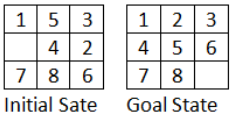
1. Write a python program to calculate the optimal solution using manhattan distance for the given 8 puzzle problem,



1	5	3
	4	2
7	8	6

Initial Sate

1	2	3
4	5	6
7	8	

Goal State

Program :

```
import heapq


class PuzzleNode:
    def __init__(self, state, parent=None, depth=0):
        self.state = state
        self.parent = parent
        self.depth = depth
        self.manhattan = self.calculate_manhattan_distance()

    def __lt__(self, other):
        return (self.depth + self.manhattan) < (other.depth + other.manhattan)

    def calculate_manhattan_distance(self):
        manhattan_distance = 0
        goal_positions = {1: (0, 0), 2: (0, 1), 3: (0, 2),
                          4: (1, 0), 5: (1, 1), 6: (1, 2),
                          7: (2, 0), 8: (2, 1), 0: (2, 2)}
```

```
        for i in range(3):
            for j in range(3):
                value = self.state[i][j]
                if value != 0:
                    goal_position = goal_positions[value]
                    manhattan_distance += abs(i - goal_position[0]) + abs(j - goal_position[1])

        return manhattan_distance

def get_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def get_neighbors(node):
    i, j = get_blank_position(node.state)
    neighbors = []

    for x, y in [(i-1, j), (i+1, j), (i, j-1), (i, j+1)]:
        if 0 <= x < 3 and 0 <= y < 3:
            new_state = [list(row) for row in node.state]
            new_state[i][j], new_state[x][y] = new_state[x][y], new_state[i][j]
            neighbors.append(PuzzleNode(new_state, node, node.depth + 1))
```

```
    return neighbors

def is_goal_state(state):
    return state == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

def solve_puzzle(initial_state):
    initial_node = PuzzleNode(initial_state)
    open_set = [initial_node]
    closed_set = set()

    while open_set:
        current_node = heapq.heappop(open_set)

        if is_goal_state(current_node.state):
            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            return path[::-1]

        closed_set.add(tuple(map(tuple, current_node.state)))

        for neighbor in get_neighbors(current_node):
```

```
            if tuple(map(tuple, neighbor.state)) not in closed_set:
                heapq.heappush(open_set, neighbor)

    return None  # No solution found

def print_puzzle(state):
    for row in state:
        print(row)

# Example usage:
initial_state = [
    [1, 5, 3],
    [4, 2, 0],
    [7, 8, 6]
]

solution_path = solve_puzzle(initial_state)

if solution_path:
    for step, state in enumerate(solution_path):
        print(f"Step {step + 1}:")
        print_puzzle(state)
        print()
else:
```

```
    print("No solution found.")
```

Output :

```
[1, 5, 3],
[4, 2, 0],
 [7, 8, 6]
```