**Algorithm:**

To implement the A\* algorithm for the 8-puzzle problem, we will define two heuristics, h(n):

1. Number of Misplaced Tiles: This is the number of tiles that are not in their goal positions.

2. Manhattan Distance: The sum of the absolute differences between the current position and the goal position of each tile.

The A\* algorithm will use f(n) = g(n) + h(n), where:

g(n) is the depth of the node (the cost to reach the current state).

h(n) is the heuristic, either misplaced tiles or Manhattan distance.

Algorithm for A\*:

1. Initialize:

Start with the initial state of the puzzle.

Use a priority queue to hold the nodes, ordered by f(n) = g(n) + h(n).

Add the start state to the priority queue with g(n) = 0 and h(n) depending on the chosen heuristic.

Keep a set of visited nodes to avoid re-exploring nodes.

2. Expand Nodes:

While the priority queue is not empty:

Remove the node with the lowest f(n) from the queue.

If this node is the goal, return the solution.

Generate all possible child nodes (neighboring states) by moving the blank tile.

For each child:

Calculate g(n) (increment depth by 1 from the parent).

Calculate h(n) using the chosen heuristic.

If the child has not been visited or if it has a better f(n) than before, add it to the queue.

3. Termination:

If the goal is found, return the path and total cost.

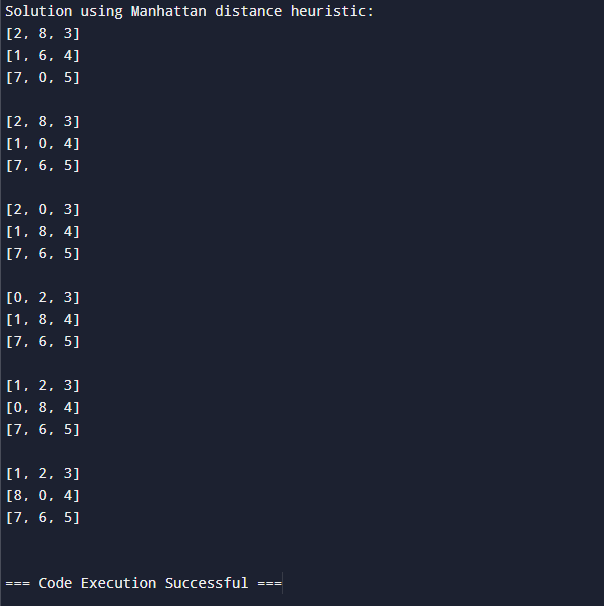
If no solution is found, return failure.

**PseudoCode:**

A\* Algorithm (initial\_state, goal\_state, heuristic)  
    open\_list ← priority\_queue  
    closed\_set ← empty set  
    g ← 0  
    f ← g + heuristic(initial\_state, goal\_state)  
     
    open\_list.push((f, g, initial\_state, []))  # (f, g, state, path)  
  
    while open\_list is not empty:  
        (f, g, current\_state, path) ← open\_list.pop()  
  
        if current\_state == goal\_state:  
            return path + [current\_state]  
  
        closed\_set.add(current\_state)  
  
        for each neighbor in get\_neighbors(current\_state):  
            if neighbor in closed\_set:  
                continue  
             
            new\_g ← g + 1  
            new\_f ← new\_g + heuristic(neighbor, goal\_state)  
            open\_list.push((new\_f, new\_g, neighbor, path + [current\_state]))  
  
    return None

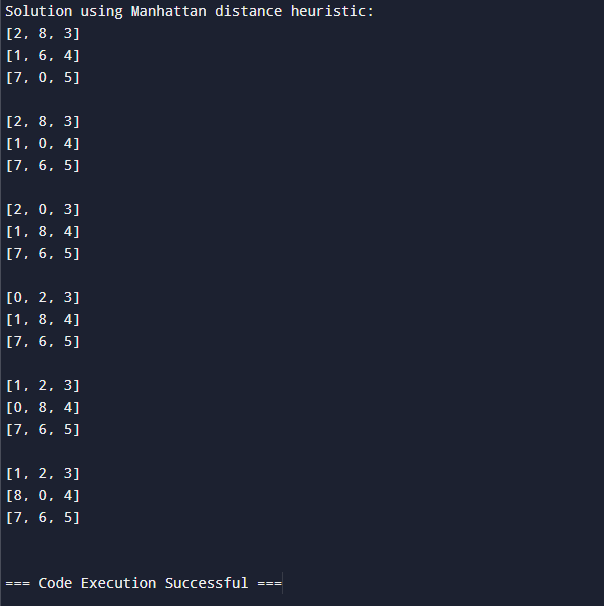
**Misplace Tiles Code:**

import heapq  
  
# Define the goal state for the puzzle  
goal\_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]  
  
# Helper function to calculate the misplaced tiles heuristic  
def misplaced\_tiles(state, goal\_state):  
    """Calculate the number of misplaced tiles."""  
    count = 0  
    for i in range(3):  
        for j in range(3):  
            if state[i][j] != 0 and state[i][j] != goal\_state[i][j]:  
                count += 1  
    return count  
  
def is\_goal(state):  
    """Check if the current state is the goal state."""  
    return state == goal\_state  
  
def get\_neighbors(state):  
    """Generate neighbors for a given state."""  
    neighbors = []  
    # Find the position of the empty space (0)  
    x, y = [(ix, iy) for ix, row in enumerate(state) for iy, i in enumerate(row) if i == 0][0]  
    # Possible moves: up, down, left, right  
    moves = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]  
  
    for move\_x, move\_y in moves:  
        if 0 <= move\_x < 3 and 0 <= move\_y < 3:  
            # Create a copy of the current state and swap the empty space  
            new\_state = [row[:] for row in state]  
            new\_state[x][y], new\_state[move\_x][move\_y] = new\_state[move\_x][move\_y], new\_state[x][y]  
            neighbors.append(new\_state)  
    return neighbors  
  
# A\* algorithm using misplaced tiles heuristic  
def a\_star\_misplaced(initial\_state):  
    """A\* algorithm implementation for 8-puzzle problem with misplaced tiles heuristic."""  
    open\_list = []  
    closed\_set = set()  
  
    # Initial cost  
    g = 0  
    f = g + misplaced\_tiles(initial\_state, goal\_state)  
  
    # Add the initial state to the open list  
    heapq.heappush(open\_list, (f, g, initial\_state, []))  
  
    while open\_list:  
        f, g, current\_state, path = heapq.heappop(open\_list)  
  
        if is\_goal(current\_state):  
            return path + [current\_state]  
  
        closed\_set.add(tuple(map(tuple, current\_state)))  
  
        for neighbor in get\_neighbors(current\_state):  
            if tuple(map(tuple, neighbor)) in closed\_set:  
                continue  
  
            new\_g = g + 1  
            new\_f = new\_g + misplaced\_tiles(neighbor, goal\_state)  
            heapq.heappush(open\_list, (new\_f, new\_g, neighbor, path + [current\_state]))  
  
    return None  
  
# Helper function to print a state  
def print\_state(state):  
    for row in state:  
        print(row)  
    print()  
  
# Initial state  
initial\_state = [[2, 8, 3], [1, 6, 4], [7, 0, 5]]  
  
# Run the A\* algorithm with misplaced tiles heuristic  
print("Solution using misplaced tiles heuristic:")  
solution\_misplaced = a\_star\_misplaced(initial\_state)  
if solution\_misplaced:  
    for step in solution\_misplaced:  
        print\_state(step)  
else:  
    print("No solution found using misplaced tiles heuristic.")

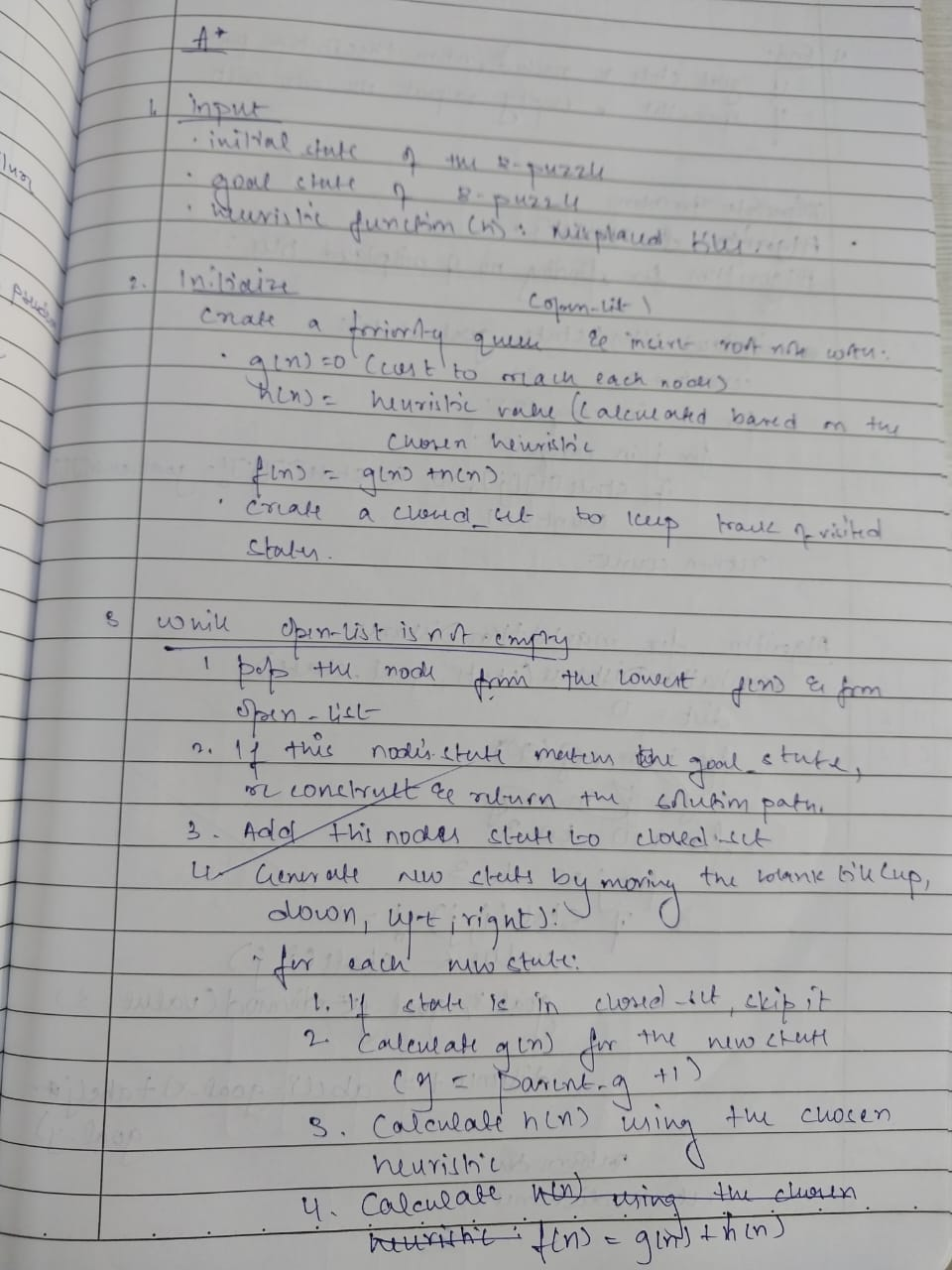


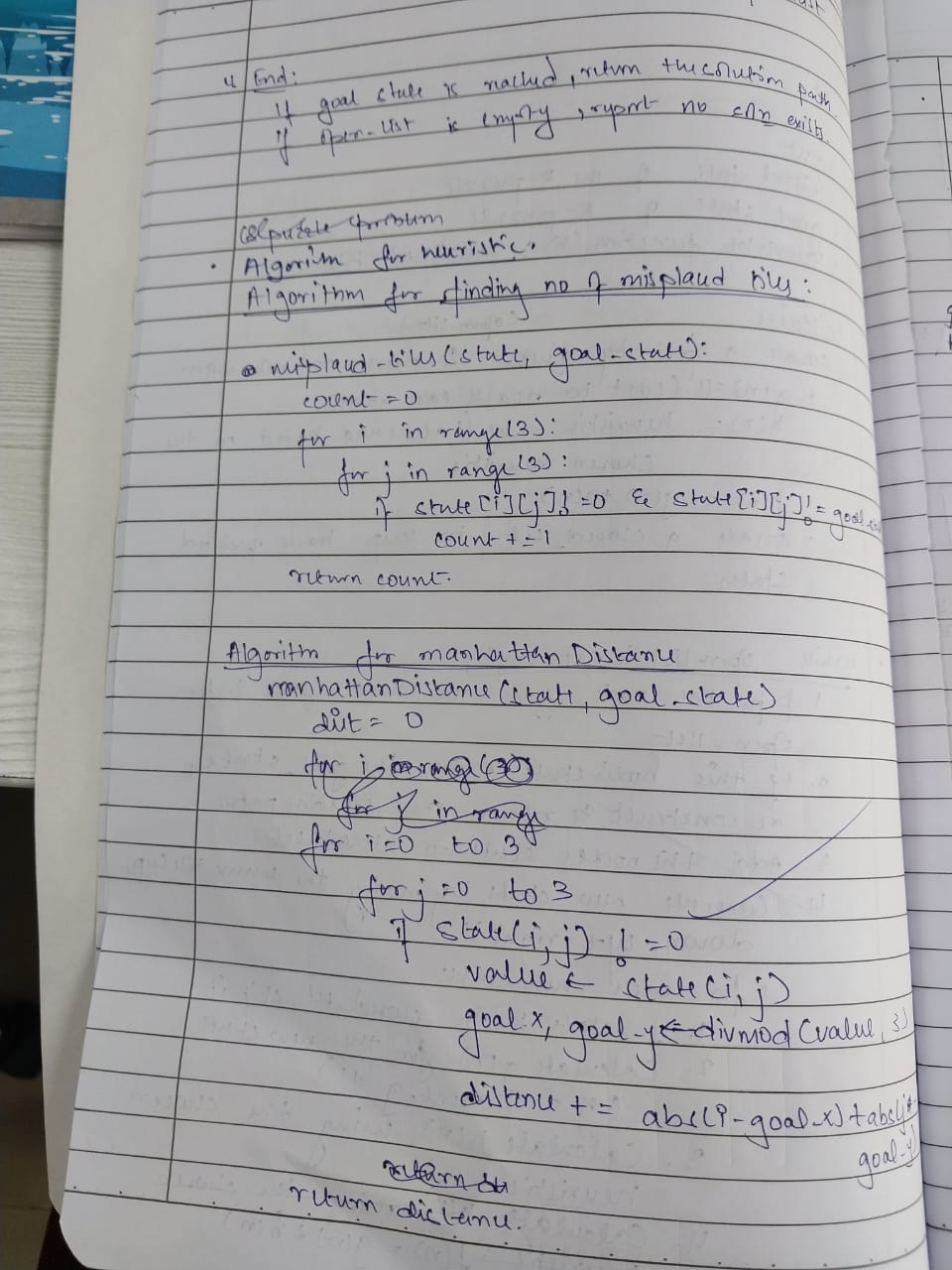
**Manhattan distance**

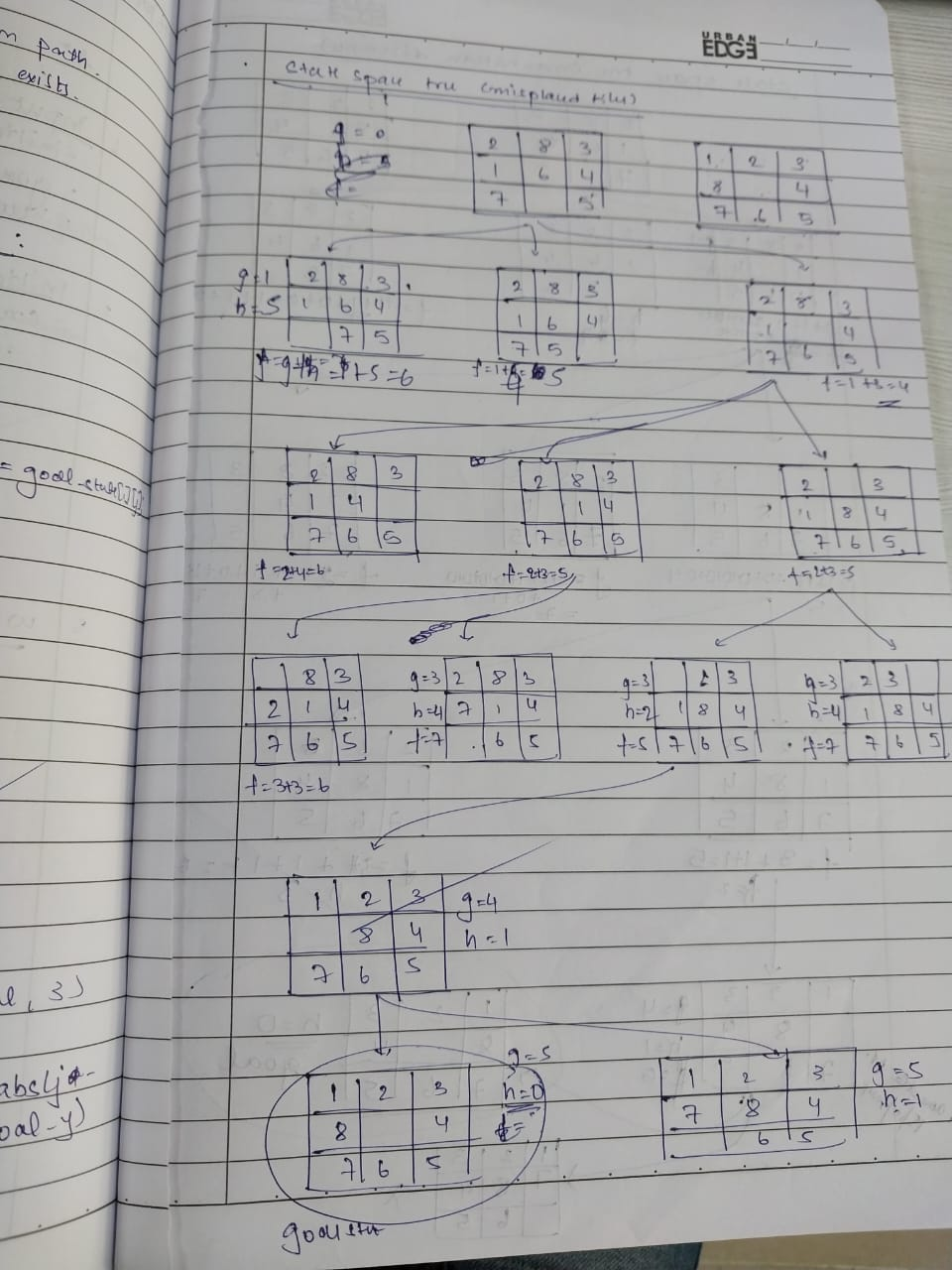
import heapq  
  
# Define the goal state for the puzzle  
goal\_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]  
  
# Helper function to calculate the Manhattan distance heuristic  
def manhattan\_distance(state, goal\_state):  
    """Calculate the Manhattan distance heuristic."""  
    distance = 0  
    for i in range(3):  
        for j in range(3):  
            if state[i][j] != 0:  
                # Get current tile number (1-8)  
                value = state[i][j]  
                # Find its goal position in the goal state  
                goal\_x, goal\_y = divmod(value, 3)  
                # Calculate Manhattan distance  
                distance += abs(i - goal\_x) + abs(j - goal\_y)  
    return distance  
  
def is\_goal(state):  
    """Check if the current state is the goal state."""  
    return state == goal\_state  
  
def get\_neighbors(state):  
    """Generate neighbors for a given state."""  
    neighbors = []  
    # Find the position of the empty space (0)  
    x, y = [(ix, iy) for ix, row in enumerate(state) for iy, i in enumerate(row) if i == 0][0]  
    # Possible moves: up, down, left, right  
    moves = [(x-1, y), (x+1, y), (x, y-1), (x, y+1)]  
  
    for move\_x, move\_y in moves:  
        if 0 <= move\_x < 3 and 0 <= move\_y < 3:  
            # Create a copy of the current state and swap the empty space  
            new\_state = [row[:] for row in state]  
            new\_state[x][y], new\_state[move\_x][move\_y] = new\_state[move\_x][move\_y], new\_state[x][y]  
            neighbors.append(new\_state)  
    return neighbors  
  
# A\* algorithm using Manhattan distance heuristic  
def a\_star\_manhattan(initial\_state):  
    """A\* algorithm implementation for 8-puzzle problem with Manhattan distance heuristic."""  
    open\_list = []  
    closed\_set = set()  
  
    # Initial cost  
    g = 0  
    f = g + manhattan\_distance(initial\_state, goal\_state)  
  
    # Add the initial state to the open list  
    heapq.heappush(open\_list, (f, g, initial\_state, []))  
  
    while open\_list:  
        f, g, current\_state, path = heapq.heappop(open\_list)  
  
        if is\_goal(current\_state):  
            return path + [current\_state]  
  
        closed\_set.add(tuple(map(tuple, current\_state)))  
  
        for neighbor in get\_neighbors(current\_state):  
            if tuple(map(tuple, neighbor)) in closed\_set:  
                continue  
  
            new\_g = g + 1  
            new\_f = new\_g + manhattan\_distance(neighbor, goal\_state)  
            heapq.heappush(open\_list, (new\_f, new\_g, neighbor, path + [current\_state]))  
  
    return None  
  
# Helper function to print a state  
def print\_state(state):  
    for row in state:  
        print(row)  
    print()  
  
# Initial state  
initial\_state = [[2, 8, 3], [1, 6, 4], [7, 0, 5]]  
  
# Run the A\* algorithm with Manhattan distance heuristic  
print("Solution using Manhattan distance heuristic:")  
solution\_manhattan = a\_star\_manhattan(initial\_state)  
if solution\_manhattan:  
    for step in solution\_manhattan:  
        print\_state(step)  
else:  
    print("No solution found using Manhattan distance heuristic.")

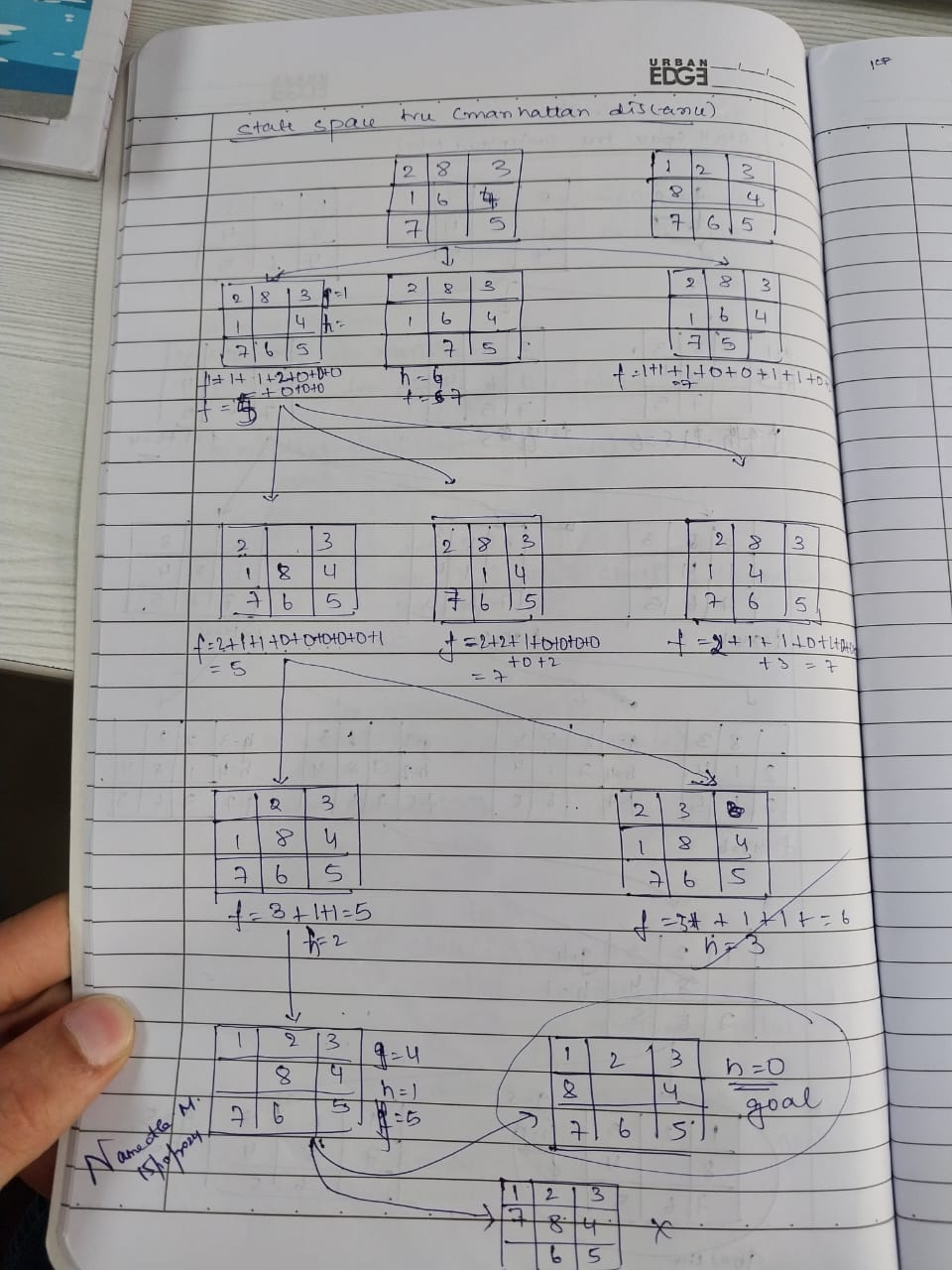


Observation:







s