LAB 2:8 Puzzle Problem

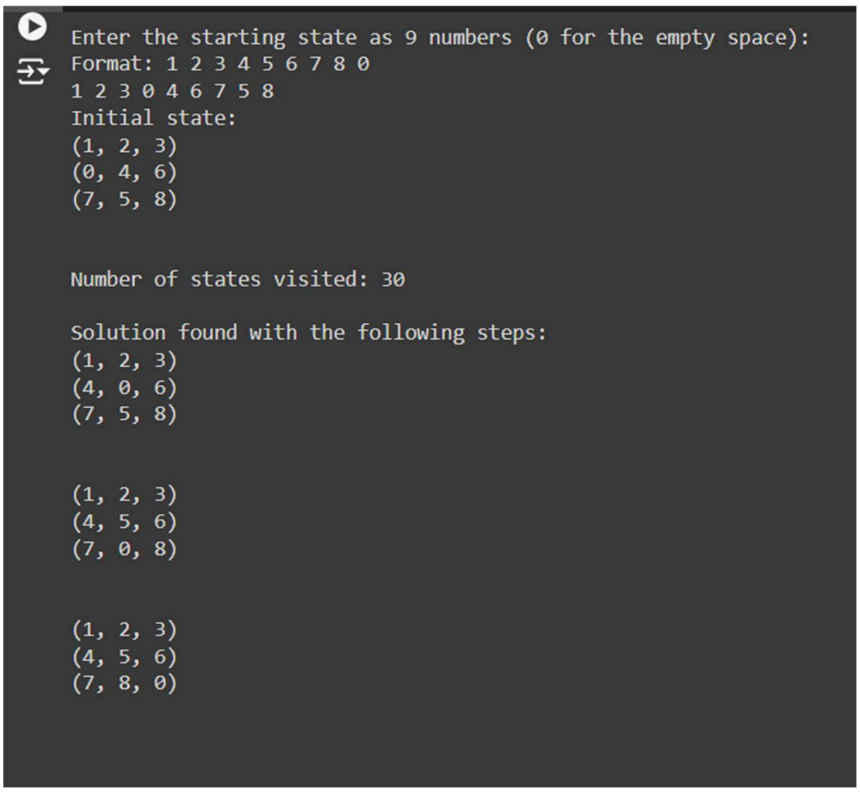
1.Using BFS

```
from collections import deque
GOAL_STATE = (1, 2, 3, 4, 5, 6, 7, 8, 0)

def find_empty(state):
    return state.index(0)

def get_neighbors(state):
    neighbors = []
    empty_index = find_empty(state)
    row, col = divmod(empty_index, 3)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for dr, dc in directions:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = list(state)
            new_state[empty_index], new_state[new_index] = new_state[new_index],
new_state[empty_index]
            neighbors.append(tuple(new_state))
    return neighbors

def bfs(initial_state):
    queue = deque([(initial_state, [])])
    visited = set()
    visited.add(initial_state)
    visited_count = 1  # Initialize visited count
```

```
    while queue:
        current_state, path = queue.popleft()

        if current_state == GOAL_STATE:
            return path, visited_count  # Return path and count

        for neighbor in get_neighbors(current_state):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [neighbor]))
                visited_count += 1  # Increment visited count
    return None, visited_count  # Return count if no solution found

def input_start_state():
    print("Enter the starting state as 9 numbers (0 for the empty space):")
    input_state = input("Format: 1 2 3 4 5 6 7 8 0\n")
    numbers = list(map(int, input_state.split()))

    if len(numbers) != 9 or set(numbers) != set(range(9)):
        print("Invalid input. Please enter numbers from 0 to 8 with no duplicates.")
        return input_start_state()
    return tuple(numbers)

def print_matrix(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
if __name__ == "__main__":
    initial_state = input_start_state()
    print("Initial state:")
    print_matrix(initial_state)
```

```
solution, visited_count = bfs(initial_state)

print(f"Number of states visited: {visited_count}")

if solution:
    print("\nSolution found with the following steps:")
    for step in solution:
        print_matrix(step)
    print()
else:
    print("No solution found.")
```

Output:



2.Using DFS

```
cnt = 0

def print_state(in_array):
    global cnt
    cnt += 1
    for row in in_array:
        print(' '.join(str(num) for num in row))
    print()  # Print a blank line for better readability

def helper(goal, in_array, row, col, vis):
    # Mark the current position as visited
    vis[row][col] = 1

    # Directions for row movements: up, right, down, left
    drow = [-1, 0, 1, 0]
    dcol = [0, 1, 0, -1]
    dchange = ['U', 'R', 'D', 'L']

    # Print the current state
    print("Current state:")
    print_state(in_array)

    # Check if the current state is the goal state
    if in_array == goal:
        print_state(in_array)
        print(f"Number of states visited: {cnt}")
        return True

    # Explore all possible directions
```

```
    for i in range(4):
        nrow = row + drow[i]
        ncol = col + dcol[i]

        # Check if the new position is within bounds and not visited
        if 0 <= nrow < len(in_array) and 0 <= ncol < len(in_array[0]) and not vis[nrow][ncol]:
            # Make the move (swap the empty space with the adjacent tile)
            print(f"Took a {dchange[i]} move")
            in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol], in_array[row][col]

            # Recursive call
            if helper(goal, in_array, nrow, ncol, vis):
                return True

            # Backtrack (undo the move)
            in_array[row][col], in_array[nrow][ncol] = in_array[nrow][ncol], in_array[row][col]

    # Mark the position as unvisited before returning
    vis[row][col] = 0
    return False

# Example usage
initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]  # 0 represents the empty space
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
visited = [[0] * 3 for _ in range(3)]  # 3x3 visited matrix
empty_row, empty_col = 1, 0  # Initial position of the empty space

# Run the helper function to find the solution
found_solution = helper(goal_state, initial_state, empty_row, empty_col, visited)
print("Solution found:", found_solution)
```

Output:

```
Took a L move
Current state:
1 2 3
4 6 8
0 7 5

Took a D move
Current state:
1 2 3
4 5 6
7 0 8

Took a R move
Current state:
1 2 3
4 5 6
7 8 0

1 2 3
4 5 6
7 8 0

Number of states visited: 42
Solution found: True
```