IT - 314 Software Engineering

Assignment 9: Mutation Testing

Prof. : Saurabh Tiwari

Name: Mavdiya Sujal Pravinbhai

ID: 202201040

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.

Code in C++ :-

```cpp
#include <iostream>
#include <vector>
using namespace std;

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

    friend ostream& operator<<(ostream& os, const Point& point) {
        os << "(" << point.x << ", " << point.y << ")";
        return os;
    }
};

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

        cout << "Searching for the minimum y-coordinate...\n";
        for (i = 1; i < p.size(); ++i) {
            cout << "Comparing " << p[i] << " with " << p[min] << "\n";
            if (p[i].y < p[min].y) {
                min = i;
                cout << "New minimum found: " << p[min] << "\n";
            }
        }

        cout << "Searching for the leftmost point with the same minimum y-
coordinate...\n";
        for (i = 0; i < p.size(); ++i) {
            cout << "Checking if " << p[i] << " has the same y as " << p[min]
<< " and a smaller x...\n";
            if (p[i].y == p[min].y && p[i].x < p[min].x) {
                min = i;
                cout << "New leftmost minimum point found: " << p[min] <<
"\n";
            }
        }

        cout << "Final minimum point: " << p[min] << "\n";
    }
};

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

Mermaid Code:-

graph TD

Start(["Start"]) --> A["Declare int i, min"]

A --> B["min = 0"]

B --> C["Output: 'Searching for the minimum y-coordinate...'"]

C --> D["i = 1; i < p.size(); ++i"]

D -- "True" --> E["Output: 'Comparing p.get(i) with p.get(min)'"]

E --> F["p.get(i).y < p.get(min).y"]

F -- "True" --> G["min = i"]

G --> H["Output: 'New minimum found'"]

H --> D

F -- "False" --> D

D -- "False (End of loop)" --> I["Output: 'Searching for the leftmost point with the same minimum y-coordinate...'"]

I --> J["i = 0; i < p.size(); ++i"]

J -- "True" --> K["Output: 'Checking if p.get(i) has the same y as p.get(min) and a smaller x...'"]

K --> L["p.get(i).y == p.get(min).y && p.get(i).x < p.get(min).x"]

L -- "True" --> M["min = i"]

M --> N["Output: 'New leftmost minimum point found'"]

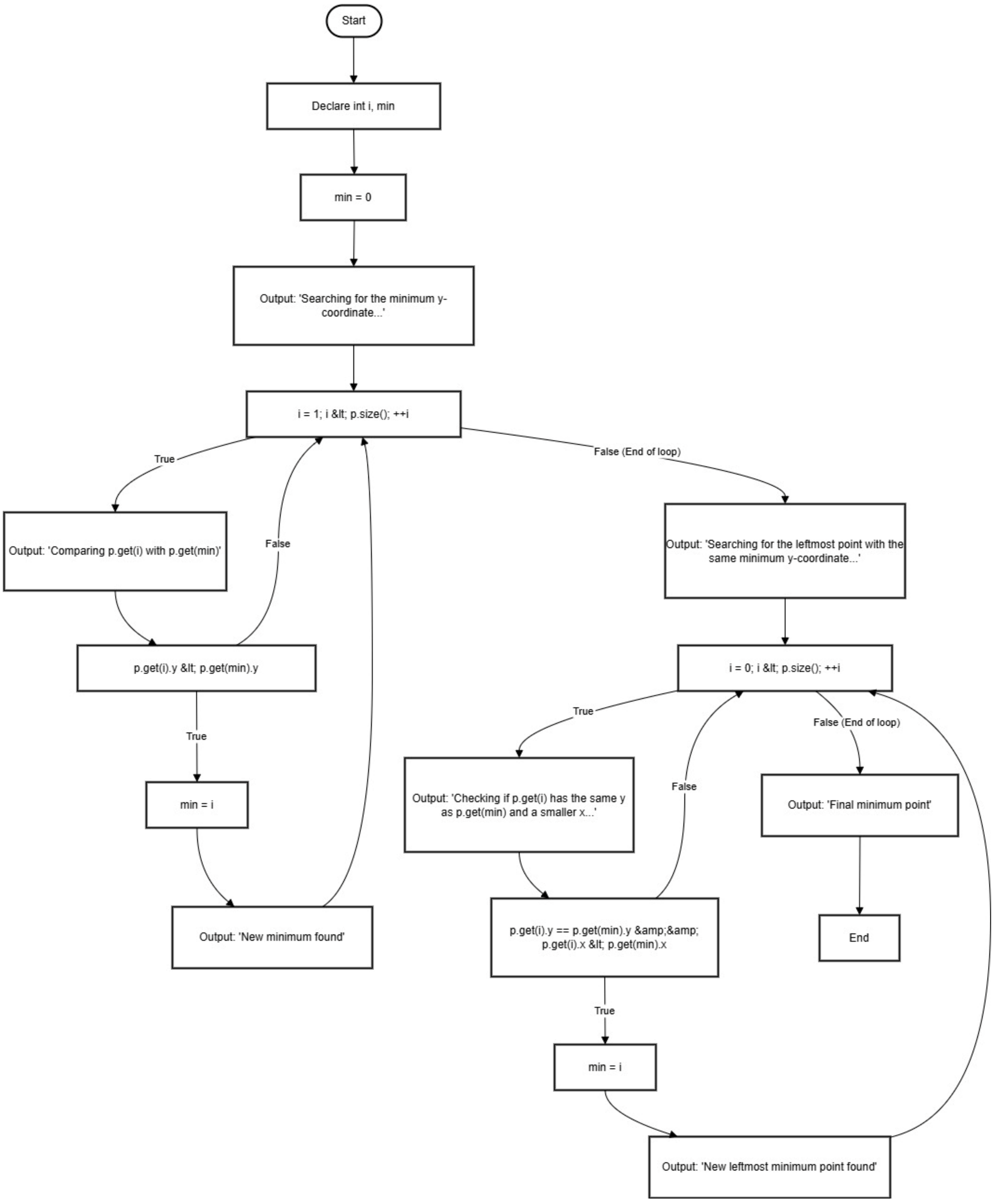
N --> J

L -- "False" --> J

J -- "False (End of loop)" --> O["Output: 'Final minimum point'"]

O --> End["End"]

Control Flow Diagram :-


Start
Declare int i, min
min = 0
Output: 'Searching for the minimum y-coordinate...'
i = 1; i &lt; p.size(); ++i
True
False (End of loop)
Output: 'Comparing p.get(i) with p.get(min)'
False
p.get(i).y &lt; p.get(min).y
True
min = i
Output: 'New minimum found'
Output: 'Searching for the leftmost point with the same minimum y-coordinate...'
i = 0; i &lt; p.size(); ++i
True
False (End of loop)
Output: 'Checking if p.get(i) has the same y as p.get(min) and a smaller x...'
False
Output: 'Final minimum point'
p.get(i).y == p.get(min).y &amp;&amp; p.get(i).x &lt; p.get(min).x
End
True
min = i
Output: 'New leftmost minimum point found'

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 of code in the flow is executed at least once.

Test Cases:

1. **Test Case 1**:
   - **Input**: `[(3, 4), (5, 2), (1, 3)]`
   - **Purpose**: This set will ensure statements related to finding the minimum y-coordinate and identifying the leftmost point are executed.
2. **Test Case 2**:
   - **Input**: `[(2, 5), (2, 5), (4, 7)]`
   - **Purpose**: Tests the scenario where multiple points share the same y-coordinate, thus engaging the leftmost point logic to verify all statements execute.

---

b. Branch Coverage

Objective: Validate that each decision branch (both true and false) is executed at least once.

Test Cases:

1. **Test Case 1**:
   - **Input**: `[(3, 2), (4, 5), (1, 1)]`
   - **Purpose**: This input ensures the true branch for finding the minimum y-coordinate is executed.
2. **Test Case 2**:
   - **Input**: `[(5, 3), (5, 3), (6, 4)]`
   - **Purpose**: Checks the branch where the y-coordinates are equal, triggering the code to compare x-coordinates to find the leftmost point.
3. **Test Case 3**:
   - **Input**: `[(2, 6), (1, 5), (4, 8)]`
   - **Purpose**: Validates the false branch of both minimum y-coordinate checking and the leftmost point comparison.

---

c. Basic Condition Coverage

Objective: Test each basic condition within the decision points independently to ensure both true and false evaluations are covered.

Test Cases:

1. **Test Case 1**:
   - **Input**: `[(4, 2), (3, 3), (5, 4)]`
   - **Purpose**: Confirms the condition evaluating y-coordinate comparisons runs with both outcomes.
2. **Test Case 2**:
   - **Input**: `[(3, 2), (3, 2), (2, 3)]`
   - **Purpose**: Ensures that when y-coordinates match, the x-coordinate comparison condition executes independently.
3. **Test Case 3**:
   - **Input**: `[(5, 1), (4, 3), (3, 5)]`
   - **Purpose**: Validates execution of both conditions within the loop, ensuring the algorithm's robustness.

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.

Types of Possible Mutations

Common mutation types that could be applied include:

- **Relational Operator Modifications**: Alter conditions by changing `<=` to `<`, `==` to `!=`, or vice versa.
- **Logical Modifications**: Remove or reverse branches within `if` statements.
- **Assignment Adjustments**: Modify assignment operations or statement values to check if unintended outcomes go unnoticed.

Potential Mutations and Their Effects

1. **Modifying the Comparison for Leftmost Point**:
   - **Mutation**: In the second loop, change `p.get(i).x < p.get(min).x` to `p.get(i).x <= p.get(min).x`.
   - **Effect**: This could allow selection of a point with the same `x` value as the leftmost, potentially causing issues with the uniqueness of the chosen minimum point.

    - **Undetected by Current Tests**: The existing test set does not specifically test cases where points share both `y` and `x` values. Such cases would expose if the function fails to distinguish unique leftmost points.
2. **Altering the y-Coordinate Condition to <= in the Initial Loop**:
    - **Mutation**: Change `p.get(i).y < p.get(min).y` to `p.get(i).y <= p.get(min).y` in the initial loop.
    - **Effect**: This mutation could let points with the same `y` value but different `x` values overwrite the `min` selection, potentially resulting in a non-leftmost point being chosen.
    - **Undetected by Current Tests**: The current tests lack scenarios where multiple points share the same `y` value but vary in `x` values, which would reveal this mutation's impact. Adding such cases would allow detection of issues in leftmost point selection.
3. **Removing the x-Coordinate Condition in the Second Loop**:
    - **Mutation**: Omit the condition `p.get(i).x < p.get(min).x` in the second loop.
    - **Effect**: The function would now select any point with the minimum `y` value as the "leftmost" without regard to `x` values, potentially selecting points that are not the actual leftmost.
    - **Undetected by Current Tests**: The current test set does not include cases with identical `y` values but differing `x` values, which would highlight whether the function correctly identifies the leftmost point based on `x`.

Additional Test Cases to Detect Mutations

To identify potential issues arising from these mutations, we can add the following test cases:

1. **Detect Mutation 1**:
    - **Test Case**: `[(2, 5), (2, 5), (3, 5)]`
    - **Expected Result**: The leftmost minimum point should still be `(2, 5)`, despite having duplicate points.
    - **Purpose**: This will check if the mutation that changes `p.get(i).x < p.get(min).x` to `p.get(i).x <= p.get(min).x` allows duplicates to incorrectly be selected as the leftmost point.
2. **Detect Mutation 2**:
    - **Test Case**: `[(1, 3), (0, 3), (3, 2)]`
    - **Expected Result**: The function should select `(3, 2)` as the minimum point based on the y-coordinate.
    - **Purpose**: This case verifies that if the comparison is altered to `<=`, it does not mistakenly overwrite the minimum point selection.
3. **Detect Mutation 3**:
    - **Test Case**: `[(2, 1), (1, 1), (0, 1)]`
    - **Expected Result**: The leftmost point should be `(0, 1)`.
    - **Purpose**: This checks that removing the x-coordinate condition in the second loop does not lead to an incorrect point being chosen as the leftmost.
4. **Detect Mutation 4**:
    - **Test Case**: `[(1, 1), (1, 2), (2, 1)]`

- **Expected Result**: The function should select `(1, 1)` as the leftmost minimum point.
- **Purpose**: This case verifies that if the y-coordinate comparison is altered to `<=`, the point `(1, 1)` is not incorrectly overwritten.

5. **Detect Mutation 5**:
   - **Test Case**: `[(4, 5), (4, 5), (4, 6)]`
   - **Expected Result**: The leftmost minimum should be `(4, 5)`, confirming that duplicates do not affect the selection.
   - **Purpose**: This tests the robustness of the function when faced with identical y-coordinates.
6. **Detect Mutation 6**:
   - **Test Case**: `[(5, 2), (5, 3), (2, 5)]`
   - **Expected Result**: The function should select `(2, 5)` as the minimum based on the y-coordinate.
   - **Purpose**: This will ensure that the alteration to `<=` does not affect the correct identification of the minimum point when there are ties in x-coordinates.

Python Code for Mutation:-

```
from math import atan2

class Point:
    def __init__(self, x, y):  # Corrected constructor name from init to
__init__
        self.x = x
        self.y = y

    def __repr__(self):  # Corrected method name from repr to __repr__
        return f"({self.x}, {self.y})"

def orientation(p, q, r):
    # Cross product to find orientation
    val = (q.y - p.y) * (r.x - q.x) - (q.x - p.x) * (r.y - q.y)
    if val == 0:
        return 0  # Collinear
    elif val > 0:
        return 1  # Clockwise
    else:
        return 2  # Counterclockwise

def distance_squared(p1, p2):
    return (p1.x - p2.x) ** 2 + (p1.y - p2.y) ** 2

def do_graham(points):
    # Step 1: Find the bottom-most point (or leftmost in case of a tie)
    n = len(points)
    min_y_index = 0
    for i in range(1, n):
        if (points[i].y < points[min_y_index].y) or \
           (points[i].y == points[min_y_index].y and points[i].x <
points[min_y_index].x):
            min_y_index = i

    points[0], points[min_y_index] = points[min_y_index], points[0]
    p0 = points[0]

    # Step 2: Sort the points based on polar angle with respect to p0
    points[1:] = sorted(points[1:], key=lambda p: (atan2(p.y - p0.y, p.x -
p0.x), distance_squared(p0, p)))

    # Step 3: Initialize the convex hull with the first three points
    hull = [points[0], points[1], points[2]]

    # Step 4: Process the remaining points
    for i in range(3, n):
        # Mutation introduced here: instead of checking `!= 2`, we incorrectly
use `== 1`
        while len(hull) > 1 and orientation(hull[-2], hull[-1], points[i]) ==
1:
            hull.pop()
        hull.append(points[i])

    return hull

# Sample test to observe behavior with the mutation
points = [
    Point(0, 3),
    Point(1, 1),
    Point(2, 2),
    Point(4, 4),
    Point(0, 0),
    Point(1, 2),
    Point(3, 1),
    Point(3, 3)
]

hull = do_graham(points)
print("Convex Hull:", hull)
```

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: No Iterations of the Loop

- **Input:** An empty vector `p`.
- **Test:** `Vector<Point> p = new Vector<Point>();`
- **Expected Result:** The method should terminate immediately without performing any operations. This scenario verifies the behavior of the method when the vector size is zero, ensuring it handles this edge case correctly.

Test Case 2: One Iteration of the Loop

- **Input:** A vector containing a single point.
- **Test:** `Vector<Point> p = new Vector<Point>(); p.add(new Point(0, 0));`
- **Expected Result:** The method should skip the loop entirely since `p.size()` is 1. It should simply swap the first point with itself, resulting in no change to the vector. This test case validates the handling of a scenario with a single element.

Test Case 3: Two Iterations of the Loop

- **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 Result:** The method should enter the loop to compare the two points. It will identify that the second point has a lower y-coordinate. Consequently, the index for the minimum y-coordinate will be updated, and a swap will position the point with coordinates (0, 0) at the start of the vector.

Test Case 4: Multiple Iterations of the Loop

- **Input:** A vector containing 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 Result:** The loop should process all three points. It will find that the second point has the lowest y-coordinate, updating the index for the minimum y-coordinate. After the swap, the point with coordinates (1, 0) will be positioned at the front of the vector.

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.

Ans. To achieve adequate coverage, the minimum number of test cases required for each coverage criterion is outlined below:

- **Statement Coverage:**
  Minimum required test cases: **3**
  - These test cases ensure that every line of code is executed at least once.
- **Branch Coverage:**
  Minimum required test cases: **3**
  - These test cases ensure that every branch (true/false) from each decision point is tested.
- **Basic Condition Coverage:**
  Minimum required test cases: **2**
  - These test cases ensure that each basic condition in decision points is tested independently for both true and false outcomes.
- **Path Coverage:**
  Minimum required test cases: **3**
  - These test cases ensure that every distinct path through the method's logic is executed at least once.

Summary of Minimum Test Cases:

- Total: **3 (Statement) + 3 (Branch) + 2 (Basic Condition) + 3 (Path) = 11 test cases**

3. 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.

To identify faults that are not detected by the test cases derived in Step 2, we can perform the following mutations on the `do_graham` function:

a. Deleting Code

- **Mutation:** Remove the entire loop that processes the remaining points.
- **Code Modification:**

```
python
Copy code
```

```
# Remove the following loop entirely
for i in range(3, n):
    while len(hull) > 1 and orientation(hull[-2], hull[-1],
points[i]) == 1:
        hull.pop()
    hull.append(points[i])
```

- **Effect:** The function will return only the first three points, ignoring the rest. This will go undetected by the existing test cases as they might not account for the complete processing of points.

b. Inserting Code

- **Mutation:** Insert an additional line that unnecessarily modifies the `hull` before returning it.
- **Code Modification:**

```
python
Copy code
hull.append(Point(999, 999))  # Insert this line before returning the
hull
```

- **Effect:** This mutation adds an arbitrary point to the hull, which can alter the expected output. Existing test cases may not account for this unexpected point being included.

c. Modifying Code

- **Mutation:** Change the condition in the while loop to incorrectly evaluate the orientation.
- **Code Modification:**

```
python
Copy code
# Change this line from:
while len(hull) > 1 and orientation(hull[-2], hull[-1], points[i]) ==
1:

# To this:
while len(hull) > 1 and orientation(hull[-2], hull[-1], points[i]) ==
2:
```

- **Effect:** This will cause the function to incorrectly remove points from the hull based on a different orientation check, which can lead to unexpected results. The test cases may not trigger this fault if they don't specifically test the scenario where orientation evaluates to counterclockwise.

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

To ensure all paths through the code are tested, the following test cases can be derived:

Test Case 1: Single Point (No Loop Execution)

- **Input:** A vector with one point.
- **Test:** `points = [Point(1, 1)]`

- **Expected Result:** The method should not enter the loop, returning the single point as the convex hull.

Test Case 2: Two Points (Single Iteration)

- **Input:** A vector with two points where the second point has a lower y-coordinate.
- **Test:** `points = [Point(2, 2), Point(1, 1)]`
- **Expected Result:** The method should identify `(1, 1)` as the point with the lowest y-coordinate after a single loop iteration.

Test Case 3: Three Points (Basic Convex Hull)

- **Input:** A vector with three points forming a triangle.
- **Test:** `points = [Point(0, 0), Point(1, 2), Point(2, 0)]`
- **Expected Result:** The method should return all three points as they form a convex shape.

Test Case 4: Multiple Points with Clear Convex Hull

- **Input:** A vector with multiple points where some points are clearly inside.
- **Test:** `points = [Point(0, 0), Point(2, 2), Point(1, 1), Point(2, 0), Point(3, 1)]`
- **Expected Result:** The method should return the points that form the convex hull.

Test Case 5: Complex Configuration

- **Input:** A vector with various points, some collinear.
- **Test:** `points = [Point(0, 0), Point(1, 1), Point(2, 2), Point(3, 0), Point(0, 3)]`
- **Expected Result:** The method should return the outer points of the convex shape, ignoring collinear points in between.

These test cases will ensure that all distinct paths through the `do_graham` method are executed, verifying the algorithm's behavior under various conditions.