8-Puzzle DFS Algorithm

1. **Define Goal State:**
   - Set the goal state as `(1, 2, 3, 4, 5, 6, 7, 8, 0)`.
2. **Input Starting State:**
   - Prompt the user for the initial configuration of the puzzle as 9 numbers (0-8).
3. **Initialize DFS:**
   - Create a stack and push the starting state with an empty path.
   - Create a set to track visited states.
   - Initialize a counter for the visited states.
4. **DFS Loop:**
   - While the stack is not empty:
     - Pop the top state and its associated path from the stack.
     - If the current state matches the goal state, return the path and the visited count.
     - Generate all valid neighboring states by sliding tiles into the empty space:
       - For each direction (up, down, left, right):
         - If the move is valid, create a new state.
         - If the new state has not been visited:
           - Mark it as visited and push it onto the stack with the updated path.
           - Increment the visited states counter.
5. **Output Result:**
   - If the goal state is found, print the sequence of moves and the number of states visited.
   - If the stack is empty and the goal is not reached, indicate that no solution exists along with the visited states count.

LAB 4

```
import heapq

# Define the goal state as a tuple
GOAL_STATE = (1, 2, 3, 8, 0, 4, 7, 6, 5)

# Function to find the index of the empty space (0)
def find_empty(state):
    return state.index(0)

# Heuristic function that counts the number of mismatched tiles
def mismatched_tiles(state):
    return sum(1 for i, tile in enumerate(state) if tile != 0 and tile != GOAL_STATE[i])

# Function to get neighbors (all possible moves from current state)
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)]  # up, down, left, right
    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

# A* algorithm implementation using mismatched tiles heuristic
def a_star(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state, []))  # (priority, state, path)
    visited = set()
    visited.add(initial_state)
    visited_count = 1  # Initialize visited count
    cost_so_far = {initial_state: 0}  # Tracks the cost to reach each state

    while priority_queue:
        priority, current_state, path = heapq.heappop(priority_queue)
```

```
        if current_state == GOAL_STATE:
            depth = len(path)
            return path, visited_count, depth, cost_so_far[current_state]

        # Explore neighbors
        for neighbor in get_neighbors(current_state):
            new_cost = cost_so_far[current_state] + 1  # Cost to move to the neighbor

            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                priority = new_cost + mismatched_tiles(neighbor)  # A* f = g + h
                heapq.heappush(priority_queue, (priority, neighbor, path + [neighbor]))
                if neighbor not in visited:
                    visited.add(neighbor)
                    visited_count += 1  # Increment visited count

    return None, visited_count, 0, 0  # No solution found

# Function to take user input for the initial state
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)

# Function to print the state as a 3x3 matrix
def print_matrix(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])

# Main function
if __name__ == "__main__":
    initial_state = input_start_state()
    print("Initial state:")
    print_matrix(initial_state)

    # Run A* algorithm
    solution, visited_count, depth, cost = a_star(initial_state)

    print(f"Number of states visited: {visited_count}")
    if solution:
        print(f"Solution found at depth: {depth} with cost: {cost}")
        print("\nSolution steps:")
        for step in solution:
            print_matrix(step)
            print()
    else:
        print("No solution found.")
```

```
Enter the starting state as 9 numbers (0 for the empty space):
Format: 1 2 3 4 5 6 7 8 0
2 8 3 1 6 4 7 0 5
Initial state:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)
Number of states visited: 12
Solution found at depth: 5 with cost: 5

Solution steps:
(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
```

Algorithm for 8-Puzzle Problem Using A* Algorithm:

1. **Input**:
   - Take the **initial state** of the puzzle as input from the user, representing the 3x3 grid (0 represents the empty tile).
   - Define the **goal state**, which is the target arrangement of the puzzle tiles.
2. **Initialize Data Structures**:
   - **Priority Queue (min-heap)**: Start with the initial state in the queue. Each element in the queue is a tuple: `(priority, current_state, path)`.
     - `priority` = `g + h` where:
       - `g` is the number of moves made to reach the current state.
       - `h` is the heuristic value (number of mismatched tiles compared to the goal).
   - **Visited Set**: Keeps track of all previously visited states to avoid cycles and redundant processing.
   - **Cost Dictionary**: Stores the cost to reach each state, i.e., the number of moves (`g` cost).
3. **Heuristic Function (h)**:
   - Calculate the heuristic for a state as the **number of tiles that are not in their correct positions** (ignoring the empty tile).
4. **Main A* Algorithm Loop**:
   - While the priority queue is not empty:
     1. **Dequeue the state** with the lowest priority (`g + h`).
     2. If the dequeued state is the **goal state**, return the path, cost, and other information (depth, states visited).
     3. Otherwise, generate all possible **neighbor states** by moving the blank tile (0) in four possible directions: up, down, left, and right.
     4. For each neighbor state:
        - Calculate the **new cost (g)** to reach the neighbor by incrementing the cost of the current state by 1.
        - Calculate the **heuristic (h)** for the neighbor using the mismatched tiles heuristic.
        - If the neighbor state has not been visited or if a lower-cost path to the neighbor is found:
          - Update the cost to reach the neighbor.
          - Add the neighbor state to the priority queue with its new priority (`g + h`).
     5. Mark the current state as **visited**.
5. **Exit Condition**:
   - If the priority queue becomes empty without finding the goal state, the puzzle is unsolvable.
6. **Output**:
   - If the goal is reached, return the following:
     - The **solution path**, which is a sequence of states from the initial to the goal state.
     - The **depth** of the solution, which is the number of moves taken to reach the goal.
     - The **cost**, which is the same as the depth (total moves).
     - The **number of states visited** during the search.

```python
import heapq

# Define the goal state as a tuple
GOAL_STATE = (1, 2, 3, 8, 0, 4, 7, 6, 5)

# Function to find the index of the empty space (0)
def find_empty(state):
    return state.index(0)

# Heuristic function that calculates the Manhattan distance
def manhattan_distance(state):
    distance = 0
    for i, tile in enumerate(state):
        if tile != 0:  # Skip the blank tile (0)
            # Calculate the correct position (row, col) in the goal state
            correct_row, correct_col = divmod(tile - 1, 3)
            # Calculate the current position (row, col)
            current_row, current_col = divmod(i, 3)
            # Add the Manhattan distance for this tile
            distance += abs(correct_row - current_row) + abs(correct_col - current_col)
    return distance

# Function to get neighbors (all possible moves from current state)
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)]  # up, down, left, right
    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
```

```
# A* algorithm implementation using Manhattan distance heuristic
def a_star(initial_state):
    priority_queue = []
    heapq.heappush(priority_queue, (0, initial_state, []))  # (priority, state, path)
    visited = set()
    visited.add(initial_state)
    visited_count = 1  # Initialize visited count
    cost_so_far = {initial_state: 0}  # Tracks the cost to reach each state

    while priority_queue:
        priority, current_state, path = heapq.heappop(priority_queue)

        if current_state == GOAL_STATE:
            depth = len(path)
            return path, visited_count, depth, cost_so_far[current_state]

        # Explore neighbors
        for neighbor in get_neighbors(current_state):
            new_cost = cost_so_far[current_state] + 1  # Cost to move to the neighbor

            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
                cost_so_far[neighbor] = new_cost
                priority = new_cost + manhattan_distance(neighbor)  # A* f = g + h
                heapq.heappush(priority_queue, (priority, neighbor, path + [neighbor]))
                if neighbor not in visited:
                    visited.add(neighbor)
                    visited_count += 1  # Increment visited count

    return None, visited_count, 0, 0  # No solution found

# Function to take user input for the initial state
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)

# Function to print the state as a 3x3 matrix
def print_matrix(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])

# Main function
if __name__ == "__main__":
    initial_state = input_start_state()
    print("Initial state:")
    print_matrix(initial_state)

    # Run A* algorithm
    solution, visited_count, depth, cost = a_star(initial_state)

    print(f"Number of states visited: {visited_count}")
    if solution:
        print(f"Solution found at depth: {depth} with cost: {cost}")
        print("\nSolution steps:")
        for step in solution:
            print_matrix(step)
            print()
    else:
        print("No solution found.")
```

```
Enter the starting state as 9 numbers (0 for the empty space):
Format: 1 2 3 4 5 6 7 8 0
2 8 3 1 6 4 7 0 5
Initial state:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)
Number of states visited: 34
Solution found at depth: 5 with cost: 5

Solution steps:
(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

(2, 0, 3)
(1, 8, 4)
(7, 6, 5)
```

```
(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
```

Algorithm for 8-Puzzle Problem Using A* with Manhattan Distance:

1. **Input**:
   - Take the **initial state** of the puzzle from the user as a list of 9 numbers, where `0` represents the empty tile.
   - Define the **goal state**, which is the target arrangement of the puzzle tiles, usually `[1, 2, 3, 4, 5, 6, 7, 8, 0]`.
2. **Initialize Data Structures**:
   - **Priority Queue (min-heap)**: Start with the initial state. Each element in the queue is a tuple: `(priority, current_state, path)`.
     - `priority` = `g + h` where:
       - `g` is the cost to reach the current state (the number of moves taken so far).
       - `h` is the heuristic value (the Manhattan distance for the current state).
   - **Visited Set**: Keeps track of previously visited states to avoid cycles and redundant exploration.
   - **Cost Dictionary**: Stores the cost to reach each state (`g` value).
3. **Heuristic Function (Manhattan Distance)**:
   - For each tile in the current state, calculate how far the tile is from its correct position in the goal state.
   - The Manhattan distance for a tile is the sum of the vertical and horizontal distances between the tile's current position and its target position.
   - The total heuristic value `h` is the sum of the Manhattan distances for all tiles (excluding the empty tile `0`).
4. **Main A* Algorithm Loop**:
   - While the priority queue is not empty:
     1. **Dequeue the state** with the lowest priority (`g + h`).
     2. If the dequeued state is the **goal state**, return the solution path, the number of states visited, the depth, and the cost (number of moves).
     3. Otherwise, generate all possible **neighbor states** by moving the blank tile (0) in four directions: up, down, left, and right.
     4. For each neighbor state:
        - Calculate the **new cost (g)** to reach the neighbor by incrementing the cost of the current state by 1.
        - Calculate the **Manhattan distance (h)** for the neighbor state.
        - If the neighbor state has not been visited or if a lower-cost path to the neighbor is found:
          - Update the cost to reach the neighbor.
          - Add the neighbor state to the priority queue with its new priority (`g + h`).
     5. Mark the current state as **visited** and track the number of states visited.
5. **Exit Condition**:
   - If the priority queue becomes empty without finding the goal state, then the puzzle is unsolvable.
6. **Output**:
   - If the goal is reached, return the following:
     - The **solution path** showing each step from the initial state to the goal state.

- The **depth** of the solution (i.e., the number of moves taken to reach the goal).
- The **cost**, which is the total number of moves (same as depth).
- The **number of states visited** during the search process.