LAB 3:

8 Puzzle Game Problem

```
class Node:
    def __init__(self, state, parent=None, move=None, depth=0):
        self.state = state        # Current state of the puzzle (3x3 grid)
        self.parent = parent      # Parent node for backtracking
        self.move = move          # Move made to reach this node
        self.depth = depth        # Depth of this node in DFS tree

    def calculate_manhattan_distance(self):
        """Calculate the Manhattan distance for the current state."""
        total_distance = 0

        # Define goal positions of each tile in the target configuration
        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)  # 0 represents the empty space
        }

        # Calculate the total Manhattan distance
        for i in range(3):
            for j in range(3):
                tile = self.state[i][j]
                if tile != 0:  # Don't count the empty space
                    goal_x, goal_y = goal_positions[tile]
                    total_distance += abs(goal_x - i) + abs(goal_y - j)

        return total_distance
```

```
def is_solvable(state):
    """Check if the puzzle is solvable by counting inversions."""
    flat = [tile for row in state for tile in row if tile != 0]
    inversions = sum(1 for i in range(len(flat)) for j in range(i + 1, len(flat)) if flat[i] > flat[j])
    return inversions % 2 == 0  # Solvable if inversions are even


def dfs_with_manhattan(initial_state, goal_state, max_depth=30):
    """Perform DFS to solve the 8-puzzle, guided by Manhattan distance."""
    stack = [Node(initial_state)]  # DFS stack initialized with the starting configuration
    visited = set()  # To track visited states and avoid cycles

    while stack:
        node = stack.pop()  # Pop from stack (LIFO behavior for DFS)

        # If the current state matches the goal, return the solution path
        if node.state == goal_state:
            return construct_solution(node)

        # Mark the current state as visited
        visited.add(tuple(map(tuple, node.state)))

        # Limit the depth to prevent infinite loops
        if node.depth >= max_depth:
            continue

        # Get possible moves (up, down, left, right)
        for new_state, move in get_possible_moves(node.state):
            state_tuple = tuple(map(tuple, new_state))
            if state_tuple not in visited:
```

```
            # Push the new state to the stack
            new_node = Node(new_state, parent=node, move=move, depth=node.depth + 1)
            stack.append(new_node)

    return None  # If no solution found


def get_possible_moves(state):
    """Generate all valid moves from the current state."""
    moves = []
    empty_x, empty_y = find_empty_space(state)

    # Define directions for up, down, left, right
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for dx, dy in directions:
        new_x, new_y = empty_x + dx, empty_y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:  # Valid move
            new_state = [row[:] for row in state]  # Copy current state
            # Swap empty space with the adjacent tile
            new_state[empty_x][empty_y], new_state[new_x][new_y] = new_state[new_x][new_y],
new_state[empty_x][empty_y]
            moves.append((new_state, (dx, dy)))  # Append new state and move direction

    return moves


def find_empty_space(state):
    """Find the coordinates of the empty space (represented by 0)."""
    for i in range(3):
        for j in range(3):
```

```
            if state[i][j] == 0:
                return i, j
    return None


def construct_solution(node):
    """Construct the path of moves that leads to the solution."""
    path = []
    while node.parent is not None:  # Trace back from goal to start
        path.append(node)
        node = node.parent
    path.reverse()  # Reverse the path to get the correct order

    # Print each state along the solution path
    for step in path:
        print_puzzle(step.state)
        print()

    return path


def print_puzzle(state):
    """Helper function to print the puzzle state in a 3x3 grid format."""
    for row in state:
        print("  ".join(str(tile) if tile != 0 else " " for tile in row))


# Example usage:
if __name__ == "__main__":
    # Define the initial and goal states of the puzzle
    initial_state = [
```

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

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

# Check if the puzzle is solvable
if not is_solvable(initial_state):
    print("The puzzle is not solvable.")
else:
    # Solve the puzzle using DFS with Manhattan distance
    solution = dfs_with_manhattan(initial_state, goal_state)

    # Output the solution, if found
    if solution:
        print("Solution found!")
        print(f"Number of moves: {len(solution)}")
    else:
        print("No solution found.")
```

OUTPUT:

```
1  2  3
4     5
7  8  6

1  2  3
   4  5
7  8  6

1  2  3
7  4  5
   8  6

1  2  3
7  4  5
8     6

1  2  3
7  4  5
8  6

1  2  3
7  4
8  6  5

1  2  3
7     4
8  6  5

1  2  3
   7  4
8  6  5

1  2  3
8  7  4
   6  5
```

```
1   2   3
8   7   4
6       5

1   2   3
8   7   4
6   5

1   2   3
8   7
6   5   4

1   2   3
8       7
6   5   4

1   2   3
    8   7
6   5   4

1   2   3
6   8   7
    5   4

1   2   3
6   8   7
5       4

1   2   3
6   8   7
5   4

1   2   3
6   8
5   4   7
```

```
1   2   3
5   6   8
4       7

1   2   3
5   6   8
4   7

1   2   3
5   6
4   7   8

1   2   3
5       6
4   7   8

1   2   3
    5   6
4   7   8

1   2   3
4   5   6
    7   8

1   2   3
4   5   6
7       8

1   2   3
4   5   6
7   8

Solution found!
Number of moves: 29

=== Code Execution Successful ===
```