Name : **Abdul Rehman Saeed** Reg. No. : **FA22-BCS_055**

Question 1:
A robot named ReOrder is trapped in a control grid consisting of 8 movable panels and one empty slot. Each tile is labeled 1-8, and the empty slot (0) allows tiles to move. The robot must rearrange the tiles to restore the control system by reaching the target configuration. Rules:

1. Only the blank tile (0) can move up, down, left, or right.
2. Only one move is allowed at a time.
3. The robot must reach the goal state safely and efficiently. Tasks:
4. Represent the 3×3 puzzle grid as a Python list.
5. Implement the BFS, DFS, and A* algorithms to solve the puzzle.
6. Use Manhattan Distance as the heuristic in A*. Page 2 of 2
7. Display the full sequence of puzzle states from the start to the goal.

```
"""
8-Puzzle Solver - ReOrder Robot Control Grid
CLO-6 Assignment Solution
Implements BFS, DFS, and A* algorithms with Manhattan Distance heuristic
"""

import heapq
from collections import deque
import time

# ============================================================================
# PUZZLE STATE CLASS
# ============================================================================

class PuzzleState:
    """Represents a state of the 8-puzzle control grid"""
    
    def __init__(self, board, parent=None, move=None, depth=0):
        self.board = board  # 3x3 grid
        self.parent = parent  # Previous state
        self.move = move  # Move that led to this state
        self.depth = depth  # Depth in search tree
        self.blank_pos = self.find_blank()
        self.f_score = 0  # For A* (f = g + h)
    
    def find_blank(self):
        """Find the position of the blank tile (0)"""
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return (i, j)
        return None
    
    def get_possible_moves(self):
        """Generate all valid neighboring states"""
        neighbors = []
        row, col = self.blank_pos
        
        # Possible moves: UP, DOWN, LEFT, RIGHT
        directions = [
            ('UP', -1, 0),
            ('DOWN', 1, 0),
            ('LEFT', 0, -1),
            ('RIGHT', 0, 1)
        ]
        
        for move_name, drow, dcol in directions:
            new_row = row + drow
            new_col = col + dcol
            
```

```
            # Check if move is within grid boundaries
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                # Create new board by swapping blank with adjacent tile
                new_board = [row[:] for row in self.board]
                new_board[row][col], new_board[new_row][new_col] = \
                    new_board[new_row][new_col], new_board[row][col]
                
                # Create new state
                new_state = PuzzleState(new_board, self, move_name, self.depth + 1)
                neighbors.append(new_state)
        
        return neighbors
    
    def is_goal(self, goal_board):
        """Check if this state matches the goal configuration"""
        return self.board == goal_board
    
    def to_tuple(self):
        """Convert board to tuple for use in sets/dictionaries"""
        return tuple(tuple(row) for row in self.board)
    
    def __lt__(self, other):
        """Less than comparison for priority queue"""
        return self.f_score < other.f_score
    
    def __eq__(self, other):
        """Equality comparison"""
        return self.board == other.board
    
    def __hash__(self):
        """Hash function for use in sets"""
        return hash(self.to_tuple())

# ============================================================================
# HEURISTIC FUNCTION
# ============================================================================

def manhattan_distance(board, goal):
    """
    Calculate Manhattan Distance heuristic
    Sum of distances of each tile from its goal position
    """
    distance = 0
    
    for i in range(3):
        for j in range(3):
            value = board[i][j]
            if value != 0:  # Don't count the blank tile
                # Find where this value should be in goal state
                for goal_i in range(3):
                    for goal_j in range(3):
                        if goal[goal_i][goal_j] == value:
                            # Add Manhattan distance
                            distance += abs(i - goal_i) + abs(j - goal_j)
                            break
    
    return distance

# ============================================================================
# DISPLAY FUNCTIONS
# ============================================================================

def print_board(board):
    """Display the puzzle board in a nice format"""
    print("┌───────────┐")
    for row in board:
        print("│ ", end="")
        for val in row:
            if val == 0:
                print("  ", end=" ")  # Empty slot
```

```
            else:
                print(f"{val} ", end=" ")
        print("│")
    print("└─────────────┘")

def reconstruct_path(state):
    """Build the solution path from start to goal"""
    path = []
    while state is not None:
        path.append(state)
        state = state.parent
    return path[::-1]  # Reverse to get start -> goal order

def display_solution(solution_state, algorithm_name):
    """Display the complete solution sequence"""
    if solution_state is None:
        print(f"\n❌ {algorithm_name}: No solution found!")
        return None
    
    path = reconstruct_path(solution_state)
    
    print(f"\n{'='*60}")
    print(f"✅ {algorithm_name} - SOLUTION FOUND!")
    print(f"{'='*60}")
    print(f"Total Moves: {len(path) - 1}")
    print(f"{'='*60}\n")
    
    for step_num, state in enumerate(path):
        if state.move:
            print(f"Move {step_num}: {state.move}")
        else:
            print(f"Step 0: Initial State")
        print_board(state.board)
        print()
    
    return len(path) - 1

# ============================================================================
# ALGORITHM 1: BREADTH-FIRST SEARCH (BFS)
# ============================================================================

def bfs_solve(start_board, goal_board):
    """
    BFS Algorithm - Explores level by level
    Guarantees shortest path solution
    """
    print("\n🔵 Running BFS (Breadth-First Search)...")
    start_time = time.time()
    
    start_state = PuzzleState(start_board)
    
    # Check if already at goal
    if start_state.is_goal(goal_board):
        return start_state, 0, 0
    
    # Initialize queue and visited set
    queue = deque([start_state])
    visited = {start_state.to_tuple()}
    nodes_explored = 0
    
    while queue:
        current_state = queue.popleft()
        nodes_explored += 1
        
        # Explore all possible moves
        for neighbor in current_state.get_possible_moves():
            neighbor_tuple = neighbor.to_tuple()
            
            if neighbor_tuple not in visited:
                if neighbor.is_goal(goal_board):
                    elapsed_time = time.time() - start_time
```

```
                    elapsed_time = time.time() - start_time
                    print(f"✓ BFS completed in {elapsed_time:.4f} seconds")
                    print(f"✓ Nodes explored: {nodes_explored}")
                    return neighbor, nodes_explored, elapsed_time

                visited.add(neighbor_tuple)
                queue.append(neighbor)

    return None, nodes_explored, time.time() - start_time

# ============================================================================
# ALGORITHM 2: DEPTH-FIRST SEARCH (DFS)
# ============================================================================

def dfs_solve(start_board, goal_board, max_depth=30):
    """
    DFS Algorithm - Explores depth-first with a depth limit
    Uses less memory but may not find shortest path
    """
    print("\n🟢 Running DFS (Depth-First Search)...")
    start_time = time.time()

    start_state = PuzzleState(start_board)

    # Check if already at goal
    if start_state.is_goal(goal_board):
        return start_state, 0, 0

    # Initialize stack and visited set
    stack = [start_state]
    visited = {start_state.to_tuple()}
    nodes_explored = 0

    while stack:
        current_state = stack.pop()
        nodes_explored += 1

        # Skip if too deep
        if current_state.depth >= max_depth:
            continue

        # Explore all possible moves
        for neighbor in current_state.get_possible_moves():
            neighbor_tuple = neighbor.to_tuple()

            if neighbor_tuple not in visited:
                if neighbor.is_goal(goal_board):
                    elapsed_time = time.time() - start_time
                    print(f"✓ DFS completed in {elapsed_time:.4f} seconds")
                    print(f"✓ Nodes explored: {nodes_explored}")
                    return neighbor, nodes_explored, elapsed_time

                visited.add(neighbor_tuple)
                stack.append(neighbor)

    return None, nodes_explored, time.time() - start_time

# ============================================================================
# ALGORITHM 3: A* SEARCH
# ============================================================================

def a_star_solve(start_board, goal_board):
    """
    A* Algorithm - Uses Manhattan Distance heuristic
    Most efficient - finds shortest path with fewer node expansions
    """
    print("\n🔴 Running A* (A-Star Search) with Manhattan Distance...")
    start_time = time.time()

    start_state = PuzzleState(start_board)
    start_state.f_score = manhattan_distance(start_board, goal_board)
```

```python
    
    # Check if already at goal
    if start_state.is_goal(goal_board):
        return start_state, 0, 0
    
    # Initialize priority queue and cost tracking
    open_list = [start_state]
    heapq.heapify(open_list)
    visited = {start_state.to_tuple(): 0}  # State -> g_cost
    nodes_explored = 0
    
    while open_list:
        current_state = heapq.heappop(open_list)
        nodes_explored += 1
        
        # Check if goal reached
        if current_state.is_goal(goal_board):
            elapsed_time = time.time() - start_time
            print(f"✓ A* completed in {elapsed_time:.4f} seconds")
            print(f"✓ Nodes explored: {nodes_explored}")
            return current_state, nodes_explored, elapsed_time
        
        # Explore neighbors
        for neighbor in current_state.get_possible_moves():
            g_cost = neighbor.depth  # Cost from start
            h_cost = manhattan_distance(neighbor.board, goal_board)  # Heuristic
            f_cost = g_cost + h_cost  # Total cost
            
            neighbor.f_score = f_cost
            neighbor_tuple = neighbor.to_tuple()
            
            # Add to open list if not visited or found better path
            if neighbor_tuple not in visited or g_cost < visited[neighbor_tuple]:
                visited[neighbor_tuple] = g_cost
                heapq.heappush(open_list, neighbor)
    
    return None, nodes_explored, time.time() - start_time

# ============================================================================
# MAIN EXECUTION
# ============================================================================

def main():
    """Main function to run all algorithms and compare results"""
    
    print("="*60)
    print("🤖 REORDER ROBOT - 8-PUZZLE CONTROL GRID SOLVER")
    print("="*60)
    
    # Define the puzzle configuration
    # START STATE: Current configuration of the control grid
    start_state = [
        [1, 2, 3],
        [4, 0, 5],
        [7, 8, 6]
    ]
    
    # GOAL STATE: Target configuration to restore the system
    goal_state = [
        [1, 2, 3],
        [4, 5, 6],
        [7, 8, 0]
    ]
    
    print("\n📍 INITIAL STATE (Robot's Current Position):")
    print_board(start_state)
    
    print("\n🎯 GOAL STATE (Target Configuration):")
    print_board(goal_state)
    
    # Dictionary to store results
```

```
    results = {}

    # Run BFS
    print("\n" + "="*60)
    solution_bfs, nodes_bfs, time_bfs = bfs_solve(start_state, goal_state)
    if solution_bfs:
        moves_bfs = display_solution(solution_bfs, "BFS")
        results['BFS'] = {
            'moves': moves_bfs,
            'nodes': nodes_bfs,
            'time': time_bfs
        }

    # Run DFS
    print("\n" + "="*60)
    solution_dfs, nodes_dfs, time_dfs = dfs_solve(start_state, goal_state)
    if solution_dfs:
        moves_dfs = display_solution(solution_dfs, "DFS")
        results['DFS'] = {
            'moves': moves_dfs,
            'nodes': nodes_dfs,
            'time': time_dfs
        }

    # Run A*
    print("\n" + "="*60)
    solution_astar, nodes_astar, time_astar = a_star_solve(start_state, goal_state)
    if solution_astar:
        moves_astar = display_solution(solution_astar, "A*")
        results['A*'] = {
            'moves': moves_astar,
            'nodes': nodes_astar,
            'time': time_astar
        }

    # Display comparison
    print("\n" + "="*60)
    print("📊 ALGORITHM PERFORMANCE COMPARISON")
    print("="*60)
    print(f"{'Algorithm':<12} {'Moves':<10} {'Nodes':<15} {'Time (sec)':<12}")
    print("-"*60)

    for algo_name, data in results.items():
        print(f"{algo_name:<12} {data['moves']:<10} {data['nodes']:<15} {data['time']:<12.4f}")

    print("="*60)

    # Analysis
    print("\n💡 ANALYSIS:")
    print("   • BFS: Guarantees shortest path, explores many nodes")
    print("   • DFS: Uses less memory, may find longer paths")
    print("   • A*:  Most efficient with heuristic guidance (optimal!)")
    print("\n✅ Robot control system restored successfully!")
    print("="*60)

# Run the program
if __name__ == "__main__":
    main()
```

```
============================================================
🤖 REORDER ROBOT - 8-PUZZLE CONTROL GRID SOLVER
============================================================

📍 INITIAL STATE (Robot's Current Position):
┌─────────┐
│ 1  2  3 │
│ 4     5 │
│ 7  8  6 │
└─────────┘

🎯 GOAL STATE (Target Configuration):
```

```
┌───────────┐
│ 1  2  3  │
│ 4  5  6  │
│ 7  8     │
└───────────┘

============================================================

🔵 Running BFS (Breadth-First Search)...
✓ BFS completed in 0.0002 seconds
✓ Nodes explored: 5

============================================================
✅ BFS - SOLUTION FOUND!
============================================================
Total Moves: 2
============================================================

Step 0: Initial State
┌───────────┐
│ 1  2  3  │
│ 4     5  │
│ 7  8  6  │
└───────────┘

Move 1: RIGHT
┌───────────┐
│ 1  2  3  │
│ 4  5     │
│ 7  8  6  │
└───────────┘

Move 2: DOWN
┌───────────┐
│ 1  2  3  │
│ 4  5  6  │
│ 7  8     │
└───────────┘


============================================================

🟢 Running DFS (Depth-First Search)...
✓ DFS completed in 0.0001 seconds
✓ Nodes explored: 2
```