LAB-3 - 8-Puzzle Game

Code: (Using DFS)

```
class PuzzleState:
    def __init__(self, board, empty_pos, moves=[]):
        self.board = board
        self.empty_pos = empty_pos
        self.moves = moves

    def is_goal(self):
        return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]

    def get_possible_moves(self):
        x, y = self.empty_pos
        moves = []
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            nx, ny = x + dx, y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_board = self.board[:]
                new_board[x * 3 + y], new_board[nx * 3 + ny] = new_board[nx * 3 + ny],
new_board[x * 3 + y]
                moves.append((new_board, (nx, ny)))
        return moves

def dfs(initial_state):
    stack, visited = [initial_state], set()
    while stack:
        current_state = stack.pop()
        if current_state.is_goal():
            return current_state.moves
        visited.add(tuple(current_state.board))
        for new_board, new_empty_pos in current_state.get_possible_moves():
            new_state = PuzzleState(new_board, new_empty_pos, current_state.moves +
[new_board])
            if tuple(new_board) not in visited:
                stack.append(new_state)
    return None
```

```
def print_matrix(board):
    for i in range(0, 9, 3):
        print(board[i:i + 3])
    print()

def main():
    initial_board = [1, 2, 3, 4, 0, 5, 7, 8, 6]
    empty_pos = initial_board.index(0)
    initial_state = PuzzleState(initial_board, (empty_pos // 3, empty_pos % 3))

    print("Initial state:")
    print_matrix(initial_board)

    solution = dfs(initial_state)
    if solution:
        print("Solution found:")
        for step in solution:
            print_matrix(step)
    else:
        print("No solution found.")

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

Output:

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

Solution found:
[1, 2, 3]
[4, 5, 0]
[7, 8, 6]

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

Code: (Using Manhattan Distance)

```
def manhattan_distance(state, goal_state):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_i = (state[i][j] - 1) // 3
                goal_j = (state[i][j] - 1) % 3
                distance += abs(i - goal_i) + abs(j - goal_j)
    return distance

def get_neighbors(state):
    i, j = next((i, j) for i in range(3) for j in range(3) if state[i][j] == 0)
    moves = [(i-1, j), (i+1, j), (i, j-1), (i, j+1)]
    return [swap(state, i, j, x, y) for x, y in moves if 0 <= x < 3 and 0 <= y < 3]

def swap(state, i1, j1, i2, j2):
    new_state = [row[:] for row in state]
    new_state[i1][j1], new_state[i2][j2] = new_state[i2][j2], new_state[i1][j1]
    return new_state

def dfs_with_manhattan(state, goal, visited=set()):
    if state == goal: return [state]
    visited.add(str(state))
    neighbors = sorted(get_neighbors(state), key=lambda x: manhattan_distance(x, goal))
    for neighbor in neighbors:
        if str(neighbor) not in visited:
            path = dfs_with_manhattan(neighbor, goal, visited)
            if path: return [state] + path
    return None

# Take user input for initial state
initial_state = [[int(x) for x in input(f"Enter row {i+1}: ").split()] for i in range(3)]
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

solution = dfs_with_manhattan(initial_state, goal_state)
if solution:
    print("Solution found:")
    for state in solution: print(*state, sep='\n', end='\n\n')
```

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

Output:

```
============== RESTART: /Users/vi
Enter row 1: 1 0 3
Enter row 2: 4 2 6
Enter row 3: 7 5 8
Solution found:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]

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

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

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