Indian Institute of Engineering Science & Technology, Shibpur
Department of Computer Science & Technology
Artificial Intelligence Laboratory 2025 CS 4271

ASSIGNMENT – 01

NAME - RAKSHA PAHARIYA

ENROLLMENT NO - 2021CSB029

https://colab.research.google.com/drive/1qARKsCBIqNo-qE1KExdNyio2T8vwtbkl?usp=sharing

---

Question 1 : In the realm of Artificial Intelligence, contemplate a problem involving two containers of indeterminate capacity, referred to as jugs. One jug has a capacity of 3 units, while the other holds up to 4 units. There is no markings or additional measuring instruments, the objective is to develop a strategic approach to precisely fill the 4-unit jug with 2 units of water. The restriction stipulates the use of solely the aforementioned jugs, excluding any supplementary tools. Both jugs initiate the scenario in an empty state. The aim is to attain the desired water quantity in the 4-unit jug by executing a sequence of permissible operations, including filling, emptying, and pouring water between the jugs. The challenge in this scenario involves crafting an algorithm:

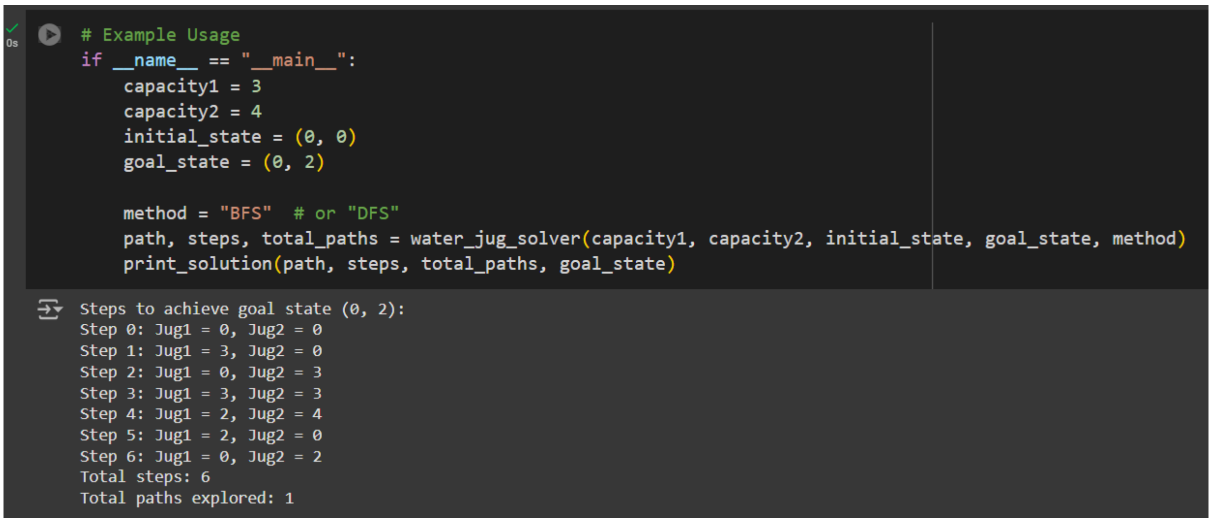
a. Define the permissible operations carefully that includes filling, emptying, and pouring water between the jugs.
b. Use both Depth First Search and Breadth First Search to systematically explore and determine the optimal sequence of moves for accomplishing the task while adhering to the defined constraints. Also determine the total path count to reach to the goal state.
c. The initial and the goal state of the jugs may be varied.

Solution 1 : Language Used is Python

```
from collections import deque

def is_valid_state(state, capacity1, capacity2):
    """
    Check if the given state is valid within the jug capacities.
    """
    x, y = state
    return 0 <= x <= capacity1 and 0 <= y <= capacity2

def generate_next_states(state, capacity1, capacity2):
    """
    Generate all possible next states from the current state.
    """
    x, y = state
    return [
        (capacity1, y),  # Fill jug1 completely
        (x, capacity2),  # Fill jug2 completely
        (0, y),          # Empty jug1
        (x, 0),          # Empty jug2
        (max(0, x - (capacity2 - y)), min(capacity2, x + y)),  # Pour jug1 -> jug2
        (min(capacity1, x + y), max(0, y - (capacity1 - x)))   # Pour jug2 -> jug1
    ]

def water_jug_solver(capacity1, capacity2, initial_state, goal_state, method="BFS"):
    """
    Solve the water jug problem using the specified method (BFS or DFS).

    Args:
        capacity1 (int): Capacity of the first jug.
        capacity2 (int): Capacity of the second jug.
        initial_state (tuple): Starting state, e.g., (0, 0).
        goal_state (tuple): Target state to reach, e.g., (0, 2).
        method (str): Method to use ("BFS" or "DFS").

    Returns:
        path (list): Sequence of states to reach the target.
        steps (int): Total number of steps to reach the target.
        total_paths (int): Total paths explored during the search.
    """
    visited = set()
    total_paths = 0

    if method == "BFS":
        queue = deque([(initial_state, [])])
        while queue:
            current_state, path = queue.popleft()

            if current_state in visited:
                continue
            visited.add(current_state)

            path = path + [current_state]

            # Check if the goal state is reached
            if current_state == goal_state:
                total_paths += 1
                return path, len(path) - 1, total_paths

            # Generate and explore next states
            for next_state in generate_next_states(current_state, capacity1, capacity2):
                if is_valid_state(next_state, capacity1, capacity2) and next_state not in visited:
                    queue.append((next_state, path))

    elif method == "DFS":
```

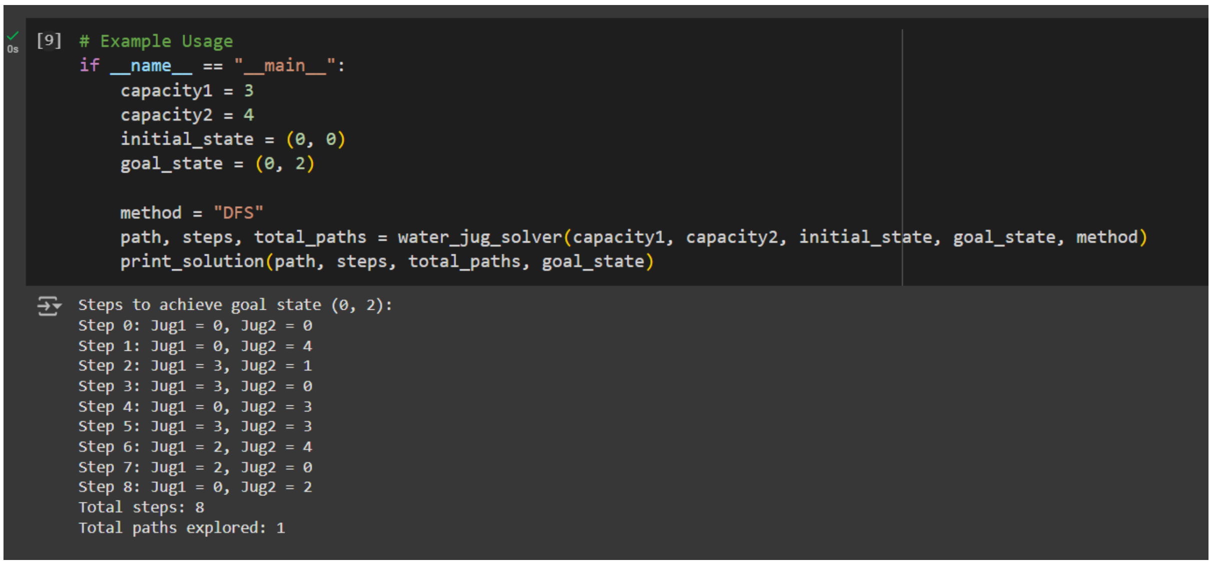
```
    stack = [(initial_state, [])]
    while stack:
        current_state, path = stack.pop()

        if current_state in visited:
            continue
        visited.add(current_state)

        path = path + [current_state]

        # Check if the goal state is reached
        if current_state == goal_state:
            total_paths += 1
            return path, len(path) - 1, total_paths

        # Generate and explore next states
        for next_state in generate_next_states(current_state, capacity1, capacity2):
            if is_valid_state(next_state, capacity1, capacity2) and next_state not in visited:
                stack.append((next_state, path))

    return None, -1, 0

def print_solution(path, steps, total_paths, goal_state):
    """
    Print the solution details, including the path, total steps, and paths explored.
    """
    if path:
        print(f"Steps to achieve goal state {goal_state}:")
        for i, step in enumerate(path):
            print(f"Step {i}: Jug1 = {step[0]}, Jug2 = {step[1]}")
        print(f"Total steps: {steps}")
        print(f"Total paths explored: {total_paths}")
    else:
        print("No solution found.")
```

OUTPUTS :



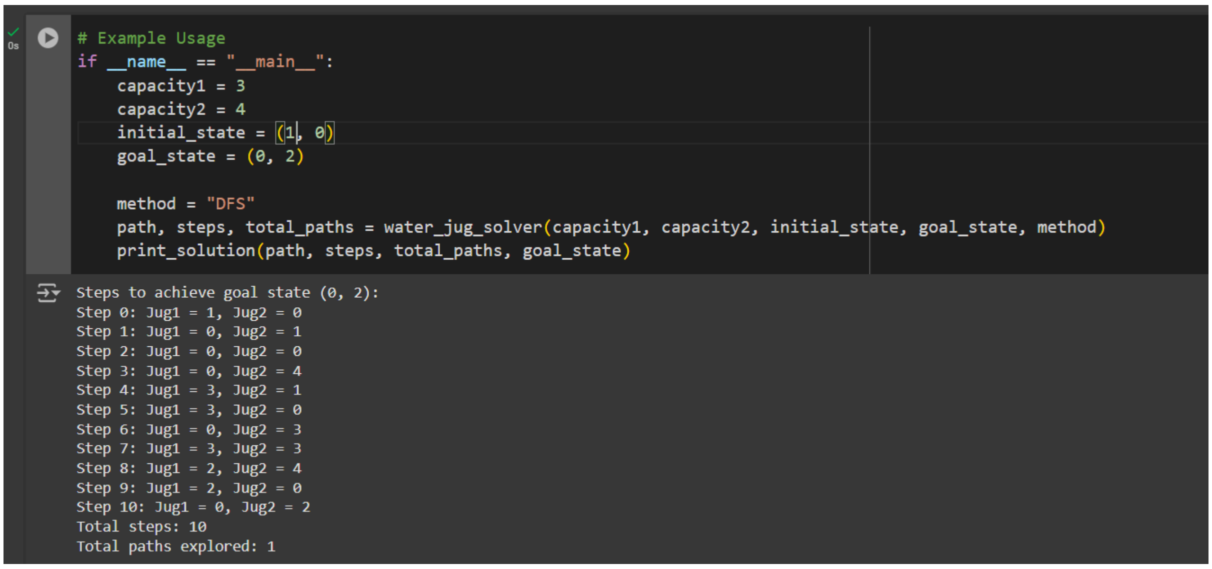
```
# Example Usage
if __name__ == "__main__":
    capacity1 = 3
    capacity2 = 4
    initial_state = (0, 0)
    goal_state = (0, 2)

    method = "BFS"  # or "DFS"
    path, steps, total_paths = water_jug_solver(capacity1, capacity2, initial_state, goal_state, method)
    print_solution(path, steps, total_paths, goal_state)
```

```
Steps to achieve goal state (0, 2):
Step 0: Jug1 = 0, Jug2 = 0
Step 1: Jug1 = 3, Jug2 = 0
Step 2: Jug1 = 0, Jug2 = 3
Step 3: Jug1 = 3, Jug2 = 3
Step 4: Jug1 = 2, Jug2 = 4
Step 5: Jug1 = 2, Jug2 = 0
Step 6: Jug1 = 0, Jug2 = 2
Total steps: 6
Total paths explored: 1
```



```
[9] # Example Usage
if __name__ == "__main__":
    capacity1 = 3
    capacity2 = 4
    initial_state = (0, 0)
    goal_state = (0, 2)

    method = "DFS"
    path, steps, total_paths = water_jug_solver(capacity1, capacity2, initial_state, goal_state, method)
    print_solution(path, steps, total_paths, goal_state)
```

```
Steps to achieve goal state (0, 2):
Step 0: Jug1 = 0, Jug2 = 0
Step 1: Jug1 = 0, Jug2 = 4
Step 2: Jug1 = 3, Jug2 = 1
Step 3: Jug1 = 3, Jug2 = 0
Step 4: Jug1 = 0, Jug2 = 3
Step 5: Jug1 = 3, Jug2 = 3
Step 6: Jug1 = 2, Jug2 = 4
Step 7: Jug1 = 2, Jug2 = 0
Step 8: Jug1 = 0, Jug2 = 2
Total steps: 8
Total paths explored: 1
```



```
# Example Usage
if __name__ == "__main__":
    capacity1 = 3
    capacity2 = 4
    initial_state = (1, 0)
    goal_state = (0, 2)

    method = "DFS"
    path, steps, total_paths = water_jug_solver(capacity1, capacity2, initial_state, goal_state, method)
    print_solution(path, steps, total_paths, goal_state)
```

```
Steps to achieve goal state (0, 2):
Step 0: Jug1 = 1, Jug2 = 0
Step 1: Jug1 = 0, Jug2 = 1
Step 2: Jug1 = 0, Jug2 = 0
Step 3: Jug1 = 0, Jug2 = 4
Step 4: Jug1 = 3, Jug2 = 1
Step 5: Jug1 = 3, Jug2 = 0
Step 6: Jug1 = 0, Jug2 = 3
Step 7: Jug1 = 3, Jug2 = 3
Step 8: Jug1 = 2, Jug2 = 4
Step 9: Jug1 = 2, Jug2 = 0
Step 10: Jug1 = 0, Jug2 = 2
Total steps: 10
Total paths explored: 1
```

<u>Question 2 :</u> Develop a comprehensive program that effectively addresses a puzzle problem. The puzzle involves a 3x3 grid with eight numbered tiles and an empty space (Given in the diagram). The task is to create a program that can systematically rearrange the tiles, around the empty cells, to reach to the predefined goal state from the initial configuration adhering to the constraints of permissible moves.

a. Use two different heuristic functions: one, the total count of the number of misplaced cells to reach to the goal state, second, consider the Manhattan distance as a heuristic function to determine the distance to reach to the goal state.

1	2	3
8		4
7	6	5

Initial State

2	8	1
	4	3
7	6	5

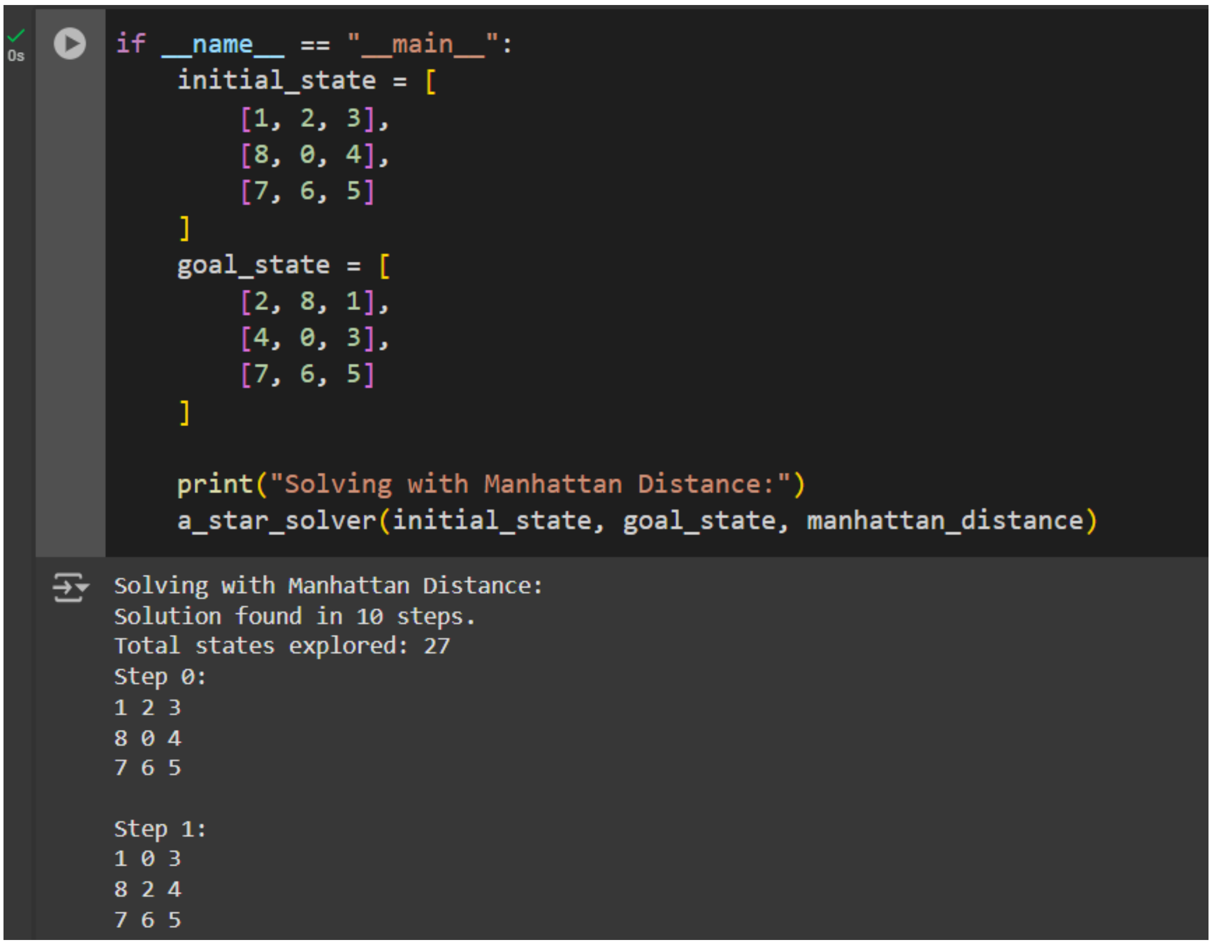
Goal State

<u>Solution 2 :</u> Language Used is Python

```
import heapq
import copy


class Puzzle:
   def __init__(self, goal):
      self.goal = goal

   def find_empty(self, board):
      """
      Find the position of the empty space (0) in the puzzle.
      """
      for i, row in enumerate(board):
         for j, val in enumerate(row):
            if val == 0:
               return i, j

   def move(self, board, x, y, new_x, new_y):
      """
      Move the empty space (0) to a new position.
      """
      new_board = copy.deepcopy(board)
      new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
      return new_board

   def get_neighbors(self, board):
      """
      Generate all possible moves (neighbors) for a given board.
      """
      x, y = self.find_empty(board)
```

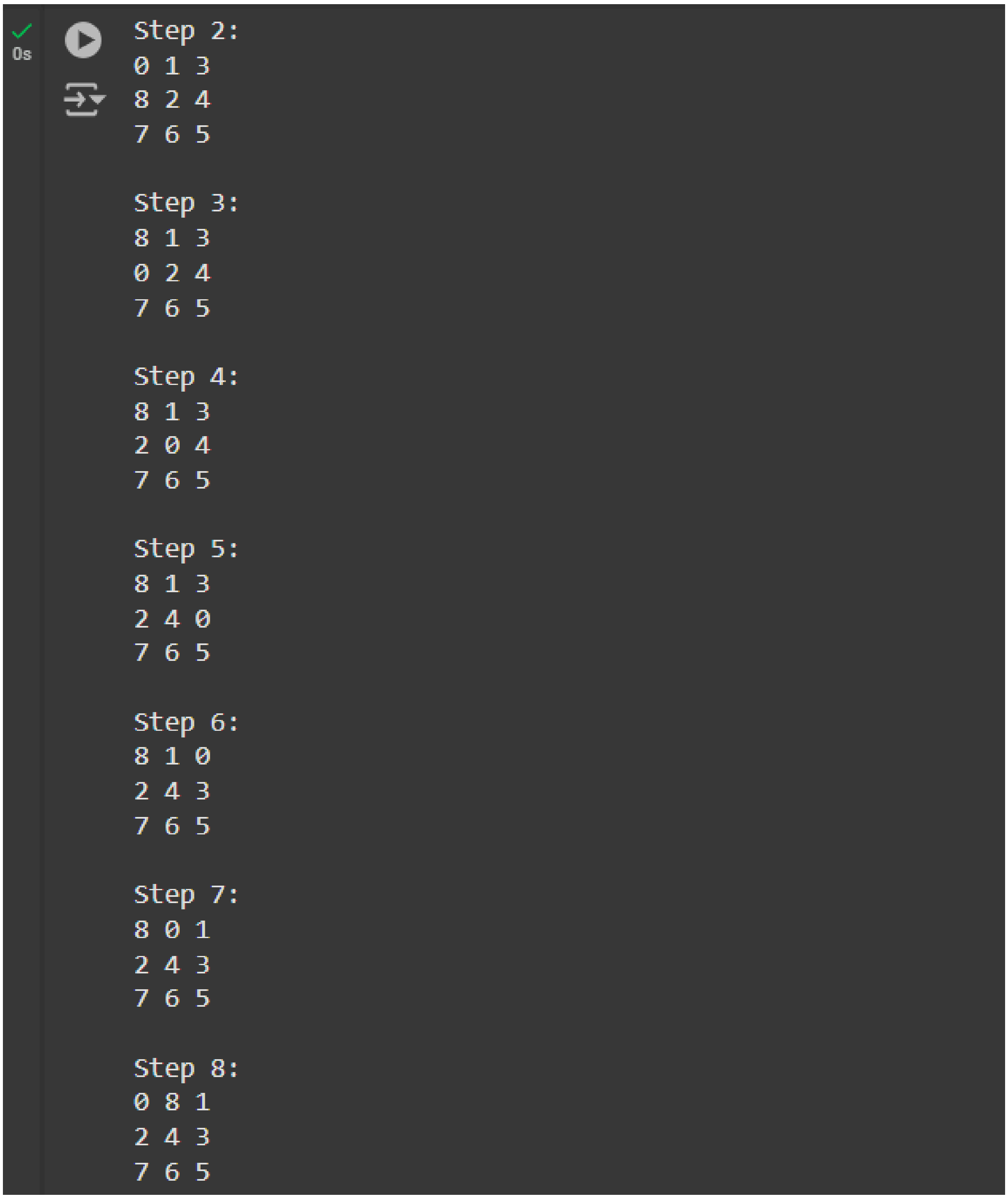
```
        neighbors = []
        moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up
        for dx, dy in moves:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                neighbors.append(self.move(board, x, y, new_x, new_y))
        return neighbors

    def is_goal(self, board):
        """
        Check if the given board is the goal state.
        """
        return board == self.goal


def a_star_solver(initial_state, goal_state, heuristic_function):
    """
    Solve the 8-puzzle problem using the A* algorithm.

    Args:
        initial_state (list): The starting state of the puzzle as a 2D list.
        goal_state (list): The goal state of the puzzle as a 2D list.
        heuristic_function (function): A heuristic function that takes a board
                                        and the goal state and returns a heuristic value.

    Returns:
        path (list): Sequence of states leading to the solution.
        steps (int): Number of steps in the solution.
        explored_states (int): Total number of states explored.
    """
    puzzle = Puzzle(goal_state)
    open_set = []
    heapq.heappush(open_set, (0, initial_state, []))  # (priority, board, path)
    closed_set = set()
    explored_states = 0  # Counter for explored states

    while open_set:
        cost, current_board, path = heapq.heappop(open_set)
        explored_states += 1  # Increment on every state expansion

        if tuple(map(tuple, current_board)) in closed_set:
            continue

        closed_set.add(tuple(map(tuple, current_board)))
        path = path + [current_board]

        if puzzle.is_goal(current_board):
            print(f"Solution found in {len(path) - 1} steps.")
            print(f"Total states explored: {explored_states}")
            print_solution(path)
            return path, len(path) - 1, explored_states

        for neighbor in puzzle.get_neighbors(current_board):
            if tuple(map(tuple, neighbor)) not in closed_set:
                g = len(path)  # Cost so far
            h = heuristic_function(neighbor, goal_state)  # Use the provided heuristic function
            heapq.heappush(open_set, (g + h, neighbor, path))

    print("No solution found.")
    return None, -1, explored_states


def print_solution(path):
    """
    Print the sequence of moves to reach the goal state.
    """
    for step, board in enumerate(path):
        print(f"Step {step}:")
        for row in board:
            print(" ".join(map(str, row)))
        print()


# Heuristic Functions
def misplaced_tiles(board, goal):
    """
    Calculate the number of misplaced tiles.
    """
    return sum(
        1 for i in range(3) for j in range(3) if board[i][j] != 0 and board[i][j] != goal[i][j]
    )


def manhattan_distance(board, goal):
    """
    Calculate the total Manhattan distance of tiles from their goal positions.
    """
    dist = 0
    for i in range(3):
        for j in range(3):
            val = board[i][j]
            if val != 0:
                goal_x, goal_y = [
                    (x, y) for x, row in enumerate(goal) for y, val_g in enumerate(row) if val_g == val
                ][0]
                dist += abs(i - goal_x) + abs(j - goal_y)
    return dist
```

OUTPUTS :

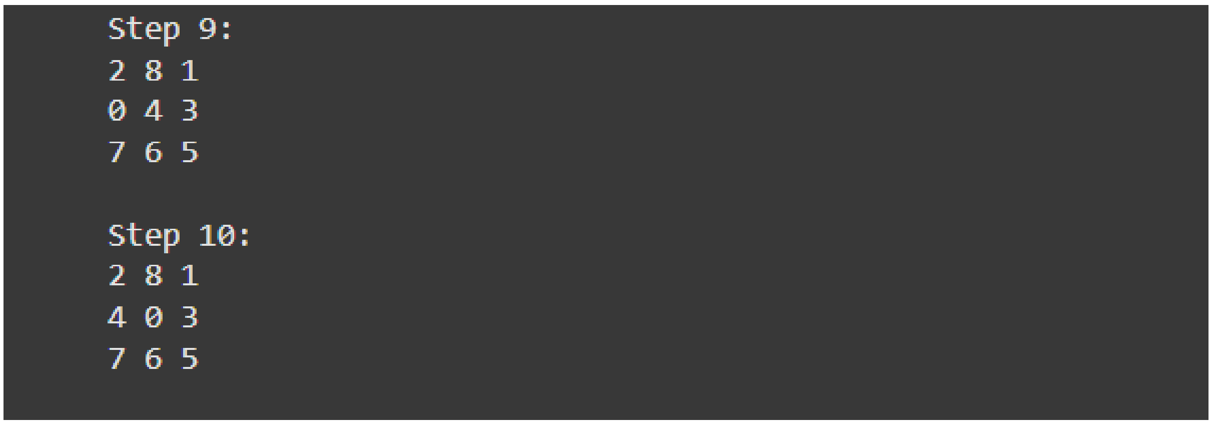


```
if __name__ == "__main__":
    initial_state = [
        [1, 2, 3],
        [8, 0, 4],
        [7, 6, 5]
    ]
    goal_state = [
        [2, 8, 1],
        [4, 0, 3],
        [7, 6, 5]
    ]

    print("Solving with Manhattan Distance:")
    a_star_solver(initial_state, goal_state, manhattan_distance)
```

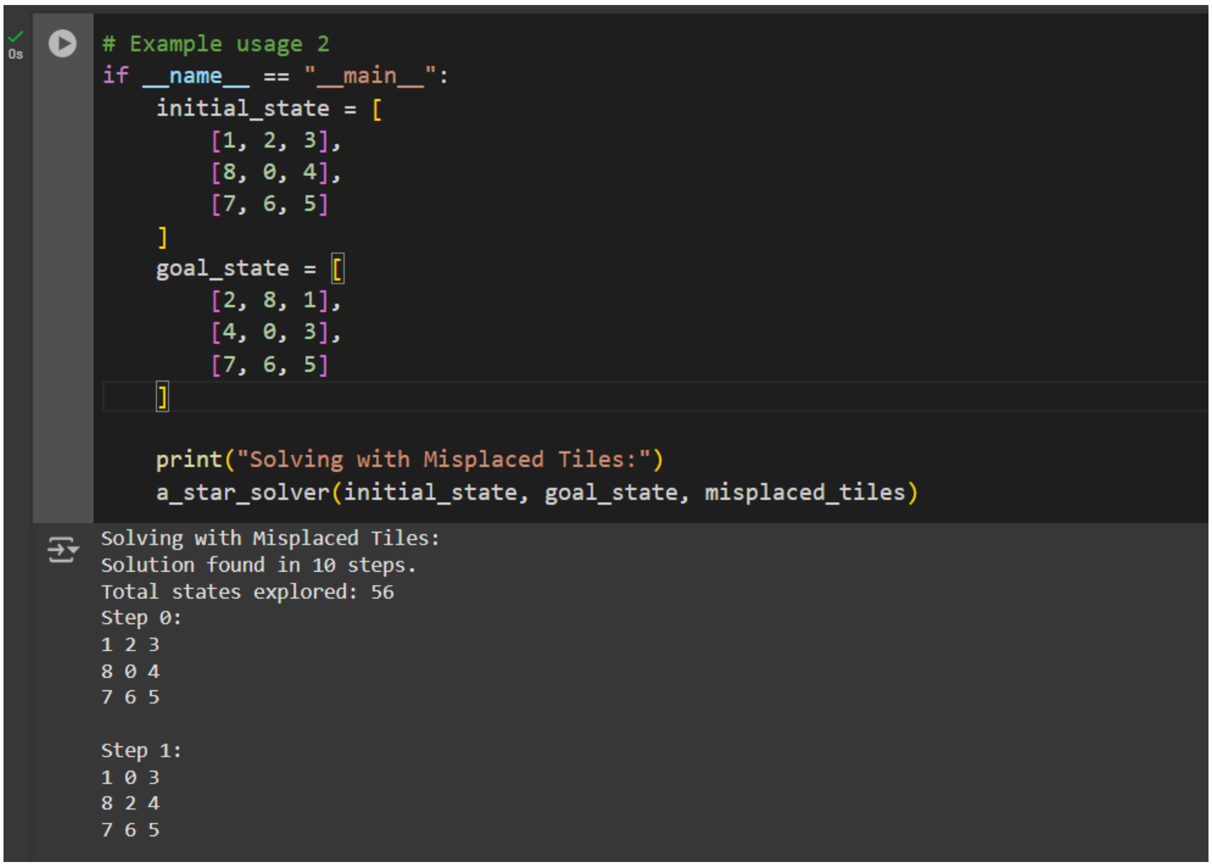
```
Solving with Manhattan Distance:
Solution found in 10 steps.
Total states explored: 27
Step 0:
1 2 3
8 0 4
7 6 5

Step 1:
1 0 3
8 2 4
7 6 5
```



```
Step 2:
0 1 3
8 2 4
7 6 5

Step 3:
8 1 3
0 2 4
7 6 5

Step 4:
8 1 3
2 0 4
7 6 5

Step 5:
8 1 3
2 4 0
7 6 5

Step 6:
8 1 0
2 4 3
7 6 5

Step 7:
8 0 1
2 4 3
7 6 5

Step 8:
0 8 1
2 4 3
7 6 5
```

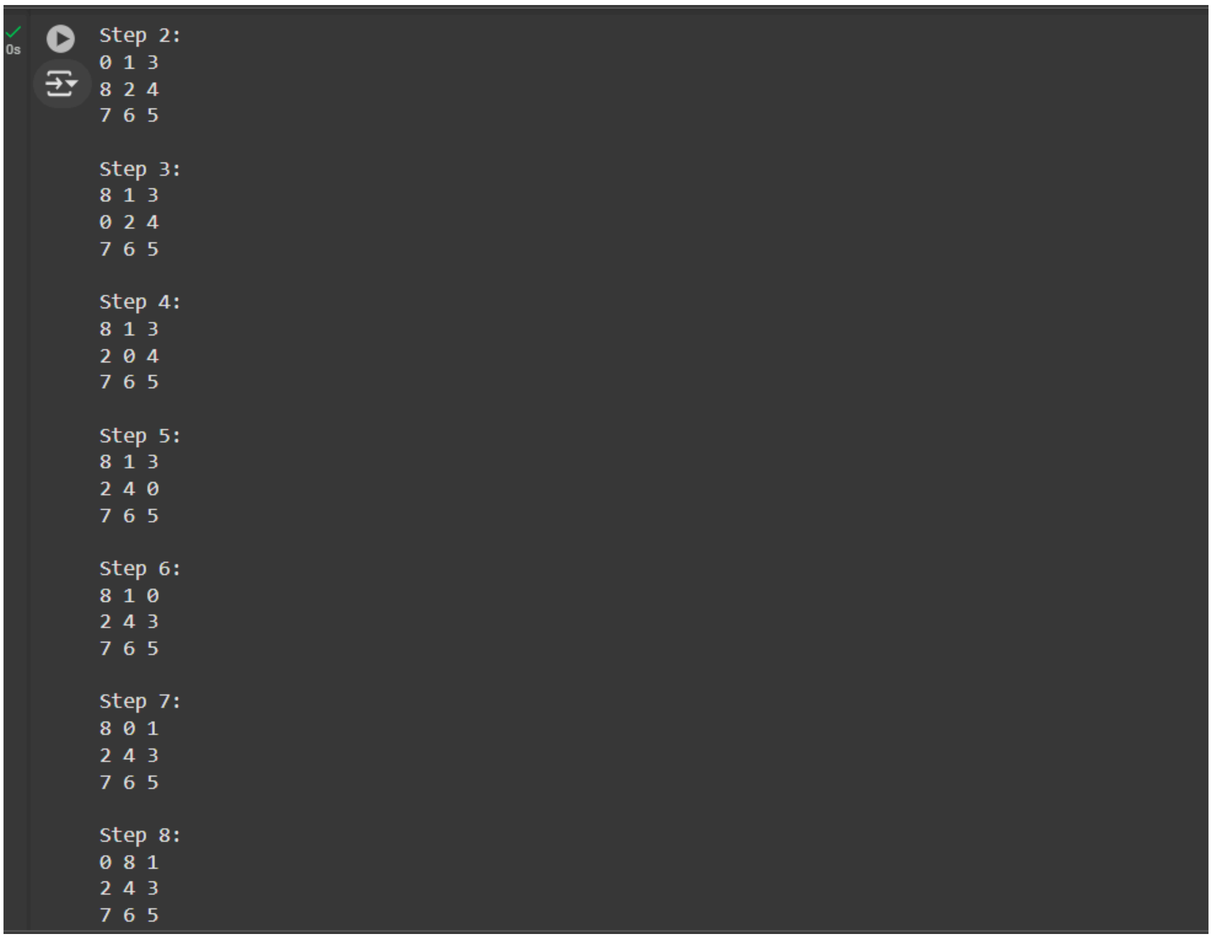


```
Step 9:
2 8 1
0 4 3
7 6 5

Step 10:
2 8 1
4 0 3
7 6 5
```

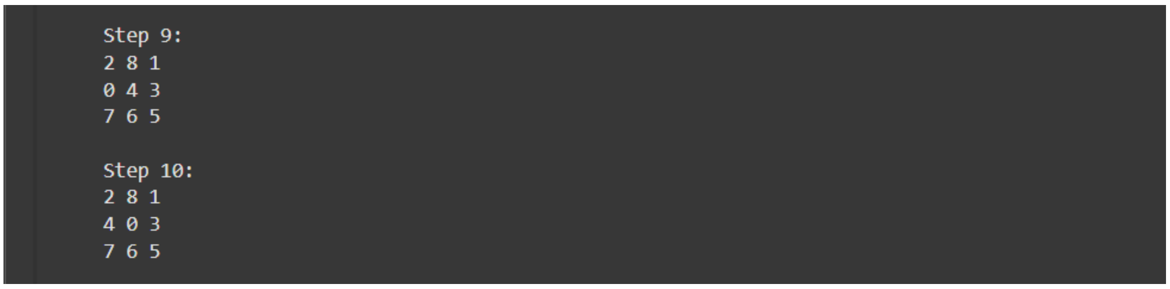


```python
# Example usage 2
if __name__ == "__main__":
    initial_state = [
        [1, 2, 3],
        [8, 0, 4],
        [7, 6, 5]
    ]
    goal_state = [
        [2, 8, 1],
        [4, 0, 3],
        [7, 6, 5]
    ]

    print("Solving with Misplaced Tiles:")
    a_star_solver(initial_state, goal_state, misplaced_tiles)
```

```
Solving with Misplaced Tiles:
Solution found in 10 steps.
Total states explored: 56
Step 0:
1 2 3
8 0 4
7 6 5

Step 1:
1 0 3
8 2 4
7 6 5
```



```
Step 2:
0 1 3
8 2 4
7 6 5

Step 3:
8 1 3
0 2 4
7 6 5

Step 4:
8 1 3
2 0 4
7 6 5

Step 5:
8 1 3
2 4 0
7 6 5

Step 6:
8 1 0
2 4 3
7 6 5

Step 7:
8 0 1
2 4 3
7 6 5

Step 8:
0 8 1
2 4 3
7 6 5
```



```
Step 9:
2 8 1
0 4 3
7 6 5

Step 10:
2 8 1
4 0 3
7 6 5
```