Name: Muhammad Maaz

Roll No: 22PWBCS0933

SECTION: B

SUBJECT: AI

LAB TASK: 3

The task requires solving the N-Puzzle problem using the A* search algorithm, and also implementing the same problem with the Hill Climbing algorithm. Let's break down the solution for this task.

N-Puzzle Problem Overview:

1. **Initial State**:

```
2 4 6
5 8
1 7 3
```

2. **Goal State**:

```
1 2 3
4 5 6
7 8
```

A* Search Algorithm for N-Puzzle:

The A* algorithm uses two costs to evaluate each state:

- **g(n)**: The cost to reach the current node (how many moves it took).
- **h(n)**: The heuristic, which estimates the cost from the current state to the goal state.
    - A common heuristic for N-puzzle is the **Manhattan Distance** between the tiles and their goal positions.

The total cost **f(n)** is computed as:

`f(n) = g(n) + h(n)`

A* Algorithm Steps:

1. **Initialization**: Start from the initial state and enqueue it in a priority queue with priority 0.
2. **Main Loop**: Dequeue the state with the lowest `f(n)` and explore its successors.
3. **Termination**: When the goal state is dequeued, print the sequence of states.

Hill Climbing Algorithm:

Hill Climbing is a local search algorithm that iteratively moves to a neighboring state with a lower cost. It will stop when no neighboring state improves the current state.

Python Code for A* Search:

```
import heapq

# Define the initial state and goal state
initial_state = [[2, 4, 6], [5, 8, 0], [1, 7, 3]]  # 0 represents the empty space
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

# Function to calculate Manhattan Distance heuristic
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:  # Skip the empty space
                goal_i, goal_j = divmod(value - 1, 3)
```

```
                distance += abs(i - goal_i) + abs(j - goal_j)
    return distance

# Function to find the empty space
def find_empty(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

# Function to generate possible next states
def get_neighbors(state):
    neighbors = []
    x, y = find_empty(state)
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

    for dx, dy in moves:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y],
new_state[x][y]
            neighbors.append(new_state)

    return neighbors

# A* search algorithm
def a_star_search():
    open_list = []
    heapq.heappush(open_list, (manhattan_distance(initial_state), 0,
initial_state, []))  # (f, g, state, path)
    closed_list = set()

    while open_list:
        f, g, current_state, path = heapq.heappop(open_list)

        if current_state == goal_state:
            return path + [current_state]  # Return the solution path

        closed_list.add(tuple(map(tuple, current_state)))

        for neighbor in get_neighbors(current_state):
            if tuple(map(tuple, neighbor)) not in closed_list:
                h = manhattan_distance(neighbor)
                heapq.heappush(open_list, (g + 1 + h, g + 1, neighbor, path +
[current_state]))

    return None  # No solution found

# Main function
if __name__ == "__main__":
    solution_path = a_star_search()
    for step in solution_path:
        for row in step:
            print(row)
        print("-----")
```

Explanation:

1. **Manhattan Distance**: This heuristic measures the total number of moves each tile is away from its goal position.

2. **Priority Queue**: The algorithm uses a priority queue to expand the most promising states first, minimizing the combined cost (`f(n)`).
3. **State Transition**: For each state, we calculate the possible moves of the empty space (up, down, left, right) and generate the new states.
4. **Path Tracking**: The sequence of states from the initial state to the goal state is tracked and returned once the goal is reached.

Hill Climbing Algorithm:

To implement Hill Climbing, you would need to select the best neighboring state with the smallest heuristic and move towards it. The algorithm would stop when no neighboring state improves the current state.

Output:

The output will show the sequence of states leading from the initial configuration to the goal configuration. Each state will be printed as a 3x3 grid.

You can run this code to find the solution using the A* search algorithm. Let me know if you need any additional clarifications!