USN: 1BM22CS259

LAB 3: 8 PUZZLE PROBLEM

1) BFS

CODE:

```
from collections import deque


def solve_8_puzzle_bfs(initial_state):


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


    def get_possible_moves(state):
        moves = []
        row, col = find_blank_tile(state)


        if row > 0:  # Up
            new_state = [r[:] for r in state]
            new_state[row][col], new_state[row - 1][col] = new_state[row - 1][col],
new_state[row][col]
            moves.append((new_state, 'Up'))
        if row < 2:  # Down
            new_state = [r[:] for r in state]
            new_state[row][col], new_state[row + 1][col] = new_state[row + 1][col],
new_state[row][col]
            moves.append((new_state, 'Down'))
        if col > 0:  # Left
            new_state = [r[:] for r in state]
```

```
            new_state[row][col], new_state[row][col - 1] = new_state[row][col - 1],
new_state[row][col]
            moves.append((new_state, 'Left'))
        if col < 2:  # Right
            new_state = [r[:] for r in state]
            new_state[row][col], new_state[row][col + 1] = new_state[row][col + 1],
new_state[row][col]
            moves.append((new_state, 'Right'))

        return moves

    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    queue = deque([(initial_state, [])])  # (state, path)
    visited = set()
    visited.add(tuple(map(tuple, initial_state)))

    while queue:
        current_state, current_path = queue.popleft()

        if current_state == goal_state:
            return current_path + [(current_state, 'Goal')]  # Mark the goal

        for next_state, direction in get_possible_moves(current_state):
            if tuple(map(tuple, next_state)) not in visited:
                queue.append((next_state, current_path + [(current_state, direction)]))
                visited.add(tuple(map(tuple, next_state)))

    return None  # No solution found

# Function to take matrix input from the user
```

```
def input_matrix():
    print("Enter the 3x3 matrix row by row (use 0 for the empty space):")
    matrix = []
    for _ in range(3):
        row = list(map(int, input().split()))
        matrix.append(row)
    return matrix
initial_state = input_matrix()
solution = solve_8_puzzle_bfs(initial_state)


if solution:
    print("Solution found:")
    for state, direction in solution:
        for row in state:
            print(row)
        print("Move:", direction)
        print()
    print("Total moves:", len(solution) - 1)  # Count of moves to reach the goal
else:
    print("No solution found.")
```

OUTPUT:

```
Enter the 3x3 matrix row by row (use 0 for the empty space):
1 2 3
0 4 6
7 5 8
Solution found:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
Move: Right

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

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

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

Total moves: 3
```

2) DFS:

CODE:

```
class Puzzle:

    def __init__(self, start):

        self.start = start

        self.goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]  # Default goal state

        self.n = 3  # 8 puzzle is 3x3


    def get_neighbors(self, state):

        # Find the blank space (0)

        blank_index = state.index(0)

        row, col = divmod(blank_index, self.n)

        neighbors = []


        # Define possible moves: up, down, left, right

        moves = [

            (-1, 0),  # up

            (1, 0),   # down

            (0, -1),  # left

            (0, 1)    # right

        ]


        for move in moves:

            new_row, new_col = row + move[0], col + move[1]


            if 0 <= new_row < self.n and 0 <= new_col < self.n:

                new_index = new_row * self.n + new_col

                new_state = list(state)
```

```
            # Swap the blank space (0) with the adjacent tile
            new_state[blank_index], new_state[new_index] = new_state[new_index],
new_state[blank_index]
            neighbors.append((tuple(new_state), (row, col), (new_row, new_col)))

        return neighbors

    def dfs(self):
        # Stack for DFS, starting from the initial state
        stack = [(tuple(self.start), [], None)]  # (state, path, previous_blank_pos)
        visited = set()

        while stack:
            state, path, previous_blank_pos = stack.pop()

            if state == tuple(self.goal):
                return path + [(state, previous_blank_pos)]  # Return path to the goal

            if state in visited:
                continue

            visited.add(state)

            for neighbor, prev_pos, new_pos in self.get_neighbors(state):
                if neighbor not in visited:
                    stack.append((neighbor, path + [(state, prev_pos, new_pos)], new_pos))

        return None  # No solution found

    def get_move_direction(self, prev_pos, new_pos):
```

```
        if prev_pos is None:
            return None
        prev_row, prev_col = prev_pos
        new_row, new_col = new_pos
        if new_row == prev_row and new_col == prev_col + 1:
            return "right"
        elif new_row == prev_row and new_col == prev_col - 1:
            return "left"
        elif new_row == prev_row - 1 and new_col == prev_col:
            return "up"
        elif new_row == prev_row + 1 and new_col == prev_col:
            return "down"

    def print_solution(self, solution):
        move_count = 0
        for i in range(len(solution) - 1):
            state, prev_pos, new_pos = solution[i]
            move_count += 1
            move_direction = self.get_move_direction(prev_pos, new_pos)

            print(f"Move {move_count} ({move_direction}):")
            for i in range(0, self.n * self.n, self.n):
                print(state[i:i + self.n])
            print("")  # Print empty line between moves

        # Print the final goal state
        final_state, _, _ = solution[-1]
        move_count += 1
        print(f"Move {move_count} (final state):")
```

```
        for i in range(0, self.n * self.n, self.n):
            print(final_state[i:i + self.n])
        print("")  # Final step empty line

        return move_count

# Input function to take the start state from the user row by row
def get_user_input_row_by_row(prompt):
    print(prompt)
    state = []
    for i in range(3):  # 8 puzzle is 3x3, so we need 3 rows
        row = input(f"Enter row {i+1} (space-separated numbers, 0 for blank): ").split()
        state.extend([int(x) for x in row])
    return state

# Main function
if __name__ == "__main__":
    print("Enter the start state of the puzzle (0 represents the blank space):")
    start_state = get_user_input_row_by_row("Start State")

    puzzle = Puzzle(start_state)
    solution = puzzle.dfs()

    if solution:
        print("Solution found! Here's how the puzzle solves step by step:")
        total_moves = puzzle.print_solution(solution)
        print(f"Total moves: {total_moves - 1}")  # Minus 1 as the initial state is included in the
solution
    else:
        print("No solution found.")
```