IT-314: SOFTWARE ENGINEERING

LAB-9: Mutation Testing

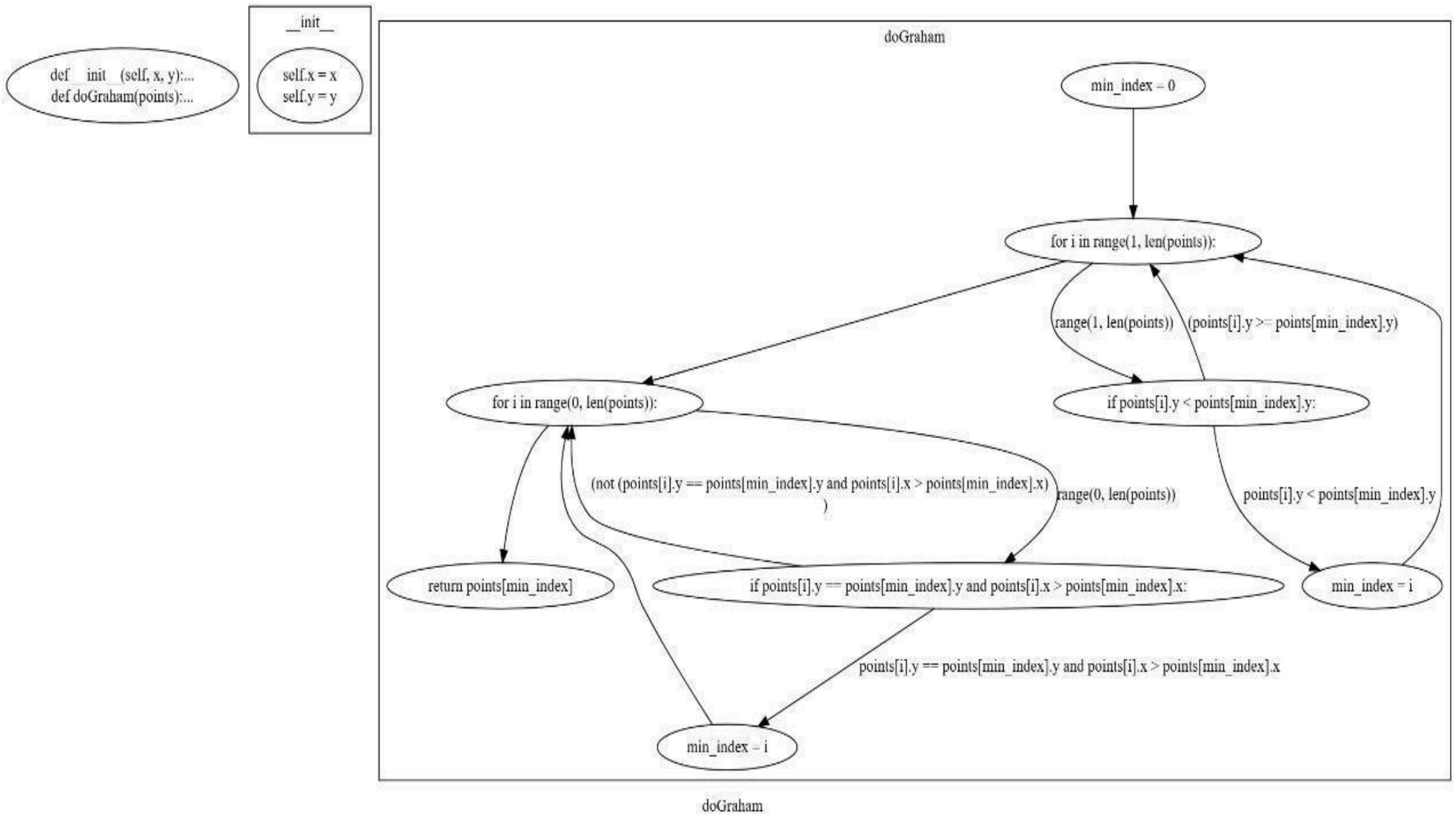
Name: Pritish N Desai

Student Id: 202201312

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.

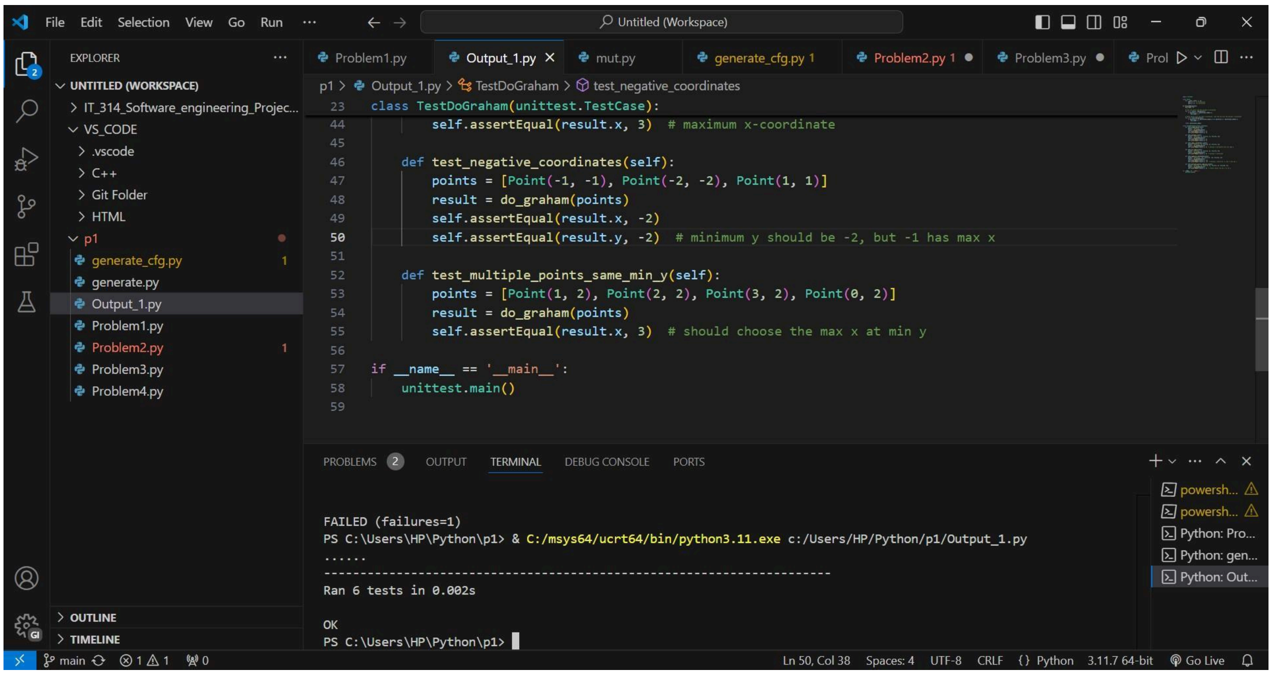
```
Vector doGraham(Vector p) {
        int i,j,min,M;

        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;
            }
        }
```

1. The Control Flow graph for the code is as follows:



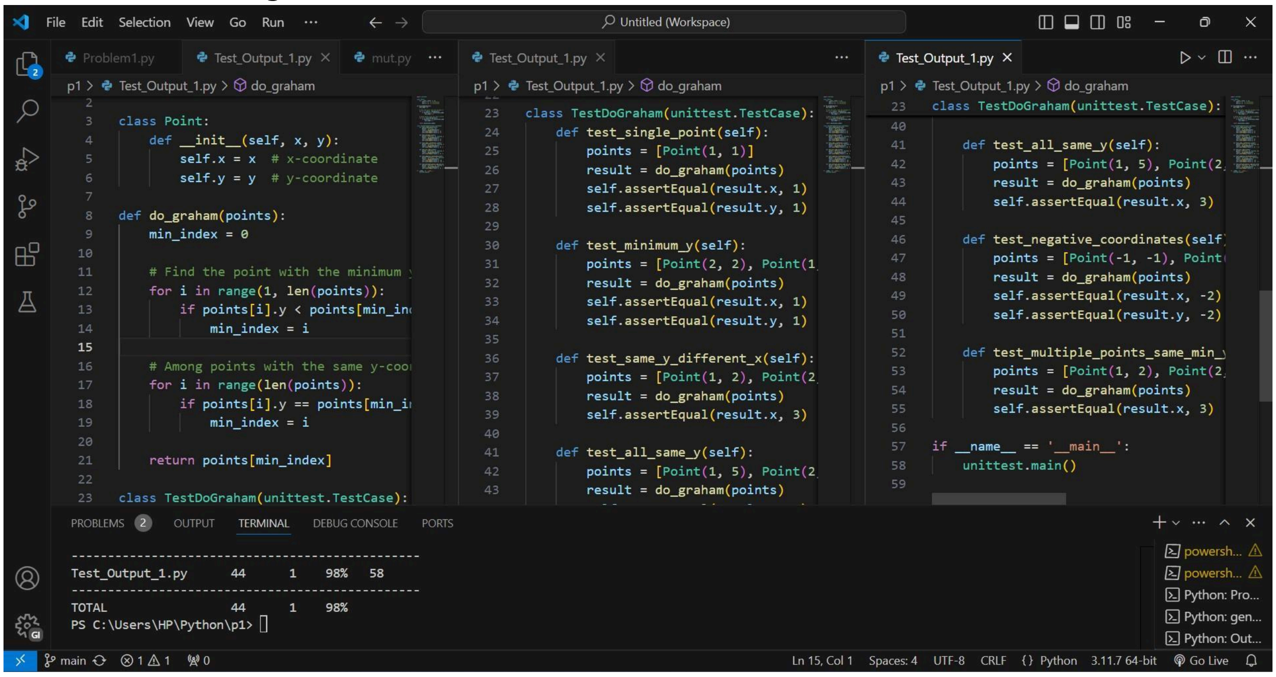
def __init__(self, x, y):...
def doGraham(points):...
__init__
self.x = x
self.y = y
doGraham
min_index = 0
for i in range(1, len(points)):
range(1, len(points))
(points[i].y >= points[min_index].y)
for i in range(0, len(points)):
if points[i].y < points[min_index].y:
(not (points[i].y == points[min_index].y and points[i].x > points[min_index].x))
range(0, len(points))
points[i].y < points[min_index].y
return points[min_index]
if points[i].y == points[min_index].y and points[i].x > points[min_index].x:
min_index = i
points[i].y == points[min_index].y and points[i].x > points[min_index].x
min_index = i
doGraham

2. Unit Testing for the code is as follows:


```python
class TestDoGraham(unittest.TestCase):
        self.assertEqual(result.x, 3)  # maximum x-coordinate

    def test_negative_coordinates(self):
        points = [Point(-1, -1), Point(-2, -2), Point(1, 1)]
        result = do_graham(points)
        self.assertEqual(result.x, -2)
        self.assertEqual(result.y, -2)  # minimum y should be -2, but -1 has max x

    def test_multiple_points_same_min_y(self):
        points = [Point(1, 2), Point(2, 2), Point(3, 2), Point(0, 2)]
        result = do_graham(points)
        self.assertEqual(result.x, 3)  # should choose the max x at min y

if __name__ == '__main__':
    unittest.main()
```

```
FAILED (failures=1)
PS C:\Users\HP\Python\p1> & C:/msys64/ucrt64/bin/python3.11.exe c:/Users/HP/Python/p1/Output_1.py
......
----------------------------------------------------------------------
Ran 6 tests in 0.002s

OK
PS C:\Users\HP\Python\p1>
```

- Code Coverage for the code as follows:

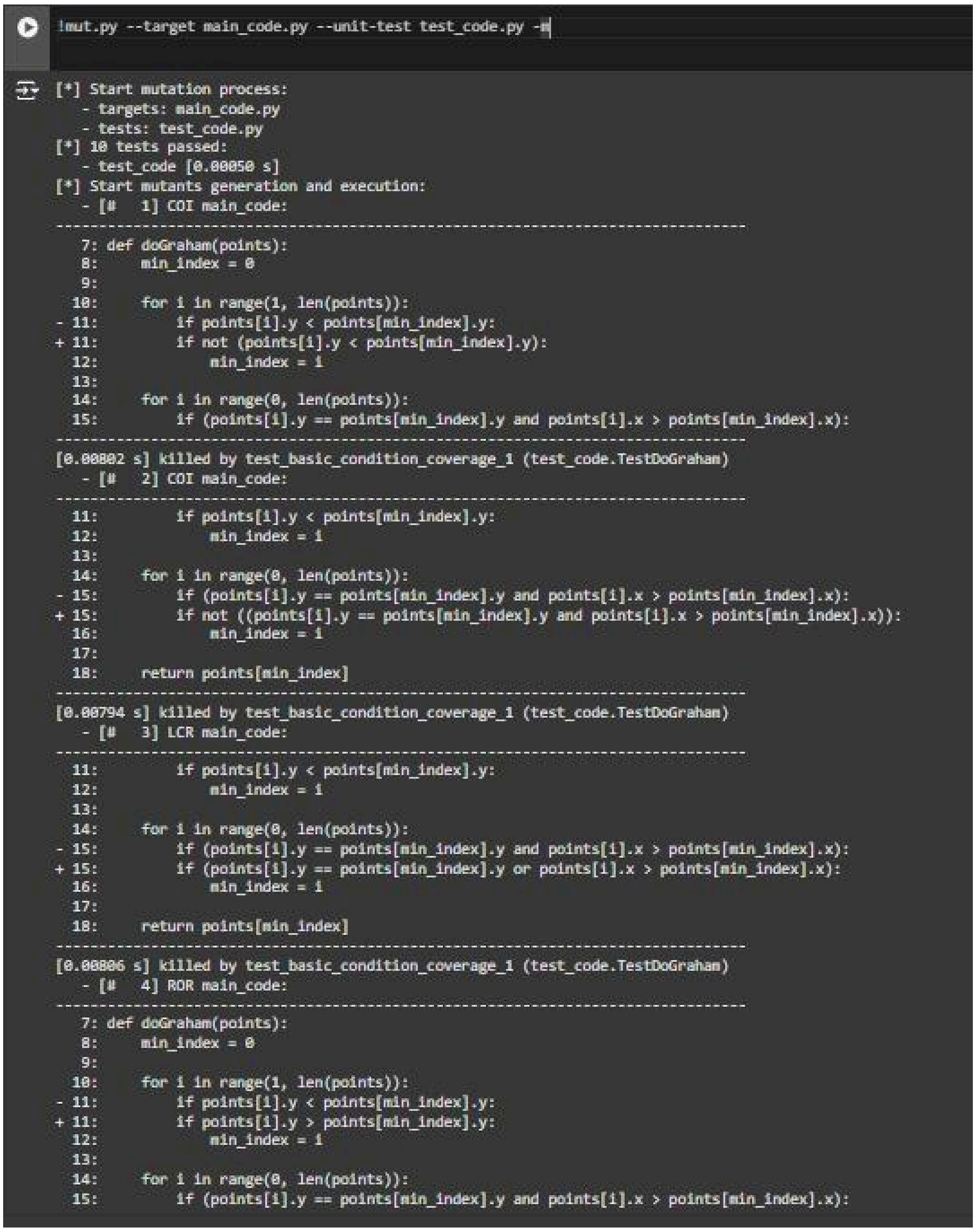

```python

class Point:
    def __init__(self, x, y):
        self.x = x  # x-coordinate
        self.y = y  # y-coordinate

def do_graham(points):
    min_index = 0

    # Find the point with the minimum y
    for i in range(1, len(points)):
        if points[i].y < points[min_inc
            min_index = i

    # Among points with the same y-coo
    for i in range(len(points)):
        if points[i].y == points[min_in
            min_index = i

    return points[min_index]

class TestDoGraham(unittest.TestCase):
```

```python
class TestDoGraham(unittest.TestCase):
    def test_single_point(self):
        points = [Point(1, 1)]
        result = do_graham(points)
        self.assertEqual(result.x, 1)
        self.assertEqual(result.y, 1)

    def test_minimum_y(self):
        points = [Point(2, 2), Point(1
        result = do_graham(points)
        self.assertEqual(result.x, 1)
        self.assertEqual(result.y, 1)

    def test_same_y_different_x(self):
        points = [Point(1, 2), Point(2
        result = do_graham(points)
        self.assertEqual(result.x, 3)

    def test_all_same_y(self):
        points = [Point(1, 5), Point(2
        result = do_graham(points)
```

```python
class TestDoGraham(unittest.TestCase):

    def test_all_same_y(self):
        points = [Point(1, 5), Point(2
        result = do_graham(points)
        self.assertEqual(result.x, 3)

    def test_negative_coordinates(self
        points = [Point(-1, -1), Point
        result = do_graham(points)
        self.assertEqual(result.x, -2)
        self.assertEqual(result.y, -2)

    def test_multiple_points_same_min_
        points = [Point(1, 2), Point(2
        result = do_graham(points)
        self.assertEqual(result.x, 3)

if __name__ == '__main__':
    unittest.main()
```

```
-----------------------------------------------
Test_Output_1.py      44      1     98%    58
-----------------------------------------------
TOTAL                 44      1     98%
PS C:\Users\HP\Python\p1>
```

3. The Mutation tool output of the code is as follows:



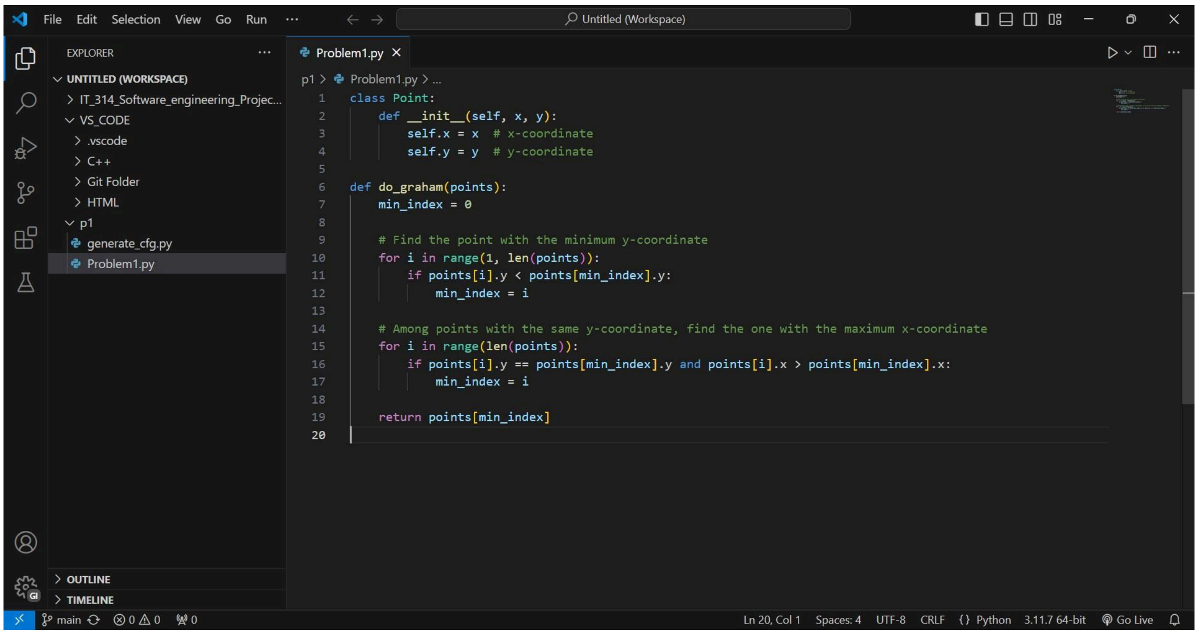
```
!mut.py --target main_code.py --unit-test test_code.py -m
```

```
[*] Start mutation process:
   - targets: main_code.py
   - tests: test_code.py
[*] 10 tests passed:
   - test_code [0.00050 s]
[*] Start mutants generation and execution:
   - [#   1] COI main_code:
--------------------------------------------------------------------------------
   7: def doGraham(points):
   8:     min_index = 0
   9: 
  10:     for i in range(1, len(points)):
- 11:         if points[i].y < points[min_index].y:
+ 11:         if not (points[i].y < points[min_index].y):
  12:             min_index = i
  13: 
  14:     for i in range(0, len(points)):
  15:         if (points[i].y == points[min_index].y and points[i].x > points[min_index].x):
--------------------------------------------------------------------------------
[0.00802 s] killed by test_basic_condition_coverage_1 (test_code.TestDoGraham)
   - [#   2] COI main_code:
--------------------------------------------------------------------------------
  11:         if points[i].y < points[min_index].y:
  12:             min_index = i
  13: 
  14:     for i in range(0, len(points)):
- 15:         if (points[i].y == points[min_index].y and points[i].x > points[min_index].x):
+ 15:         if not ((points[i].y == points[min_index].y and points[i].x > points[min_index].x)):
  16:             min_index = i
  17: 
  18:     return points[min_index]
--------------------------------------------------------------------------------
[0.00794 s] killed by test_basic_condition_coverage_1 (test_code.TestDoGraham)
   - [#   3] LCR main_code:
--------------------------------------------------------------------------------
  11:         if points[i].y < points[min_index].y:
  12:             min_index = i
  13: 
  14:     for i in range(0, len(points)):
- 15:         if (points[i].y == points[min_index].y and points[i].x > points[min_index].x):
+ 15:         if (points[i].y == points[min_index].y or points[i].x > points[min_index].x):
  16:             min_index = i
  17: 
  18:     return points[min_index]
--------------------------------------------------------------------------------
[0.00806 s] killed by test_basic_condition_coverage_1 (test_code.TestDoGraham)
   - [#   4] ROR main_code:
--------------------------------------------------------------------------------
   7: def doGraham(points):
   8:     min_index = 0
   9: 
  10:     for i in range(1, len(points)):
- 11:         if points[i].y < points[min_index].y:
+ 11:         if points[i].y > points[min_index].y:
  12:             min_index = i
  13: 
  14:     for i in range(0, len(points)):
  15:         if (points[i].y == points[min_index].y and points[i].x > points[min_index].x):
```

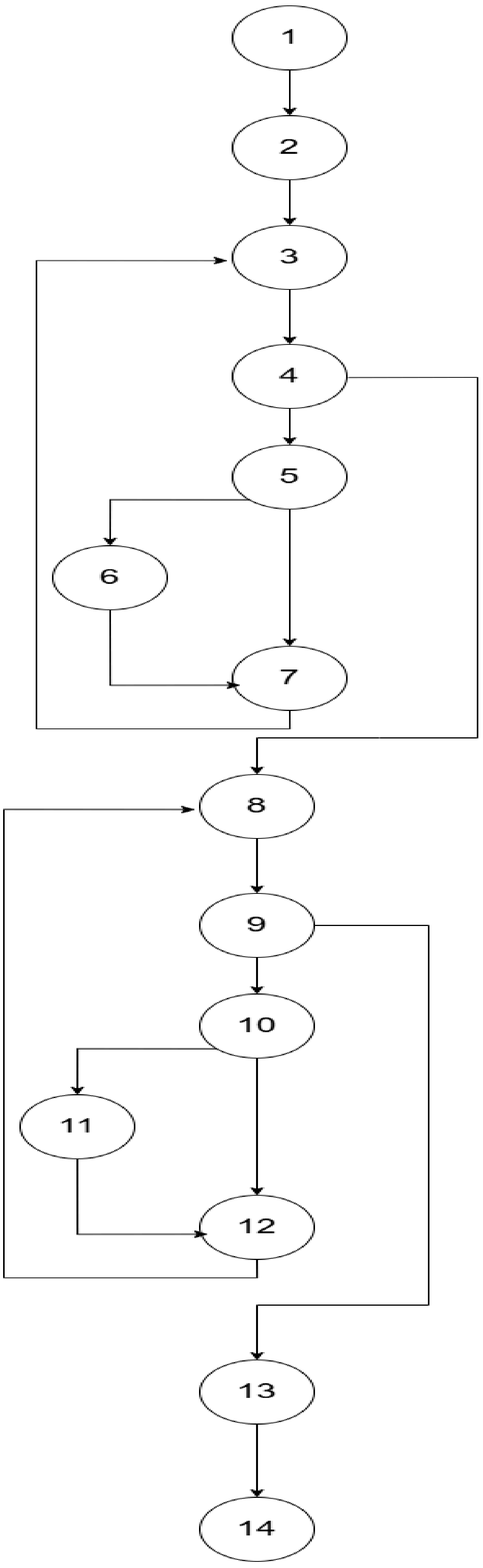
❖ <u>Exercise:</u>

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).

For the given pseudocode of vector doGraham code the required code in python is as follows:



```python
class Point:
    def __init__(self, x, y):
        self.x = x  # x-coordinate
        self.y = y  # y-coordinate

def do_graham(points):
    min_index = 0

    # Find the point with the minimum y-coordinate
    for i in range(1, len(points)):
        if points[i].y < points[min_index].y:
            min_index = i

    # Among points with the same y-coordinate, find the one with the maximum x-coordinate
    for i in range(len(points)):
        if points[i].y == points[min_index].y and points[i].x > points[min_index].x:
            min_index = i

    return points[min_index]

```

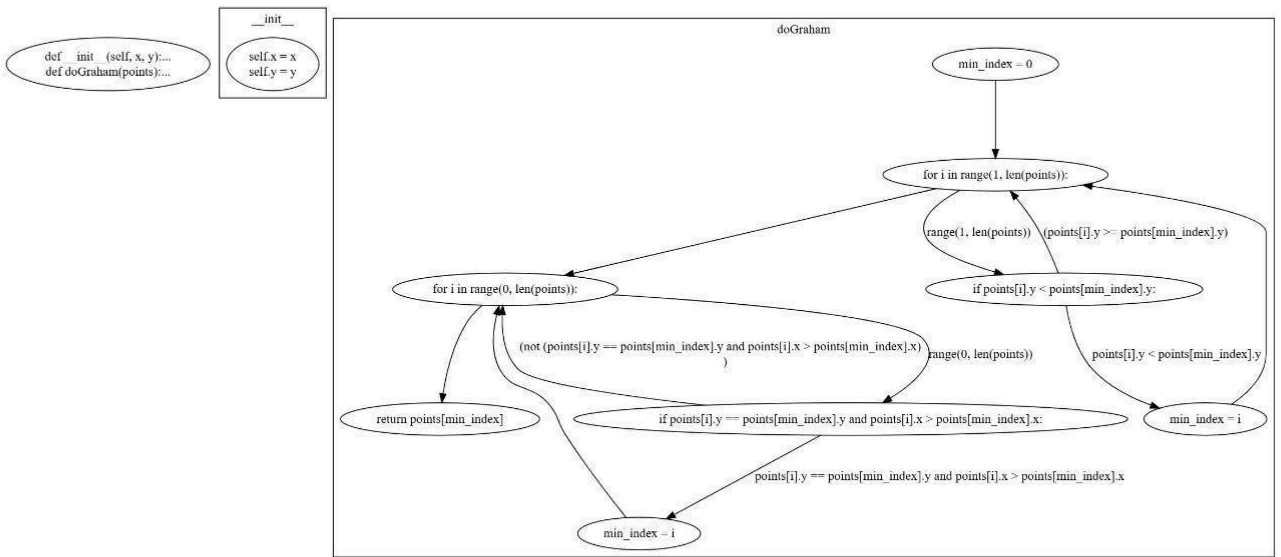

1
2
3
4
5
6
7
8
9
10
11
12
13
14

The CFG of this code fragment represents the following nodes:

- Node 1: Start of function
- Node 2: Initialising the value of min_index=0
- Node 3: Start of for loop with i=1
- Node 4: Condition check for i < points.size
- Node 5: Condition for minimum y-component
- Node 6: Updating the value of min_index=i
- Node 7: Increment of for loop
- Node 8: Start of for loop with i=1
- Node 9: Condition check for i < points.size
- Node 10: Condition for index of y-component equals to min_index and minimum x-component
- Node 11: Updating the value of min_index=i
- Node 12: Increment of for loop
- Node 13: Adding the required min_index to vector and returning the result
- Node 14: Closing the function

Flow of Nodes:

- Node 1 → Node 2 (Start of function)
- Node 2 → Node 3 (Starting of for loop)
- Node 3 → Node 4 (Checking the loop condition)
- Node 4 → Node 5 (value of i satisfies the condition)
- Node 4 → Node 8 (value of i doesn't satisfies the condition)
- Node 5 → Node 6 (if condition satisfies)
- Node 5 → Node 7 (if condition does not satisfies)
- Node 7 → Node 3 (increment the i variable)
- Node 8 → Node 9 (Checking the loop condition)
- Node 9 → Node 10 (value of j satisfies the condition
- Node 9 → Node 13 (value of j doesn't satisfies the condition)
- Node 10 → Node 11 (if condition satisfies)
- Node 10 → Node 12 (if condition does not satisfies)
- Node 12 → Node 9 (increment the i variable)
- Node 13 → Node 14 (returning the result and ending the function)The resultant CFG generated tool for python for above code as follows:


def __init__(self, x, y):...
def doGraham(points):...
__init__
self.x = x
self.y = y
doGraham
min_index = 0
for i in range(1, len(points)):
range(1, len(points))
(points[i].y >= points[min_index].y)
if points[i].y < points[min_index].y:
points[i].y < points[min_index].y
min_index = i
for i in range(0, len(points)):
(not (points[i].y == points[min_index].y and points[i].x > points[min_index].x))
range(0, len(points))
return points[min_index]
if points[i].y == points[min_index].y and points[i].x > points[min_index].x:
points[i].y == points[min_index].y and points[i].x > points[min_index].x
min_index = i

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

1. Statement Coverage

- To cover all statements: We need at least one test case that iterates through both loops and executes all statements.

Test Case 1:

- Points: [(5, 5), (4, 3), (7, 5), (6, 1)]
    - Expected Output: Point (6, 1)

2.Branch Coverage

- To satisfy branch coverage, we need: Cases where each if condition evaluates to both true and false.

Test Case 2:

- Points: [(2, 2), (3, 1), (4, 1), (5, 2)]
    - Expected Output: Point (4, 1) (as it has the lowest y-coordinate with the highest x-coordinate among ties)

3.Basic Condition Coverage

- Basic Condition Coverage requires that each individual condition evaluates to both true and false:

❖ points[i].y < points[min_index].y
❖ points[i].y == points[min_index].y
❖ points[i].x > points[min_index].x

Test Case 1 and Test Case 2 cover all basic conditions, so no additional test cases are needed here.

❖ Path Coverage

- For path coverage, each loop should be iterated 0, 1, and 2 times, so we need additional cases:
- Test Case 3: Empty list []
    - Expected Output: None (or handle with an exception if necessary)

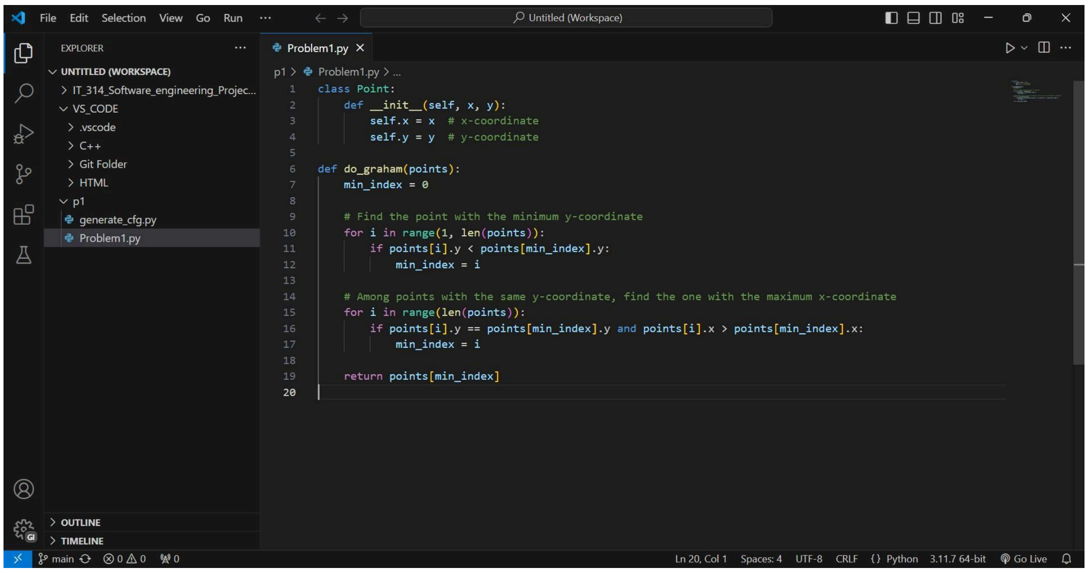
- Test Case 4: Single point list [(1, 1)]
    - Expected Output: Point (1, 1)
- Test Case 5: Two points [(2, 5), (3, 3)]
    - Expected Output: Point (3, 3)

Final Test Set

Test Case	Point	Coverage Criteria	Expected Output
1	[(5, 5), (4, 3), (7, 5), (6, 1)]	Statement, Branch, Basic Condition	Point (6, 1)
2	[(2, 2), (3, 1), (4, 1), (5, 2)]	Branch,Basic Condition	Point (4, 1)
3	[]	Path (0 iterations)	Noneor Exception
4	[(1, 1)]	Path (1 iterations)	Point (1, 1)
5	[(2, 5), (3, 3)]	Path (2 iteration)	Point (3, 3)

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 operations separately, i.e., by deleting the code, by inserting the code, by modifying the code.

The original reference of the code:

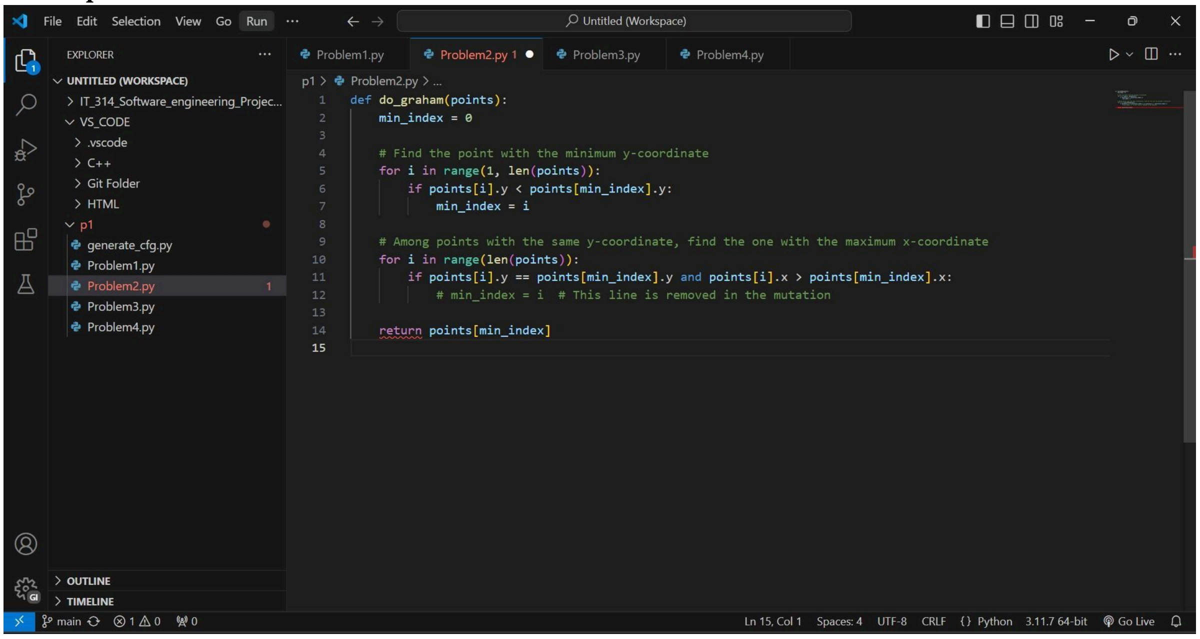


```python
class Point:
    def __init__(self, x, y):
        self.x = x  # x-coordinate
        self.y = y  # y-coordinate

def do_graham(points):
    min_index = 0

    # Find the point with the minimum y-coordinate
    for i in range(1, len(points)):
        if points[i].y < points[min_index].y:
            min_index = i

    # Among points with the same y-coordinate, find the one with the maximum x-coordinate
    for i in range(len(points)):
        if points[i].y == points[min_index].y and points[i].x > points[min_index].x:
            min_index = i

    return points[min_index]
```

Mutations:

1. Deletion Mutation:

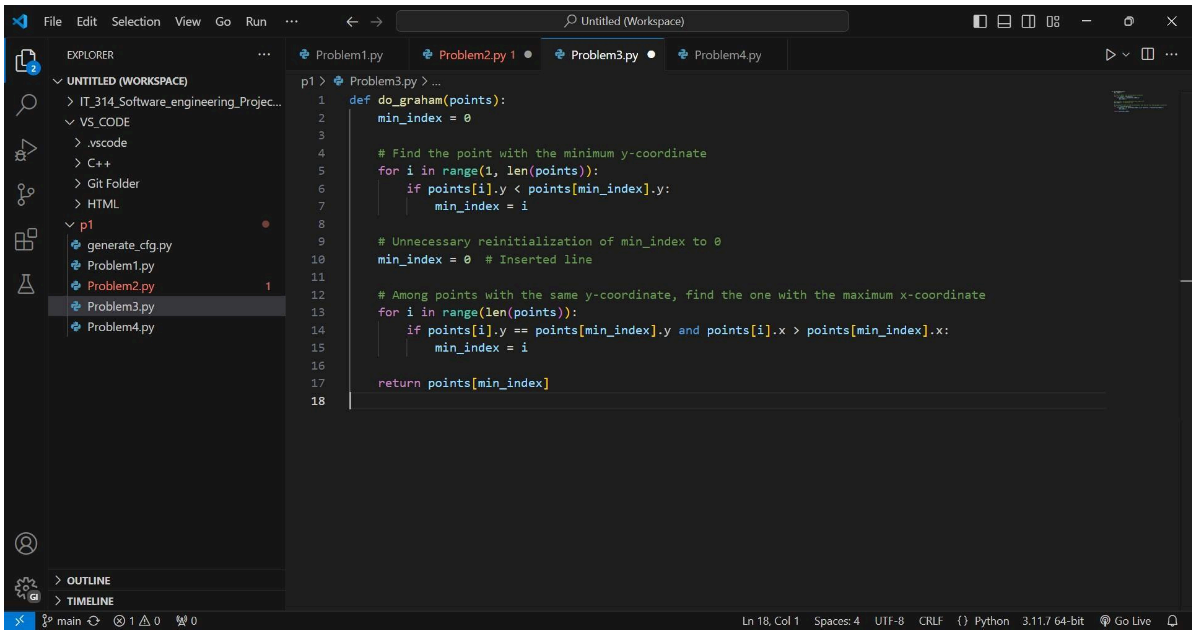
- In this Mutation, we remove a line that plays a significant role in the logic but in a way that might not immediately trigger a test failure with our current test cases.
- Example: Remove min_index = i in the second if statement.



```
def do_graham(points):
    min_index = 0

    # Find the point with the minimum y-coordinate
    for i in range(1, len(points)):
        if points[i].y < points[min_index].y:
            min_index = i

    # Among points with the same y-coordinate, find the one with the maximum x-coordinate
    for i in range(len(points)):
        if points[i].y == points[min_index].y and points[i].x > points[min_index].x:
            # min_index = i  # This line is removed in the mutation

    return points[min_index]
```

2. Insertion Mutation:

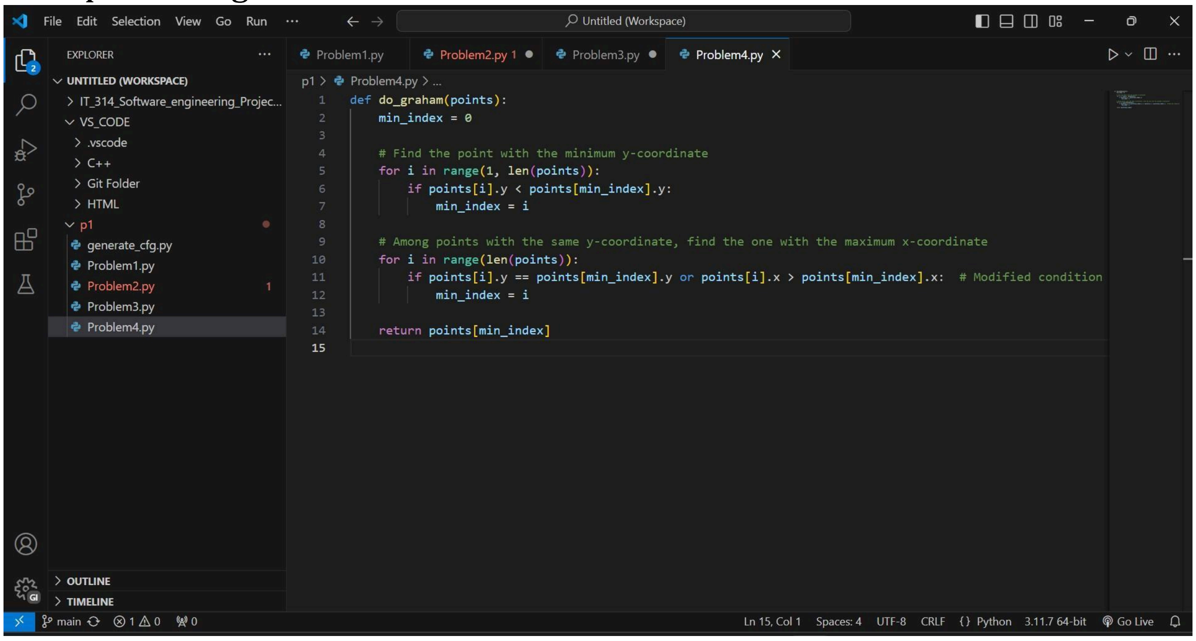
- In this Mutation, we add an unnecessary line of code that should logically have no effect, but under certain conditions, it may introduce subtle errors.
- Example: Insert an extra min_index = 0 after the first for loop.



```python
def do_graham(points):
    min_index = 0

    # Find the point with the minimum y-coordinate
    for i in range(1, len(points)):
        if points[i].y < points[min_index].y:
            min_index = i

    # Unnecessary reinitialization of min_index to 0
    min_index = 0  # Inserted line

    # Among points with the same y-coordinate, find the one with the maximum x-coordinate
    for i in range(len(points)):
        if points[i].y == points[min_index].y and points[i].x > points[min_index].x:
            min_index = i

    return points[min_index]
```

1. Modified Mutation:

- A modification mutation involves changing an operator or a condition in a way that alters the behaviour of the code.
- Example: Change the condition in the second if statement from and to or.



```python
def do_graham(points):
    min_index = 0

    # Find the point with the minimum y-coordinate
    for i in range(1, len(points)):
        if points[i].y < points[min_index].y:
            min_index = i

    # Among points with the same y-coordinate, find the one with the maximum x-coordinate
    for i in range(len(points)):
        if points[i].y == points[min_index].y or points[i].x > points[min_index].x:  # Modified condition
            min_index = i

    return points[min_index]
```

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

To achieve Path Coverage for the do_graham method, we need to ensure that all possible paths through the control flow graph (CFG) are exercised. This includes testing paths where loops are iterated zero, one, and two times. Below, we will identify and describe each path with corresponding test cases

Paths in the do_graham Method

1. Path 1: Empty list ([])
   - This path handles the case where there are no points.
   - Expected Output: None or handle with an exception if necessary.
2. Path 2: Single point list ([(1, 1)])
   - This path covers the case where there is only one point, which should directly return that point.
   - Expected Output: Point (1, 1).
3. Path 3: Two points with distinct y-coordinates ([(2, 2), (3, 1)])
   - The first loop will iterate once, determining the minimum y-coordinate.
   - Expected Output: Point (3, 1) (the point with the minimum y-coordinate).
4. Path 4: Two points with the same y-coordinate, differing x-coordinates ([(1, 2), (3, 2)])
   - The first loop will identify the first point as minimum, and the second loop will check both points.
   - Expected Output: Point (3, 2) (the point with the maximum x-coordinate among those with the same y-coordinate).
5. Path 5: Multiple points with varying y-coordinates ([(5, 5), (4, 3), (7, 5), (6, 1)])
   - Tests the function's ability to find the minimum y-coordinate among multiple points.
   - Expected Output: Point (6, 1).
6. Path 6: Multiple points with same minimum y-coordinate but differing x-coordinates ([(2, 3), (3, 3), (1, 2), (2, 2)])
   - Ensures that the function selects the point with the maximum x-coordinate among those with the same minimum y-coordinate.
   - Expected Output: Point (3, 3) (the point with the maximum x-coordinate among points with y-coordinate 3).

Summary of Test Cases for Path Coverag

Test Case	Point	Path Coverage Description	Expected Output
1	[]	Path 1: No points	None or Exception
2	[(1, 1)]	Path 2: Single point	Point (1, 1)
3	[(2, 2), (3, 1)]	Path3:Twodistinct points with different y-coordinates	Point (3, 1)
4	[(1, 2), (3, 2)]	Path 4: Two points withsame y-coordinate	Point (3, 1)
5	[(5, 5), (4, 3), (7, 5), (6, 1)]	Path 5: Multiple points withvarying y-coordinates	Point (3, 2)
6	[(2, 3), (3, 3), (1, 2), (2, 2)]	Path 6: Multiple points with same minimum y-coordinate	Point (6, 1)