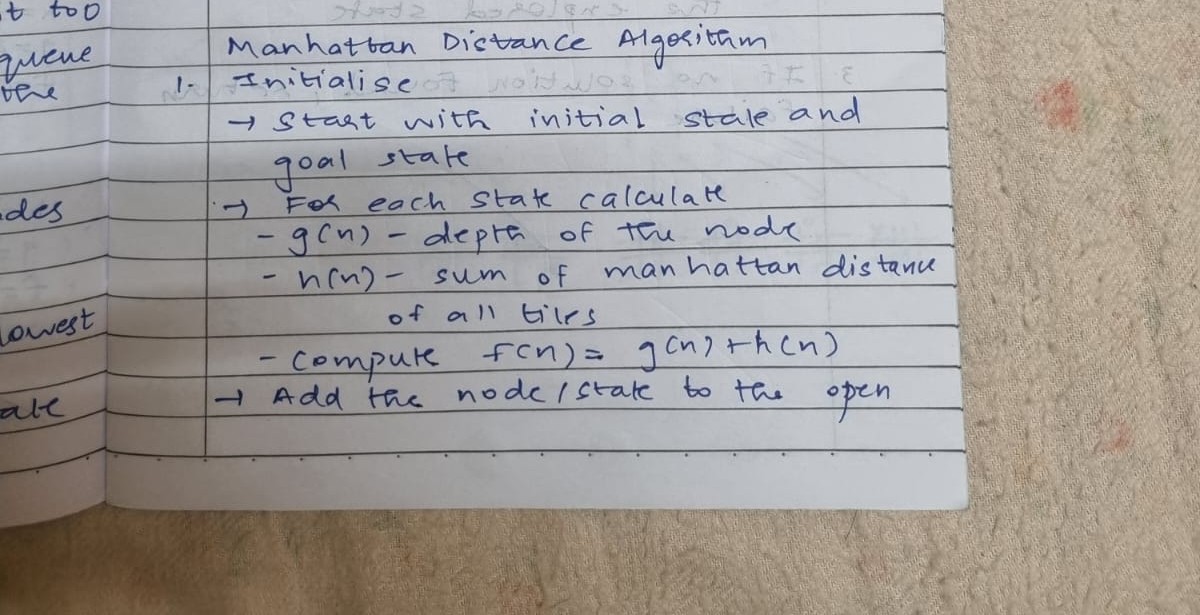
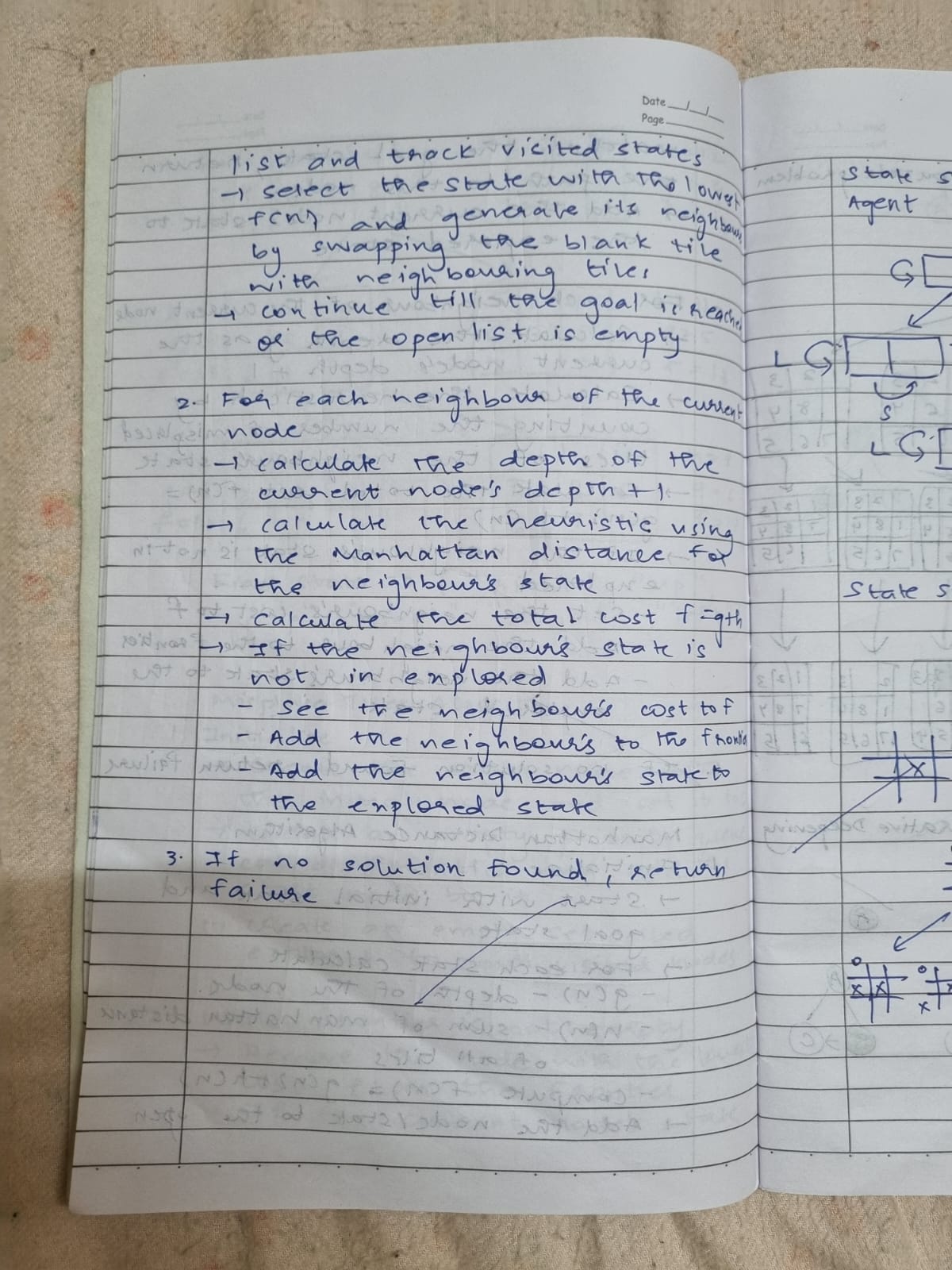
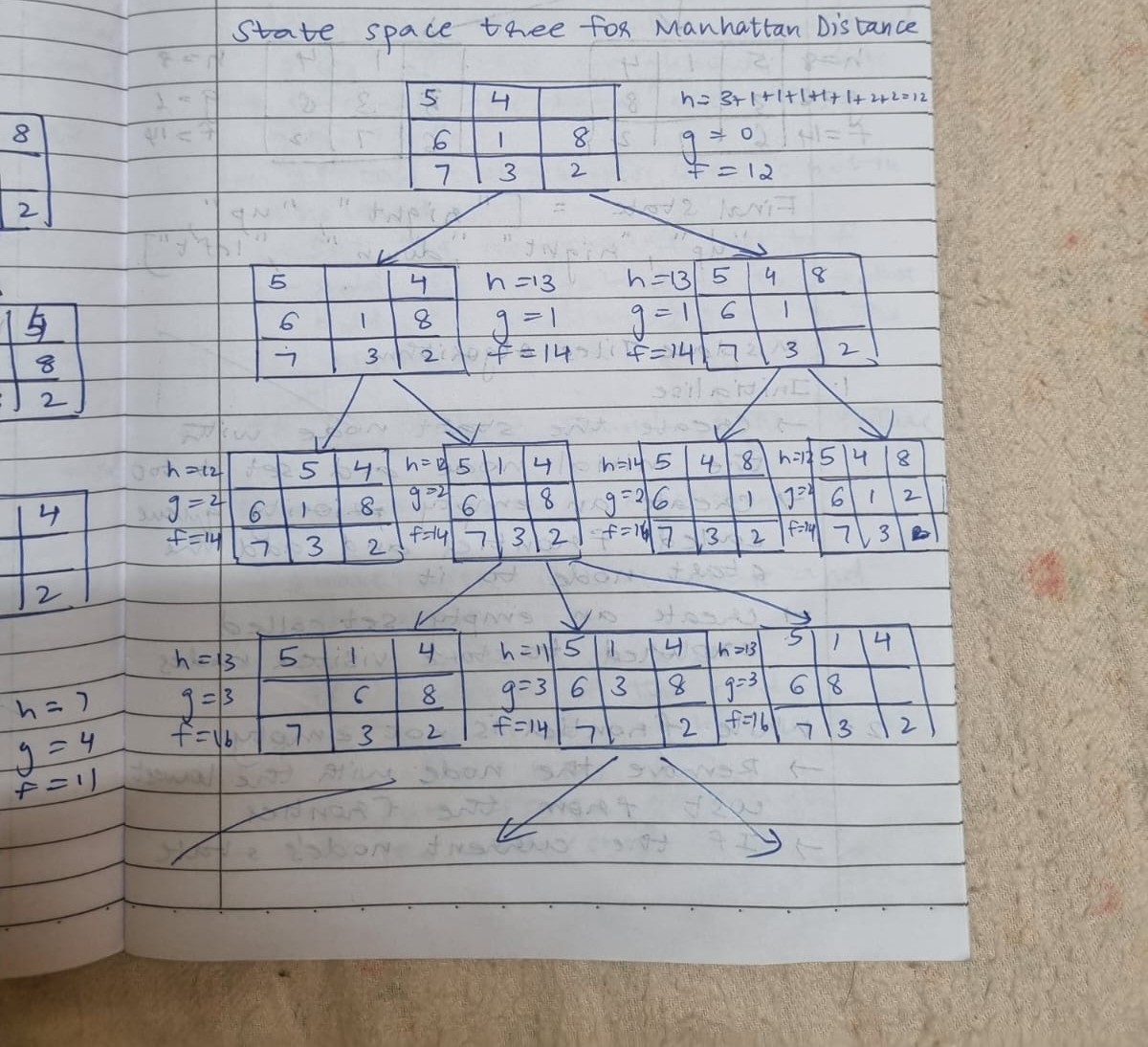
**A\* using Manhattan Distance**

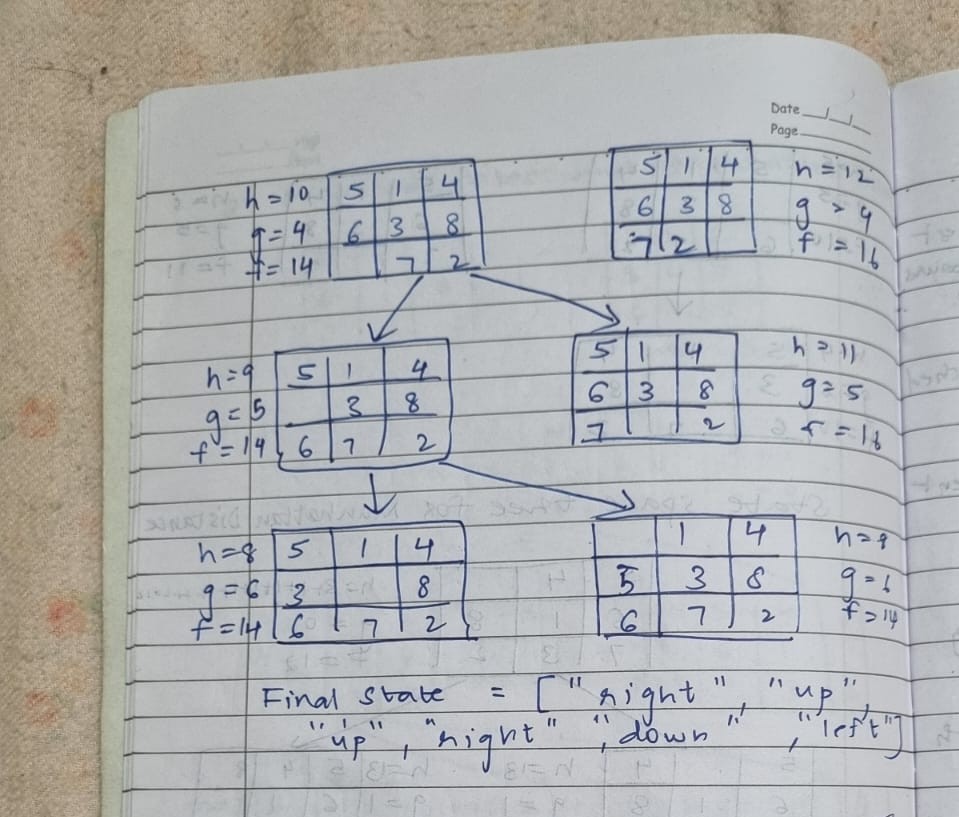
**Algorithm:**

****

****

**State Space Tree:**

****

****

**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 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.")

**Output:**

Solution using Manhattan distance heuristic:

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

[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]