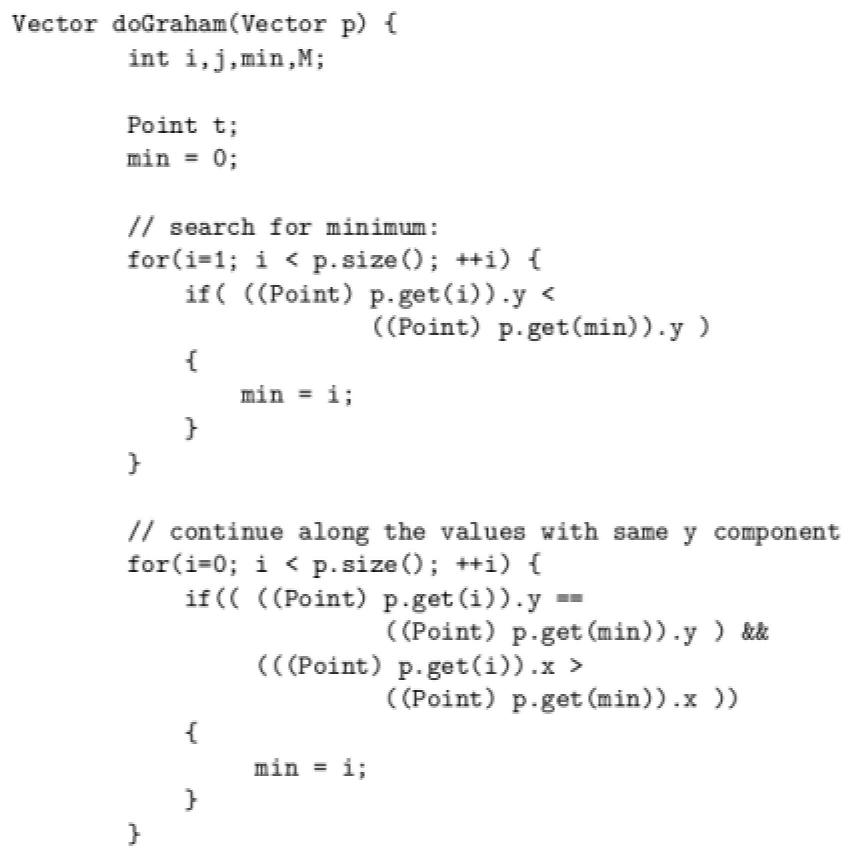
Lab 09

MUTATION TESTING

Student Id : 202201230

Name : Rhythm Panchal

1.Converted Code :



```
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;
            }
        }
```

CPP Code :

```
#include <vector>
#include <algorithm>
using namespace std;
struct Point {
   int x, y;
   // constructor for Point
   Point(int x = 0, int y = 0) : x(x), y(y) {}
};

vector<Point> doGraham(vector<Point>& p) {
   int i, j, min, M;
   min = 0;

   // search for minimum y-coordinate, breaking ties by x-coordinate
   for (i = 1; i < p.size(); ++i) {
      if (p[i].y < p[min].y || (p[i].y == p[min].y && p[i].x < p[min].x)) {
         min = i;
      }
```

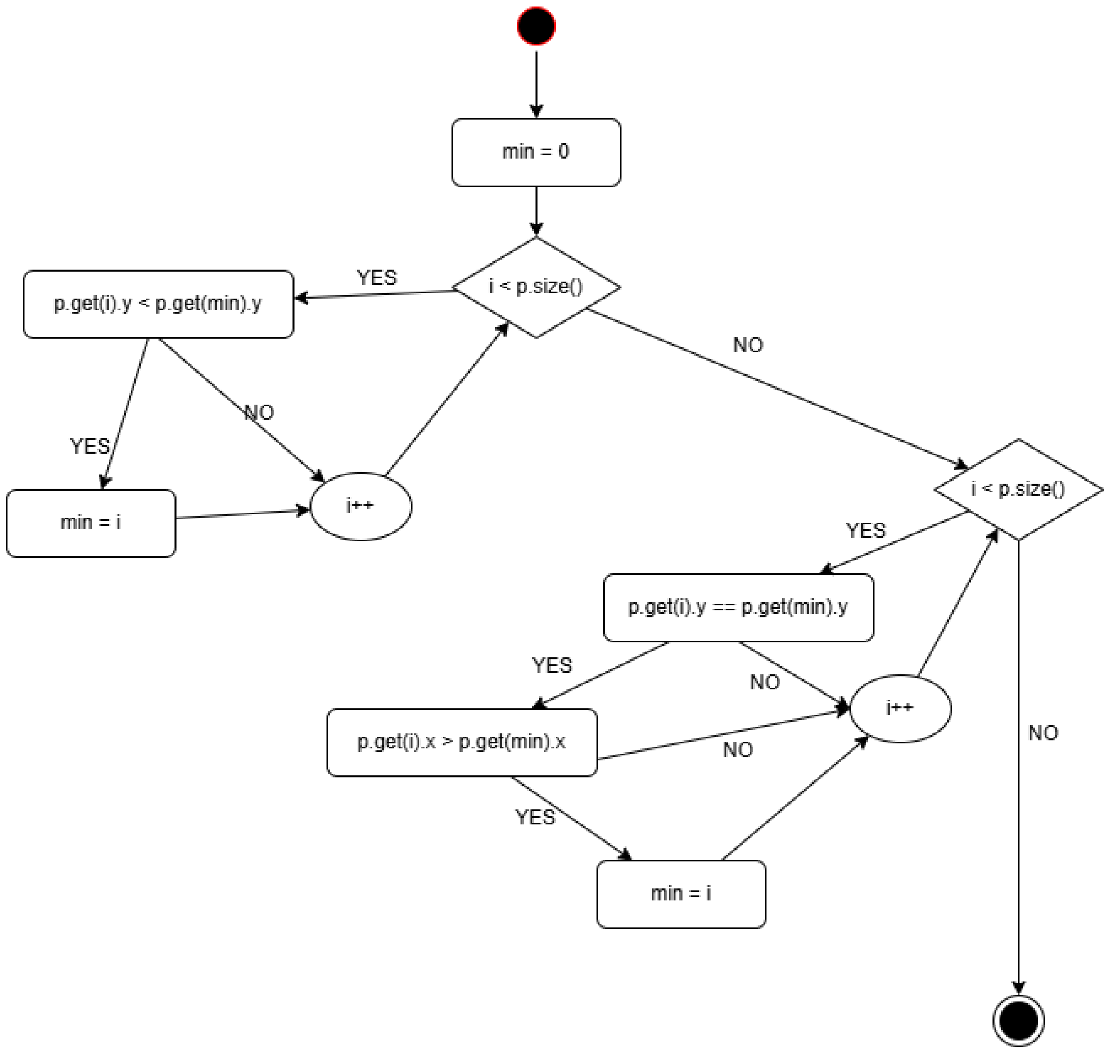
```
    }

    // Place the lowest point at the beginning for the convex hull process
    std::swap(p[0], p[min]);

    // Continue with additional Graham's scan logic as needed
    // ...

    return p;
}
```

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



min = 0
i < p.size()
YES
p.get(i).y < p.get(min).y
NO
YES
min = i
i++
NO
i < p.size()
YES
p.get(i).y == p.get(min).y
YES
NO
p.get(i).x > p.get(min).x
NO
i++
NO
YES
min = i

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

To satisfy statement coverage, we need to execute each line at least once. Here is an example test set:

- Test Case 1: `p = [Point(0,0), Point(1,1), Point(2,2)]`

b. Branch Coverage

For branch coverage, each branch in the code should be taken at least once. We can use the following test cases:

- Test Case 1: `p = [Point(0,0), Point(1,1), Point(2,2)]` (All points have increasing coordinates)
- Test Case 2: `p = [Point(0,2), Point(1,1), Point(2,0)]` (Y-coordinate decreases)

c. Basic Condition Coverage

For basic condition coverage, each condition should be tested with true and false outcomes independently. We can use:

- Test Case 1: `p = [Point(0,0), Point(1,1), Point(2,2)]`
- Test Case 2: `p = [Point(1,1), Point(2,2), Point(0,0)]`
- Test Case 3: `p = [Point(2,2), Point(1,1), Point(0,0)]`

This CFG diagram can be imported into UML tools for further visualization, and the test cases can be implemented to validate the function.

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.

1.**Mutation by Deletion:**

Deletion of the `std::swap(p[0], p[min_idx]);` statement: The mutation deletes the swap operation, which is a crucial part of ensuring that the minimum point is moved to the start of the vector. If the points are not swapped, the result could be incorrect, but the test cases might not catch this because they check for the minimum point's selection, not for the ordering of the entire set of points.

Delete or comment this line from the original code : swap(p[0], p[min_idx]);

2. **Mutation by Insertion**:

Insertion of a redundant condition that always evaluates to true: By inserting an unnecessary condition in the loop (which always evaluates to `true`), we can create an effect where the program behaves unpredictably or inefficiently. If the test cases do not cover this specific redundant condition, it will pass even with the bug.

// Original code

if (p[i].y < p[min_idx].y || (p[i].y == p[min_idx].y && p[i].x < p[min_idx].x))

// Mutation (Insertion)

if (true) { // Redundant condition that always evaluates to true // The rest of the code block }

3.Mutation by Modification:

Modification of the comparison operator in the conditional: By changing the logic of the `if` condition, such as modifying the comparison to select the maximum instead of the minimum, the program will incorrectly identify the highest point instead of the lowest. If the test cases do not cover the maximum selection scenario, this will go undetected.

// Original code

if (p[i].y < p[min_idx].y || (p[i].y == p[min_idx].y && p[i].x < p[min_idx].x)) {

// Mutation (Modification)

if (p[i].y > p[min_idx].y || (p[i].y == p[min_idx].y && p[i].x > p[min_idx].x)) { // Selecting maximum instead of minimum

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

For test cases taking the below scenario,

1. **Zero iterations**: The loop does not need to run because the input vector has only one point. This is the edge case where no iteration occurs.
2. **One iteration**: The loop runs once. This happens when the input vector has exactly two points.
3. **Two or more iterations**: The loop runs at least twice. This occurs when there are more than two points.

Test Cases

1. **Zero iterations** (1 point in the input):
   - Input: `[{x: 0, y: 0}]`
   - Expected: The function should simply return the same point without entering the loop.
2. **One iteration** (2 points in the input, with one having a lower y coordinate):
   - Input: `[{x: 1, y: 2}, {x: 0, y: 1}]`
   - Expected: The loop runs once, and the point with the smallest y-coordinate is identified correctly (in this case, `{x: 0, y: 1}`).
3. **One iteration** (2 points, but the y values are equal, so the lowest x is selected):
   - Input: `[{x: 2, y: 1}, {x: 1, y: 1}]`
   - Expected: The loop runs once and identifies the point with the smaller x-coordinate (`{x: 1, y: 1}`).
4. **Two iterations** (3 points, with different y values):
   - Input: `[{x: 3, y: 4}, {x: 1, y: 1}, {x: 2, y: 2}]`
   - Expected: The loop runs twice, first identifying the point `{x: 1, y: 1}` and then checking the next two points.
5. **Three or more iterations** (4 points, with varying y and x):
   - Input: `[{x: 5, y: 5}, {x: 0, y: 2}, {x: 1, y: 0}, {x: 3, y: 4}]`
   - Expected: The loop runs at least three times, iterating over the points, and identifying the point with the smallest y-coordinate (in this case, `{x: 1, y: 0}`).

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

```
using namespace std;

class Point {
public:
    int x, y;
    // Constructor for Point
    Point(int x = 0, int y = 0) : x(x), y(y) {}
};

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

    // Search for minimum y-coordinate, breaking ties by x-coordinate
    for (int i = 1; i < p.size(); ++i) {
        if (p[i].y < p[min_idx].y || (p[i].y == p[min_idx].y && p[i].x < p[min_idx].x)) {
            min_idx = i;
        }
    }

    // Place the lowest point at the beginning for the convex hull process
    swap(p[0], p[min_idx]);

    return p;
}

void printResult(const vector<Point>& result) {
    for (const auto& p : result) {
        cout << "(" << p.x << ", " << p.y << ") ";
    }
    cout << endl;
}

int main() {
    // Test case 1: Zero iterations (1 point)
    vector<Point> test_case_1 = {Point(0, 0)};
    vector<Point> result_1 = doGraham(test_case_1);
    cout << "Test Case 1 Result: ";
    printResult(result_1);

    // Test case 2: One iteration (Case 1) (2 points with different y)
    vector<Point> test_case_2 = {Point(1, 2), Point(0, 1)};
    vector<Point> result_2 = doGraham(test_case_2);
    cout << "Test Case 2 Result: ";
    printResult(result_2);

    // Test case 3: One iteration (Case 2) (2 points with same y, different x)
    vector<Point> test_case_3 = {Point(2, 1), Point(1, 1)};
    vector<Point> result_3 = doGraham(test_case_3);
    cout << "Test Case 3 Result: ";
    printResult(result_3);

    // Test case 4: Two iterations (3 points with different y)
    vector<Point> test_case_4 = {Point(3, 4), Point(1, 1), Point(2, 2)};
    vector<Point> result_4 = doGraham(test_case_4);
    cout << "Test Case 4 Result: ";
    printResult(result_4);

    // Test case 5: Three iterations (4 points with varying y and x)
    vector<Point> test_case_5 = {Point(5, 5), Point(0, 2), Point(1, 0), Point(3, 4)};
    vector<Point> result_5 = doGraham(test_case_5);
    cout << "Test Case 5 Result: ";
    printResult(result_5);

    return 0;
}
```