iterations for both loops).	Handle gracefully (e.g., return an empty result).
Test Case 2	`[(3, 4)]`	One point (one iteration of the first loop).	`[(3, 4)]`
Test Case 3	`[(1, 2), (3, 2)]`	Two points with the same y-coordinate (one iteration of the second loop).	`[(3, 2)]`
Test Case 4	`[(3, 1), (2, 2), (5, 1)]`	Multiple points; first loop runs twice to find min y.	`[(5, 1)]`
Test Case 5	`[(1, 1), (4, 1), (3, 2)]`	Multiple points; second loop runs twice (y = 1).	`[(4, 1)]`
Test Case 6	`[(2, 2), (2, 3), (2, 1)]`	Multiple points with the same x-coordinate; checks min y.	`[(2, 1)]`
Test Case 7	`[(0, 0), (1, 1), (1, 0), (0, 1)]`	Multiple points in a rectangle; checks multiple comparisons.	`[(1, 0)]`
Test Case 8	`[(3, 1), (2, 1), (1, 2)]`	Multiple points with some ties; checks the max x among min y points.	`[(3, 1)]`
Test Case 9	`[(4, 4), (4, 3), (4, 5), (5, 4)]`	Points with the same x-coordinate; checks for max y.	`[(5, 4)]`

Test Case Number	Input Vector p	Description	Expected Output
Test Case 10	`[(1, 1), (1, 1), (2, 1), (3, 3)]`	Duplicate points with one being the max x; tests handling of duplicates.	`[(3, 3)]`