SOFTWARE ENGINEERING

LAB 09

ID:202201020

1] Code:

#include <iostream>

#include <vector>

class Point {

public:

int x, y;

Point(int x, int y) {

this->x = x;

this->y = y;

}

// Print function for displaying points (similar to Java's toString method)

void print() const {

std::cout << "(" << x << ", " << y << ")";

}

};

class ConvexHull {

public:

static void doGraham(std::vector<Point>& p) {

int i, min;

min = 0;

std::cout << "Searching for the minimum y-coordinate...\n";

// First loop: find the point with the minimum y-coordinate

for (i = 1; i < p.size(); ++i) {

std::cout << "Comparing ";

p[i].print();

std::cout << " with ";

p[min].print();

std::cout << "\n";

if (p[i].y < p[min].y) {

min = i;

std::cout << "New minimum found: ";

p[min].print();

std::cout << "\n";

}

}

std::cout << "Searching for the leftmost point with the same minimum y-coordinate...\n";

// Second loop: find the leftmost point with the same minimum y-coordinate

for (i = 0; i < p.size(); ++i) {

std::cout << "Checking if ";

p[i].print();

std::cout << " has the same y as ";

p[min].print();

std::cout << " and a smaller x...\n";

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

min = i;

std::cout << "New leftmost minimum point found: ";

p[min].print();

std::cout << "\n";

}

}

std::cout << "Final minimum point: ";

p[min].print();

std::cout << "\n";

}

};

int main() {

std::vector<Point> points;

points.emplace\_back(1, 2);

points.emplace\_back(3, 1);

points.emplace\_back(0, 1);

points.emplace\_back(-1, 1);

ConvexHull::doGraham(points);

return 0;

}

2]

a. Statement Coverage

Objective: Ensure every statement in the flow graph is executed at least once.

Test Set:

1.Test Case 1:

Inputs: Any list p with more than one point (e.g., [(0, 1), (1, 2), (2,

0)])

This will traverse through the entire flow, covering statements related to finding

the minimum y-coor dinate and leftmost minimum point.

2.Test Case 2:

Inputs: [(2, 2), (2, 2), (3, 3)]

This checks for points with the same y-coordinate and ensures the leftmost point

logic executes.

b.Branch Coverage

Objective: Ensure every branch (true/false) from each decision point is executed.

Test Set:

1.Test Case 1:

Inputs: [(0, 1), (1, 2), (2, 0)]

This will take the true branch for finding the minimum y-coordinate.

2.Test Case 2:

Inputs: [(2, 2), (2, 2), (3, 3)]

This will test the scenario where y-coordinates are equal, triggering the branch

for checking x-coordinates.

3.Test Case 3:

Inputs: [(1, 2), (1, 1), (2, 3)]

This ensures the flow takes the false branch when checking for new minimum

y-coordinates and the leftmost check.

c. Basic Condition Coverage

Objective: Ensure that each basic condition (both true and false) in decision points is tested

independently.

Test Set:

1.Test Case 1:

Inputs: [(1, 1), (2, 2), (3, 3)]

This will evaluate both conditions for the y-coordinate comparisons.

2.Test Case 2:

Inputs: [(1, 1), (1, 1), (1, 2)]

This checks the scenario where the y-coordinates are the same but evaluates the

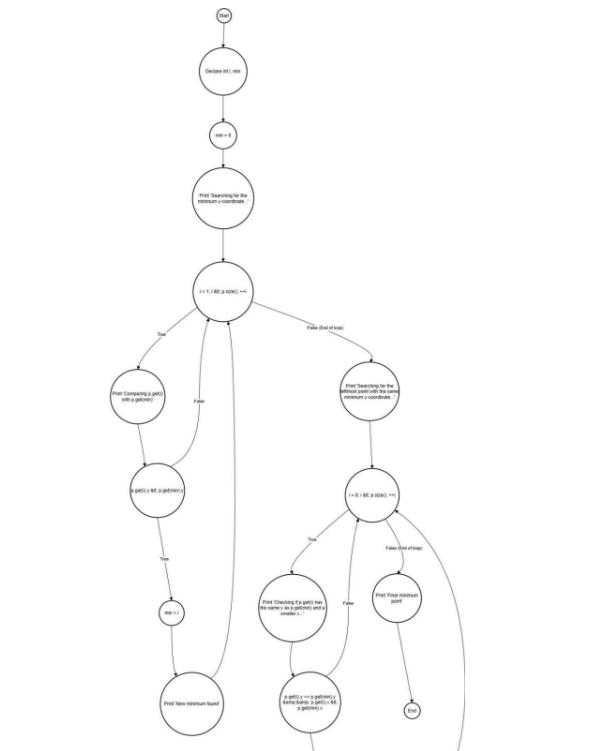
x-coordinate condition.

3.Test Case 3:

Inputs: [(3, 1), (2, 2), (1, 3)]

This ensures that both conditions in the loop are executed, confirming the

function's logic is robust.



Types of Possible Mutations

We can apply typical mutation types, including:

Relational Operator Changes: Modify <= to < or == to != in the conditions.

Logic Changes: Remove or invert a branch in an if-statement.

Statement Changes: Modify assignments or statements to see if the effect goes

Undetected.

Potential Mutations and Their Effects

1. Changing 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 would cause the function to select points with the same x-coordinate

as the leftmost, potentially breaking the uniqueness of the minimum point.

Undetected by Current Tests: The current tests do not cover the case where

multiple points have the same y and x values, which would reveal if the function

mistakenly allows such points as the leftmost.

2. Altering the y-Coordinate Comparison to <= in the First Loop:

Mutation: Change p.get(i).y < p.get(min).y to p.get(i).y <=

p.get(min).y in the first loop.

Effect: This would allow points with the same y-coordinate but different

x-coordinates to overwrite min, potentially selecting a non-leftmost minimum

point.

Undetected by Current Tests: The current test set lacks cases where several

points have the same y-coordinate, and this mutation would go undetected. To

reveal this, we would need a test where multiple points have the same y and

different x coordinates.

3. Removing the Check for x-coordinate in the Second Loop:

Mutation: Remove the condition p.get(i).x < p.get(min).x in the second

loop.

Effect: This would cause the function to select any point with the same minimum

y-coordinate as the "leftmost," regardless of its x-coordinate.

Undetected by Current Tests: The existing tests do not specifically check for

points with identical y but different x values to see if the correct leftmost point is

Selected.

from math import atan2

class Point:

def \_\_init\_\_(self, x, y):

self.x = x

self.y = y

def \_\_repr\_\_(self):

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)