#USING MANHATTAN DISTANCE

```
import heapq

# Function to calculate Manhattan distance for the current state
def manhattan_distance(state, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                # Find the goal position of the current tile
                goal_x, goal_y = divmod(goal.index(state[i][j]), 3)
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

# Function to get the position of the empty tile (0)
def get_empty_tile_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

# Function to generate new states by moving the empty tile
def generate_new_states(state):
    i, j = get_empty_tile_position(state)
    possible_moves = []
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
    for di, dj in directions:
        new_i, new_j = i + di, j + dj
        if 0 <= new_i < 3 and 0 <= new_j < 3:
            new_state = [row[:] for row in state]
```

```
            # Swap the empty tile with the neighboring tile
            new_state[i][j], new_state[new_i][new_j] = new_state[new_i][new_j], new_state[i][j]
            possible_moves.append(new_state)
    return possible_moves

# Function to print the puzzle state in a readable format
def print_puzzle(state):
    for row in state:
        print(' '.join(str(x) if x != 0 else '_' for x in row))
    print()

# A* algorithm using Manhattan distance with detailed logging
def a_star(start, goal):
    # Flatten the goal state for easy indexing during Manhattan distance calculation
    goal_flat = [num for row in goal for num in row]

    # Priority queue (min-heap) where each entry is (f(n), g(n), state, previous moves)
    queue = []
    heapq.heappush(queue, (manhattan_distance(start, goal_flat), 0, start, []))

    # Set to store visited states
    visited = set()
    visited.add(tuple(tuple(row) for row in start))

    while queue:
        # Select the node with the lowest f(n)
        f, g, current_state, path = heapq.heappop(queue)

        # Print the current step
        print(f"Step {g}:")
```

```
print(f"g(n) = {g}, h(n) = {f - g}")
print("Current puzzle state:")
print_puzzle(current_state)

# If current state is the goal, we are done
if current_state == goal:
    print("Goal state reached!")
    print(f"Total moves: {g}")
    print("Solution path:")
    for step, state in enumerate(path + [current_state], 1):
        print(f"Move {step}:")
        print_puzzle(state)
    return

# Generate possible next states
print("Generated new states (possible moves):")
trial_states = []
for new_state in generate_new_states(current_state):
    if tuple(tuple(row) for row in new_state) not in visited:
        h = manhattan_distance(new_state, goal_flat)
        trial_states.append((g + h + 1, g + 1, new_state, h))
        print(f"g(n) = {g + 1}, h(n) = {h}")
        print_puzzle(new_state)

# Add unvisited states to the queue
print("Evaluating and choosing the best state based on f(n):")
for f_new, g_new, state, h_new in trial_states:
    if tuple(tuple(row) for row in state) not in visited:
        heapq.heappush(queue, (f_new, g_new, state, path + [current_state]))
        visited.add(tuple(tuple(row) for row in state))
```

```
            print(f"Chosen state with f(n) = {f_new} (g(n) = {g_new}, h(n) = {h_new}):")
            print_puzzle(state)

    print("No solution found")
    return

# Example usage
start_state = [
    [2, 8, 3],
    [1, 6, 4],
    [0, 7, 5]
]

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

a_star(start_state, goal_state)
```

output:

```
Step 0:
g(n) = 0, h(n) = 6
Current puzzle state:
2 8 3
1 6 4
_ 7 5

Generated new states (possible moves):
g(n) = 1, h(n) = 7
2 8 3
_ 6 4
```

```
1 7 5

g(n) = 1, h(n) = 5
2 8 3
1 6 4
7 _ 5

Evaluating and choosing the best state based on f(n):
Chosen state with f(n) = 8 (g(n) = 1, h(n) = 7):
2 8 3
_ 6 4
1 7 5

Chosen state with f(n) = 6 (g(n) = 1, h(n) = 5):
2 8 3
1 6 4
7 _ 5

Step 1:
g(n) = 1, h(n) = 5
Current puzzle state:
2 8 3
1 6 4
7 _ 5

Generated new states (possible moves):
g(n) = 2, h(n) = 4
2 8 3
1 _ 4
7 6 5

g(n) = 2, h(n) = 6
2 8 3
1 6 4
7 5 _

Evaluating and choosing the best state based on f(n):
Chosen state with f(n) = 6 (g(n) = 2, h(n) = 4):
2 8 3
1 _ 4
7 6 5

Chosen state with f(n) = 8 (g(n) = 2, h(n) = 6):
2 8 3
1 6 4
7 5 _

Step 2:
g(n) = 2, h(n) = 4
Current puzzle state:
2 8 3
1 _ 4
7 6 5
```

```
Generated new states (possible moves):
g(n) = 3, h(n) = 3
2 _ 3
1 8 4
7 6 5

g(n) = 3, h(n) = 5
2 8 3
_ 1 4
7 6 5

g(n) = 3, h(n) = 5
2 8 3
1 4 _
7 6 5

Evaluating and choosing the best state based on f(n):
Chosen state with f(n) = 6 (g(n) = 3, h(n) = 3):
2 _ 3
1 8 4
7 6 5

Chosen state with f(n) = 8 (g(n) = 3, h(n) = 5):
2 8 3
_ 1 4
7 6 5

Chosen state with f(n) = 8 (g(n) = 3, h(n) = 5):
2 8 3
1 4 _
7 6 5

Step 3:
g(n) = 3, h(n) = 3
Current puzzle state:
2 _ 3
1 8 4
7 6 5

Generated new states (possible moves):
g(n) = 4, h(n) = 2
_ 2 3
1 8 4
7 6 5

g(n) = 4, h(n) = 4
2 3 _
1 8 4
7 6 5

Evaluating and choosing the best state based on f(n):
Chosen state with f(n) = 6 (g(n) = 4, h(n) = 2):
```

```
_ 2 3
1 8 4
7 6 5

Chosen state with f(n) = 8 (g(n) = 4, h(n) = 4):
2 3 _
1 8 4
7 6 5

Step 4:
g(n) = 4, h(n) = 2
Current puzzle state:
_ 2 3
1 8 4
7 6 5

Generated new states (possible moves):
g(n) = 5, h(n) = 1
1 2 3
_ 8 4
7 6 5

Evaluating and choosing the best state based on f(n):
Chosen state with f(n) = 6 (g(n) = 5, h(n) = 1):
1 2 3
_ 8 4
7 6 5

Step 5:
g(n) = 5, h(n) = 1
Current puzzle state:
1 2 3
_ 8 4
7 6 5

Generated new states (possible moves):
g(n) = 6, h(n) = 2
1 2 3
7 8 4
_ 6 5

g(n) = 6, h(n) = 0
1 2 3
8 _ 4
7 6 5

Evaluating and choosing the best state based on f(n):
Chosen state with f(n) = 8 (g(n) = 6, h(n) = 2):
1 2 3
7 8 4
_ 6 5

Chosen state with f(n) = 6 (g(n) = 6, h(n) = 0):
```

```
1 2 3
8 _ 4
7 6 5

Step 6:
g(n) = 6, h(n) = 0
Current puzzle state:
1 2 3
8 _ 4
7 6 5

Goal state reached!
Total moves: 6
Solution path:
Move 1:
2 8 3
1 6 4
_ 7 5

Move 2:
2 8 3
1 6 4
7 _ 5

Move 3:
2 8 3
1 _ 4
7 6 5

Move 4:
2 _ 3
1 8 4
7 6 5

Move 5:
_ 2 3
1 8 4
7 6 5

Move 6:
1 2 3
_ 8 4
7 6 5

Move 7:
1 2 3
8 _ 4
7 6 5
```