

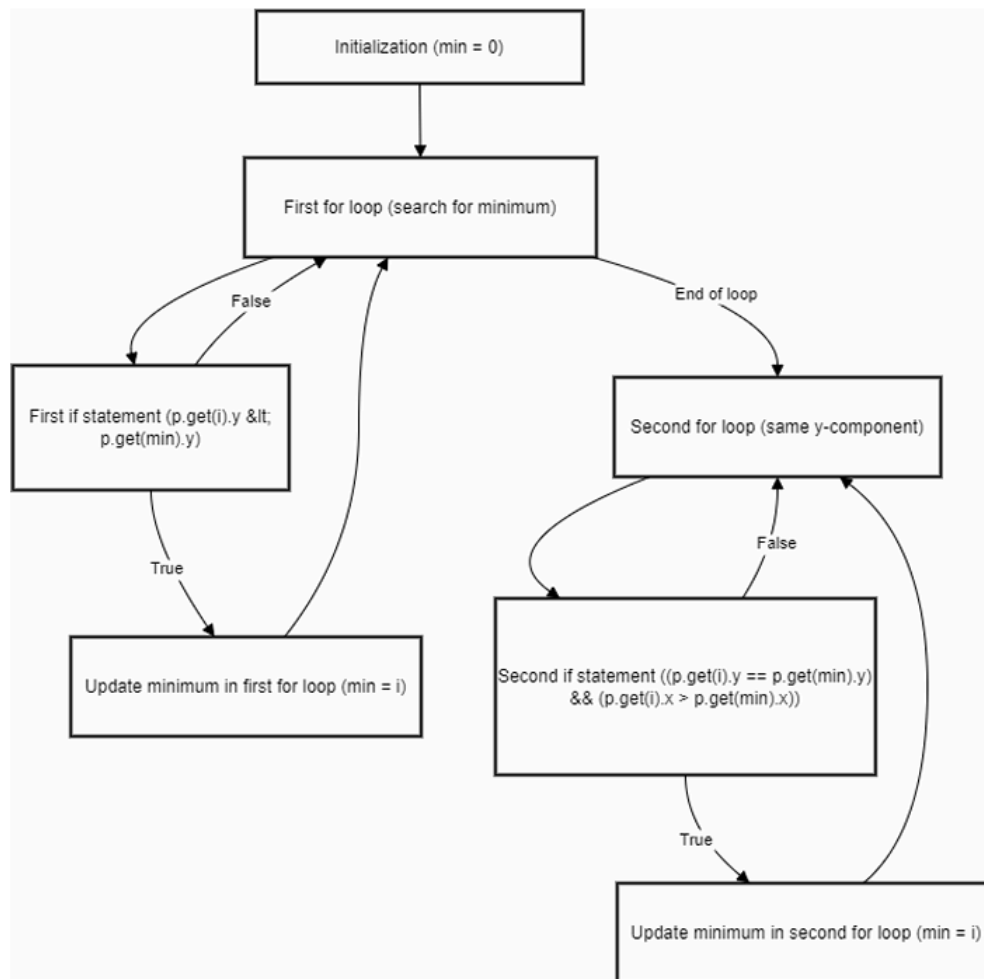
Lab09

Mutation Testing

Ramya Shah

202201409

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:-** Statement coverage requires that each statement in the code is executed at least once. We need to ensure that all lines (1 to 7 in the code) are executed in at least one of the test cases.

Test Set for Statement Coverage:

- TestCase1: Input vector with only one point (e.g., [(0, 0)])
- TestCase2: Input vector with two points, one lower y value (e.g., [(1, 1), (2, 0)])
- TestCase3: Input vector with points having the same y value but different x values (e.g., [(1, 1), (2, 1), (3, 1)])

- b. **Branch Coverage:-** Branch coverage requires that each branch in the code (True/False paths for each decision) is taken at least once.

Test Set for Branch Coverage:

- TestCase1: Input vector with only one point (e.g., [(0, 0)])
 - True branch: The first loop exits immediately, covering the loop without changes.
- TestCase2: Input vector with two points, one lower y value (e.g., [(1, 1), (2, 0)])
 - True branch: The minimum is updated to the second point.
- TestCase3: Input vector with points having the same y but different x values (e.g., [(1, 1), (3, 1), (2, 1)])
 - True branch for the second condition.
- TestCase4: Input vector with all points having the same y value (e.g., [(1, 1), (1, 1), (1, 1)])
 - False branch: Ensure the second loop is executed without changing the minimum.

- c. **Basic Condition Coverage:-** Basic Condition Coverage requires that each individual condition within every decision is evaluated as both True and False at least once.

Test Set for Basic Condition Coverage:

- TestCase1: Input vector with only one point (e.g., [(0, 0)])
 - Covers the first condition $p.get(i).y < p.get(min).y$ as false since no comparisons can be made.
- TestCase2: Input vector with two points with the second point having a lower y value (e.g., [(1, 1), (2, 0)])
 - Covers the first condition true and the second condition false.
- TestCase3: Input vector with points that have the same y but different x values (e.g., [(1, 1), (3, 1), (2, 1)])
 - Covers the second condition as true.
- TestCase4: Input vector with points that have the same y and x values (e.g., [(1, 1), (1, 1), (1, 1)])
 - Covers both conditions as false

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.

```
[*] Start mutation process:
  - targets: point
  - tests: test_points
[*] 4 tests passed:
  - test_points [0.36220 s]
[*] Start mutants generation and execution:
  - [# 1] COI point:
-----
6:
7: def find_min_point(points):
8:     min_index = 0
9:     for i in range(1, len(points)):
- 10:         if points[i].y < points[min_index].y:
+ 10:         if not (points[i].y < points[min_index].y):
11:             min_index = i
12:     for i in range(len(points)):
13:         if (points[i].y == points[min_index].y and points[i].x > points[min_index].x):
14:             min_index = i
-----
[0.23355 s] killed by test_points.py::TestFindMinPoint::test_multiple_points_with_ties
  - [# 2] COI point:
-----
0:     for i in range(1, len(points)):
-----
[0.23355 s] killed by test_points.py::TestFindMinPoint::test_multiple_points_with_ties
  - [# 2] COI point:
-----
9:     for i in range(1, len(points)):
10:         if points[i].y < points[min_index].y:
11:             min_index = i
12:     for i in range(len(points)):
- 13:         if (points[i].y == points[min_index].y and points[i].x > points[min_index].x):
+ 13:         if not ((points[i].y == points[min_index].y and points[i].x > points[min_index].x))
14:             min_index = i
15:     return points[min_index]
-----
[0.27441 s] killed by test_points.py::TestFindMinPoint::test_multiple_points_with_same_y
  - [# 3] LCR point:
-----
9:     for i in range(1, len(points)):
10:         if points[i].y < points[min_index].y:
11:             min_index = i
12:     for i in range(len(points)):
- 13:         if (points[i].y == points[min_index].y and points[i].x > points[min_index].x):
+ 13:         if (points[i].y == points[min_index].y or points[i].x > points[min_index].x):
14:             min_index = i
15:     return points[min_index]
```

```
-----
[0.18323 s] survived
- [# 6] ROR point:
-----
    9:     for i in range(1, len(points)):
    10:         if points[i].y < points[min_index].y:
    11:             min_index = i
    12:     for i in range(len(points)):
- 13:         if (points[i].y == points[min_index].y and points[i].x > points[min_index].x):
+ 13:         if (points[i].y != points[min_index].y and points[i].x > points[min_index].x):
    14:             min_index = i
    15:     return points[min_index]
-----

[0.18059 s] killed by test_points.py::TestFindMinPoint::test_multiple_points_with_same_y
- [# 7] ROR point:
-----
    9:     for i in range(1, len(points)):
    10:         if points[i].y < points[min_index].y:
    11:             min_index = i
    12:     for i in range(len(points)):
- 13:         if (points[i].y == points[min_index].y and points[i].x > points[min_index].x):
+ 13:         if (points[i].y == points[min_index].y and points[i].x < points[min_index].x):
    14:             min_index = i
    15:     return points[min_index]
-----

[0.13933 s] killed by test_points.py::TestFindMinPoint::test_multiple_points_with_same_y
- [# 8] ROR point:
-----
    9:     for i in range(1, len(points)):
    10:         if points[i].y < points[min_index].y:
    11:             min_index = i
    12:     for i in range(len(points)):
- 13:         if (points[i].y == points[min_index].y and points[i].x > points[min_index].x):
+ 13:         if (points[i].y == points[min_index].y and points[i].x >= points[min_index].x):
    14:             min_index = i
    15:     return points[min_index]
-----

[0.11494 s] survived
[*] Mutation score [2.22089 s]: 75.0%
- all: 8
- killed: 6 (75.0%)
- survived: 2 (25.0%)
- incompetent: 0 (0.0%)
- timeout: 0 (0.0%)
-----
```

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	Input	Description	Expected Output
Test Case 1	An empty vector p	Ensures no iterations of either loop occur.	Handle gracefully (e.g., return an empty result or a specific value indicating no points).
Test Case 2	$[(3, 4)]$	Ensures the first loop runs exactly once (the minimum point is the only point).	Return the only point in p : $(3, 4)$.
Test Case 3	$[(1, 2), (3, 2)]$	Ensures the first loop finds the minimum point, and the second loop runs once to compare x-coordinates.	Return the point with the maximum x-coordinate: $(3, 2)$.
Test Case 4	$[(3, 1), (2, 2), (5, 1)]$	Ensures the first loop finds the minimum y-coordinate and continues to the second loop.	Return $(5, 1)$, as it has the maximum x-coordinate among points with the same y.
Test Case 5	$[(1, 1), (4, 1), (3, 2)]$	Ensures the first loop finds $(1, 1)$, and the second loop runs twice to check other points with $y = 1$.	Return $(4, 1)$, as it has the maximum x-coordinate.

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 code using the aforementioned criteria.

Test Case	Expected Output	Input Vector p	Description
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.	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.	Test case [(3, 4)] will pass without processing correctly.
Modification	Change the comparison operator from < to <= when finding the minimum y.	Test cases like [(1, 1), (1, 1), (1, 1)] might pass while still failing in logic.

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

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