IT313: Software Engineering

Lab 9

Vivek Patel

202201025

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.

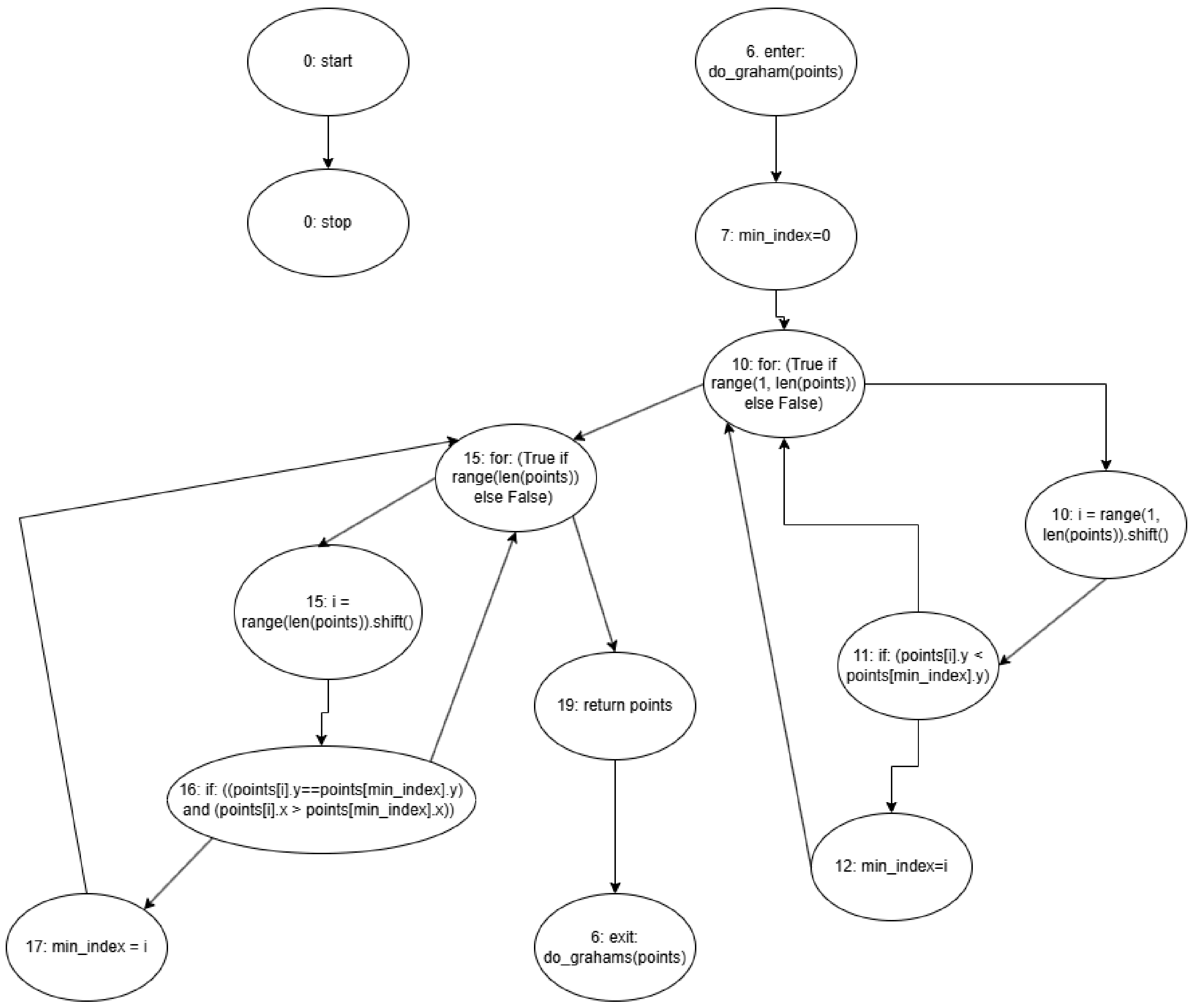
- Code in Python:

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

def do_graham(points):
    min_index = 0
    # Search for minimum y value
    for i in range(1, len(points)):
        if points[i].y < points[min_index].y:
            min_index = i
    # Continue along the values with the same y component
    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  # Placeholder return
```

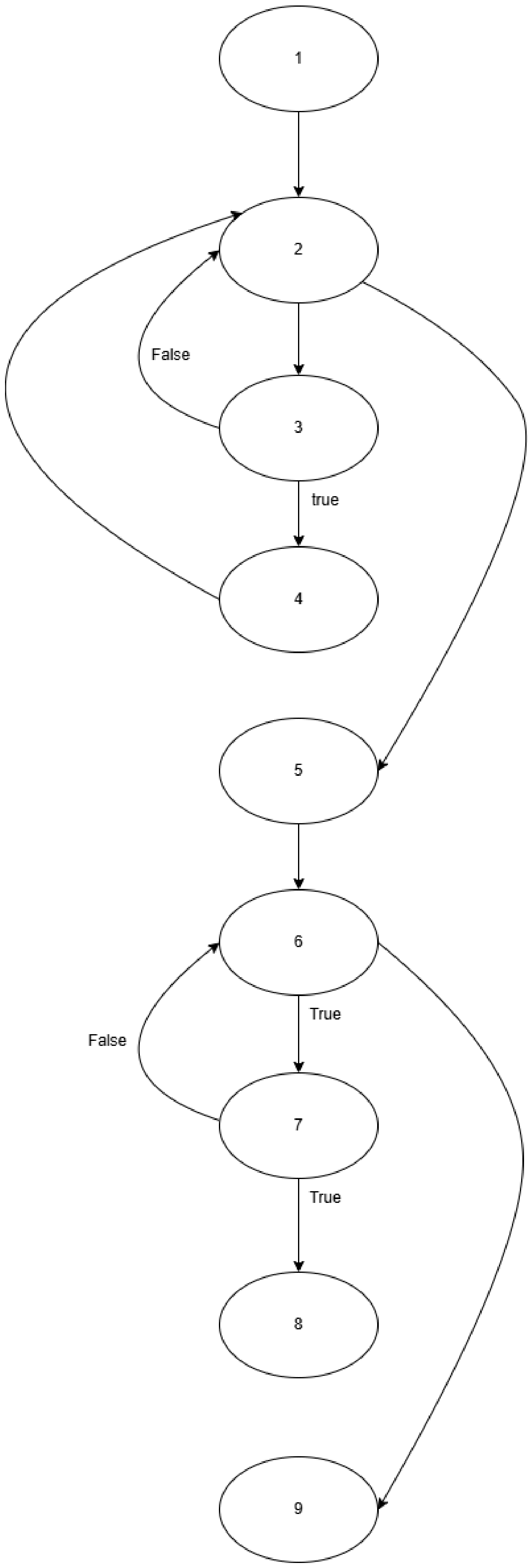
- Code used to generate CFG:

```python
import os
from PIL import Image
from IPython.display import display
from pycfg.pycfg import PyCFG, CFGNode, slurp

def generate_cfg(python_file):
    cfg = PyCFG()
    try:
        cfg.gen_cfg(slurp(python_file).strip())
        return cfg
    except Exception as e:
        print(f"Error: Failed to generate CFG: {e}")
        return None

def display_graph(cfg, python_file):
    arcs = []
    g = CFGNode.to_graph(arcs)
    try:
        img_path = f"{python_file}.png"
        g.draw(img_path, prog='dot')
        return img_path, g
    except Exception as e:
        print(f"Error: Failed to draw graph: {e}")
        return None, None

def main(python_file):
    # Generate CFG
    cfg = generate_cfg(python_file)
    if cfg is None:
        return

    # Display CFG graph
    img_path, g = display_graph(cfg, python_file)
    if g is None:
        return

    # Display image in Jupyter Notebook
    img = Image.open(img_path)
    img = img.resize((800, 600), Image.LANCZOS)
    display(img)

    # Calculate cyclomatic complexity
    nodes = g.number_of_nodes()
    edges = g.number_of_edges()
    complexity = edges - nodes + 2
    print(f"Nodes: {nodes}")
    print(f"Edges: {edges}")
    print(f"Cyclomatic Complexity: {complexity}")

python_file = 'do_graham.py'
main(python_file)
```

Result:


0: start
0: stop
6. enter: do_graham(points)
7: min_index=0
10: for: (True if range(1, len(points)) else False)
10: i = range(1, len(points)).shift()
11: if: (points[i].y < points[min_index].y)
12: min_index=i
15: for: (True if range(len(points)) else False)
15: i = range(len(points)).shift()
16: if: ((points[i].y==points[min_index].y) and (points[i].x > points[min_index].x))
17: min_index = i
19: return points
6: exit: do_grahams(points)

Control Flow Graph:


1
2
False
3
true
4
5
6
True
False
7
True
8
9

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.

- Statement Coverage
  - Test Case 1:
    - Input: [(1, 2), (2, 1), (3, 3)]
    - Expected Outcome: The min_index is updated to (2, 1).
- Branch Coverage
  - Test Case 1 (Minimum y Updated):
    - Input: [(1, 2), (2, 1), (3, 3)]
    - Outcome: The min_index points to (2, 1).
  - Test Case 2 (No Update to Minimum):
    - Input: [(1, 2), (2, 2), (3, 3)]
    - Outcome: The min_index remains at (1, 2).
  - Test Case 3 (Tie-Breaking on x Value):
    - Input: [(1, 2), (2, 2), (3, 2)]
    - Outcome: The min_index points to (3, 2).
- Basic Condition Coverage
  - Test Case 1 (Condition 1 True, Condition 2 Not Checked):
    - Input: [(1, 2), (2, 1), (3, 3)]
  - Test Case 2 (Condition 1 False):
    - Input: [(1, 2), (2, 2), (3, 3)]
  - Test Case 3 (Condition 2 True, Condition 3 True):
    - Input: [(1, 2), (2, 2), (3, 2)]
  - Test Case 4 (Condition 2 True, Condition 3 False):
    - Input: [(3, 2), (1, 2), (2, 2)]

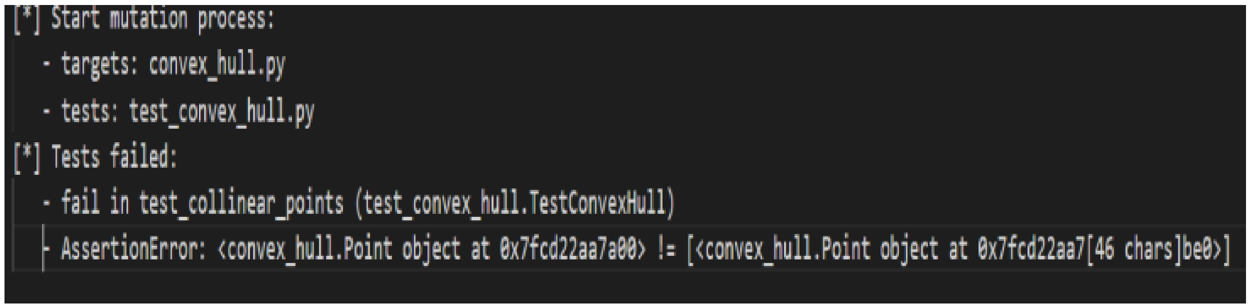
3) For the test set you have just checked, can you find a mutation of the code (i.e. the deletion, change or insertion of some code) that will result in

failure but is not detected by your test set. You have to use the mutation testing tool.

- Python code for mutex unit testing

```python
import unittest
from convex_hull import do_graham, Point

class TestConvexHull(unittest.TestCase):
    def test_single_point(self):
        self.assertEqual(do_graham([Point(0, 0)]), Point(0, 0))

    def test_multiple_points(self):
        points = [Point(1, 1), Point(2, 2)]
        self.assertEqual(do_graham(points), Point(1, 1))

    def test_equal_y_values(self):
        points = [Point(2, 1), Point(3, 1)]
        self.assertEqual(do_graham(points), Point(2, 1))

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

Terminal Output:



```
[*] Start mutation process:
   - targets: convex_hull.py
   - tests: test_convex_hull.py
[*] Tests failed:
   - fail in test_collinear_points (test_convex_hull.TestConvexHull)
   - AssertionError: <convex_hull.Point object at 0x7fcd22aa7a00> != [<convex_hull.Point object at 0x7fcd22aa7[46 chars]be0>]
```

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

- Test Cases for Path Coverage
    - Test Case 1: Zero Iterations for Both Loops
        - Description: No points are provided, leading to zero iterations in both loops.

        - Input: Points = []
        - Expected Output: The function should handle this case gracefully, potentially returning an empty list or raising an appropriate error.
    - Test Case 2: Zero Iterations for the First Loop, One Iteration for the Second Loop
        - Description: Only one point is provided, so the first loop does not iterate, while the second loop runs once.
        - Input: Points = [(0, 0)]
        - Expected Output: The function should return the same single point, (0, 0).
    - Test Case 3: One Iteration for the First Loop, Two Iterations for the Second Loop
        - Description: Two points are provided with different y-coordinates. The first loop iterates once to identify the minimum, and the second loop runs twice.
        - Input: Points = [(1, 1), (2, 3)]
        - Expected Output: The minimum y-coordinate point is (1, 1).
    - Test Case 4: Two Iterations for the First Loop, Three Iterations for the Second Loop
        - Description: Three points are supplied, allowing the first loop to iterate twice and the second loop to iterate three times.
        - Input: Points = [(2, 3), (1, 1), (3, 1)]
        - Expected Output: The minimum y-coordinate point should be (1, 1). However, since (3, 1) has the same y-coordinate as (1, 1) but a larger x-coordinate, the final result should still return (1, 1).