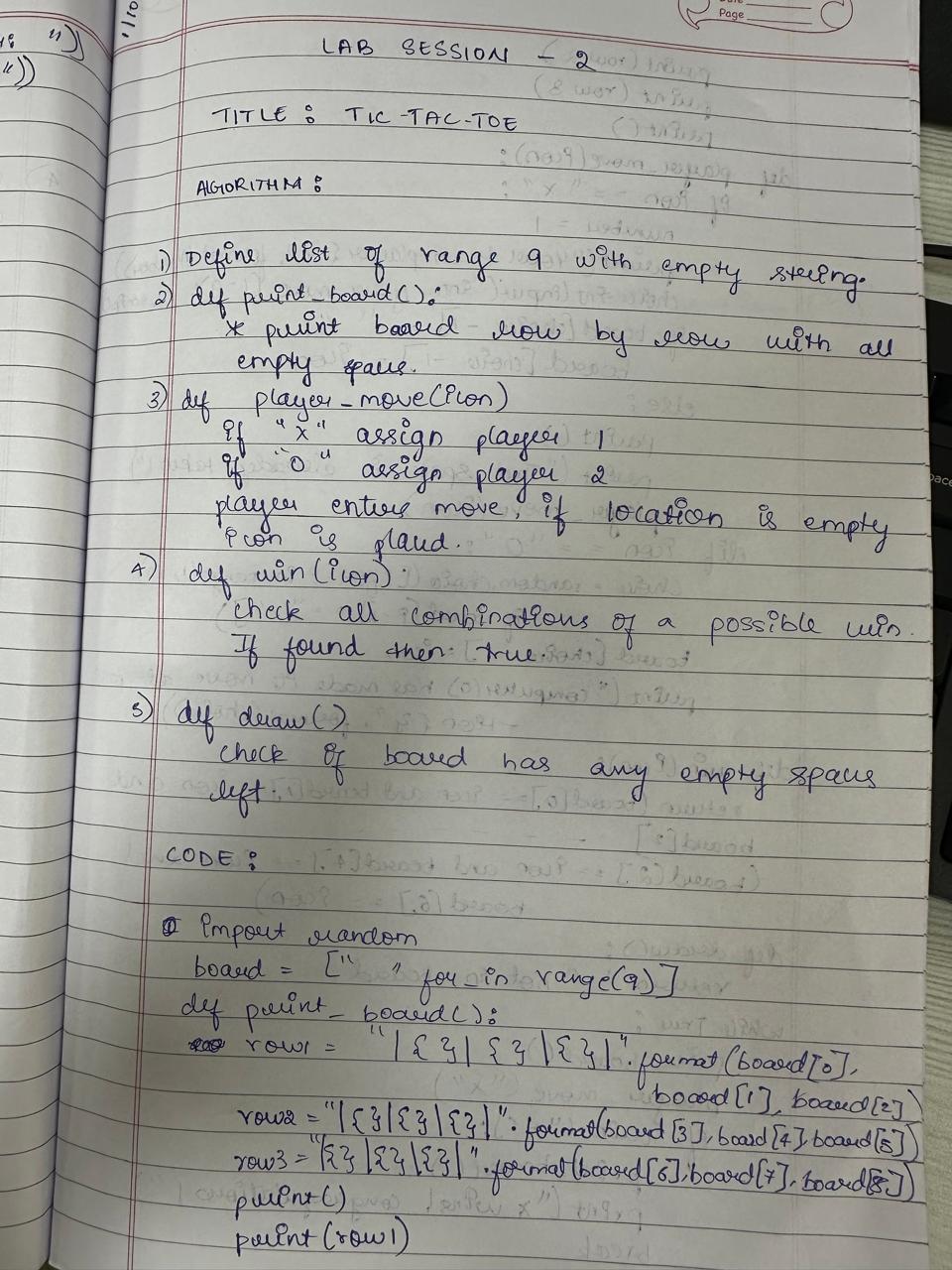
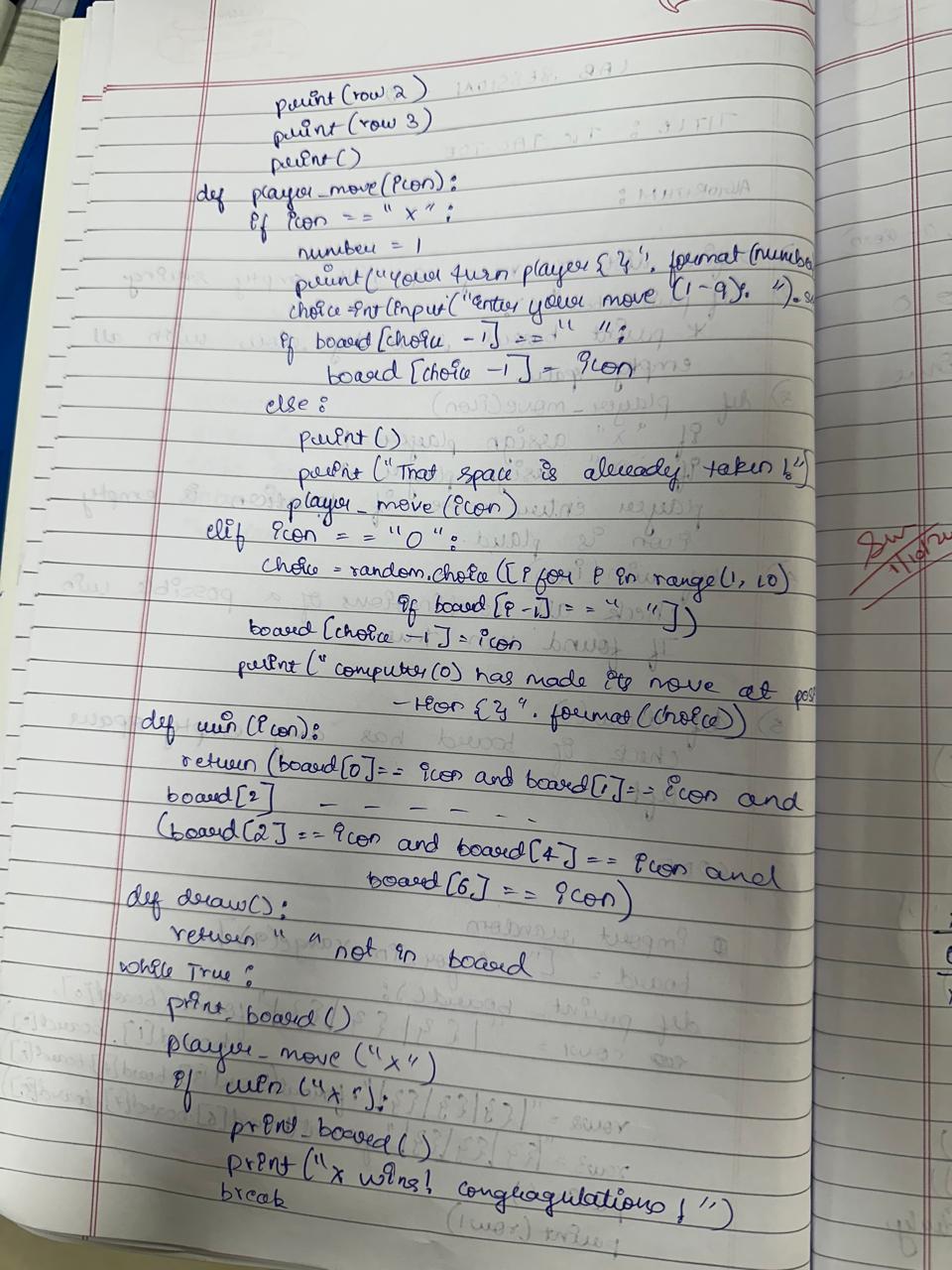
**AI LAB REPORT**

**LAB - 01**

**Program 1 : Tic Tac Toe**

**OBSERVATION:**





**CODE:**

import random

board = [" " for \_ in range(9)]

def print\_board():

row1 = "| {} | {} | {} |".format(board[0], board[1], board[2])

row2 = "| {} | {} | {} |".format(board[3], board[4], board[5])

row3 = "| {} | {} | {} |".format(board[6], board[7], board[8])

print()

print(row1)

print(row2)

print(row3)

print()

def player\_move(icon):

if icon == "X":

number = 1

print("Your turn player {}".format(number))

choice = int(input("Enter your move (1-9): ").strip())

if board[choice - 1] == " ":

board[choice - 1] = icon

else:

print()

print("That space is already taken!")

player\_move(icon)

elif icon == "O":

choice = random.choice([i for i in range(1, 10) if board[i - 1] == " "])

board[choice - 1] = icon

print("Computer (O) has made its move at position {}".format(choice))

def win(icon):

return (board[0] == icon and board[1] == icon and board[2] == icon) or \

(board[3] == icon and board[4] == icon and board[5] == icon) or \

(board[6] == icon and board[7] == icon and board[8] == icon) or \

(board[0] == icon and board[3] == icon and board[6] == icon) or \

(board[1] == icon and board[4] == icon and board[7] == icon) or \

(board[2] == icon and board[5] == icon and board[8] == icon) or \

(board[0] == icon and board[4] == icon and board[8] == icon) or \

(board[2] == icon and board[4] == icon and board[6] == icon)

def draw():

return " " not in board

while True:

print\_board()

player\_move("X")

if win("X"):

print\_board()

print("X wins! Congratulations!")

break

elif draw():

print\_board()

print("It's a draw!")

break

player\_move("O")

if win("O"):

print\_board()

print("O wins! Congratulations!")

break

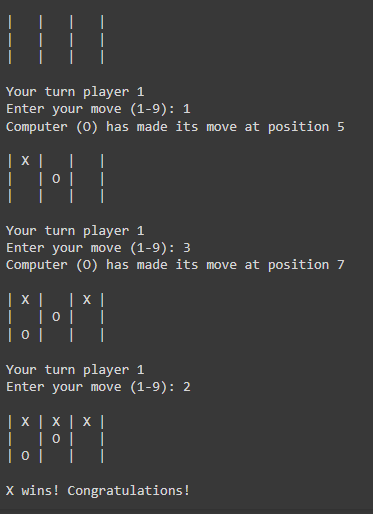
elif draw():

print\_board()

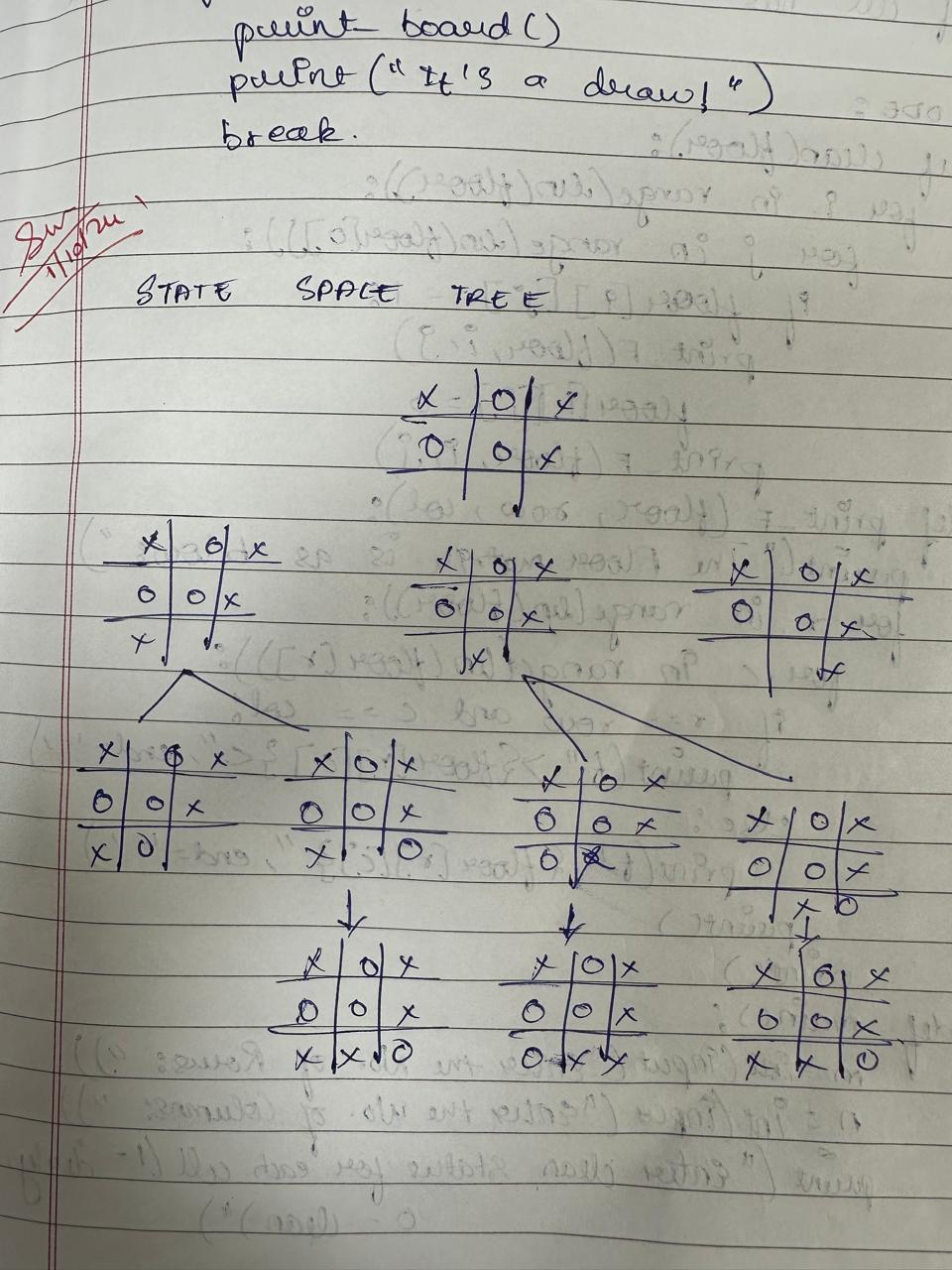
print("It's a draw!")

break

**OUTPUT:**

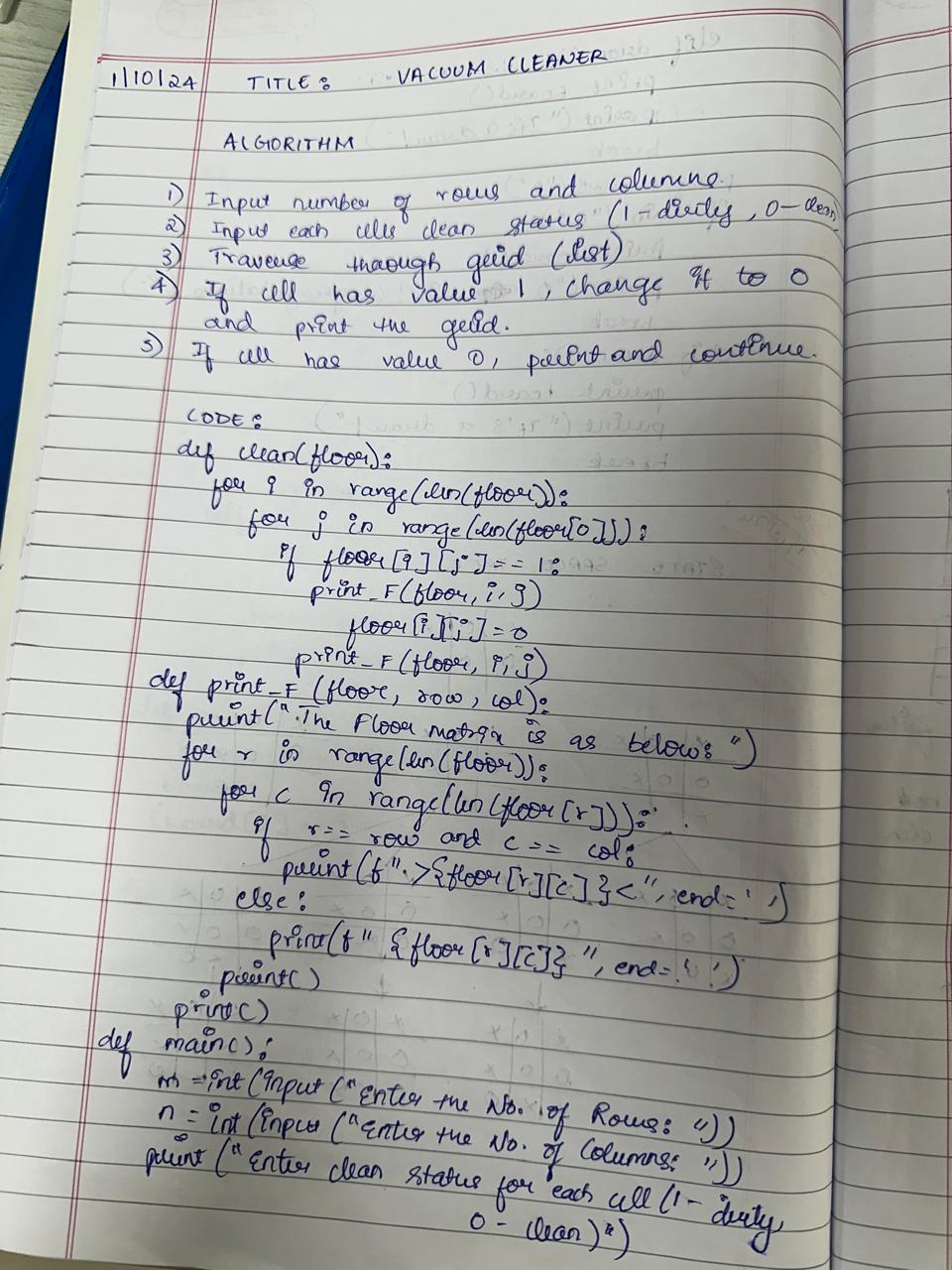


**STATE SPACE TREE:**



**Program 2 : Vacuum Cleaner Agent**

**OBSERVATION:**



**CODE:**

def clean(floor):

for i in range(len(floor)):

for j in range(len(floor[0])):

if floor[i][j] == 1:

print\_F(floor, i, j)

floor[i][j] = 0

print\_F(floor, i, j)

def print\_F(floor, row, col):

print("The Floor matrix is as below:")

for r in range(len(floor)):

for c in range(len(floor[r])):

if r == row and c == col:

print(f" >{floor[r][c]}< ", end='')

else:

print(f" {floor[r][c]} ", end='')

print()

print()

def main():

m = int(input("Enter the No. of Rows: "))

n = int(input("Enter the No. of Columns: "))

print("Enter clean status for each cell (1 - dirty, 0 - clean)")

floor = [list(map(int, input().split())) for \_ in range(m)]

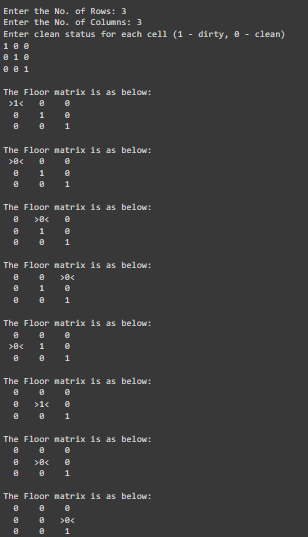
print()

clean(floor)

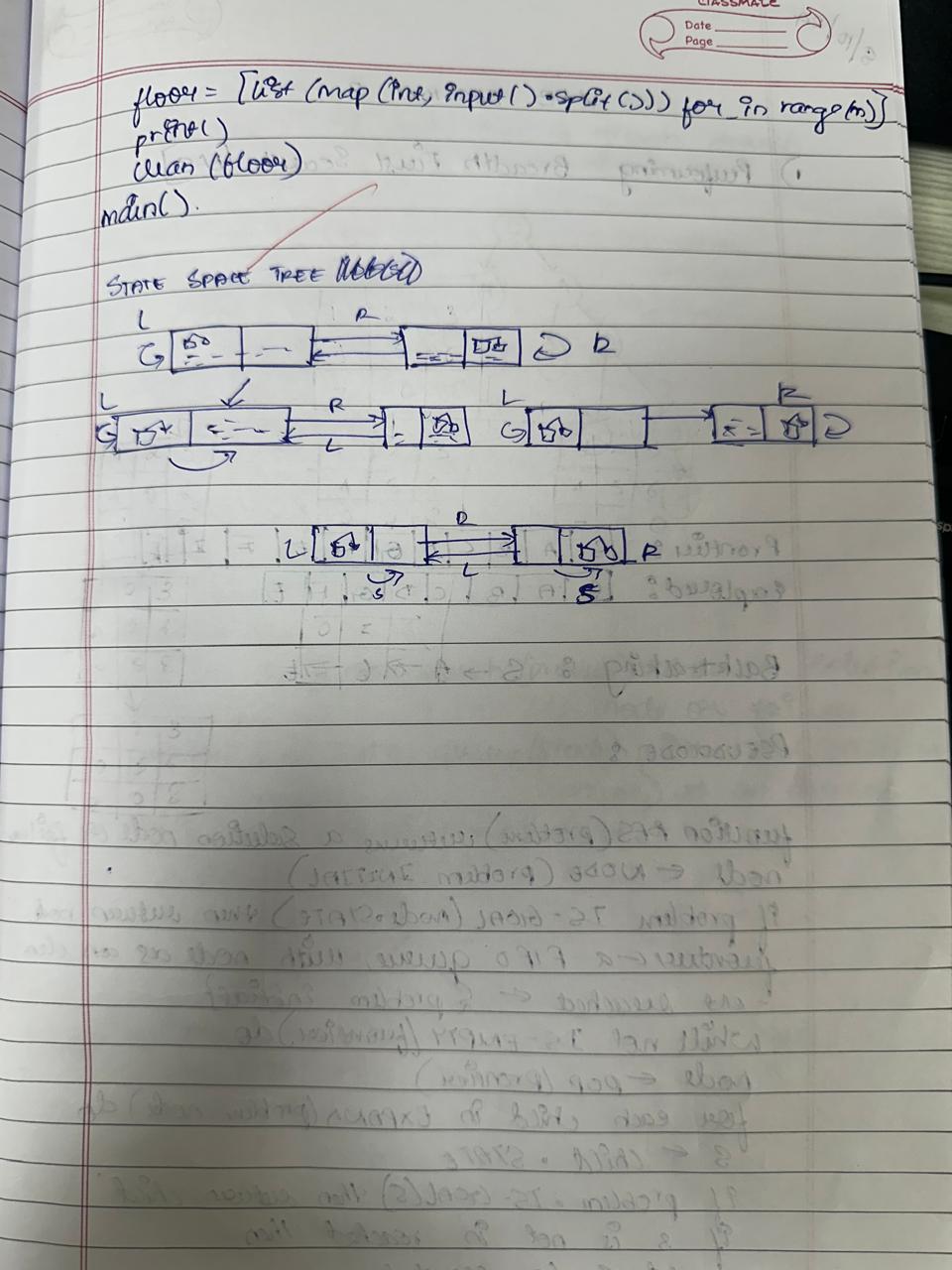
main()

\

**OUTPUT:**



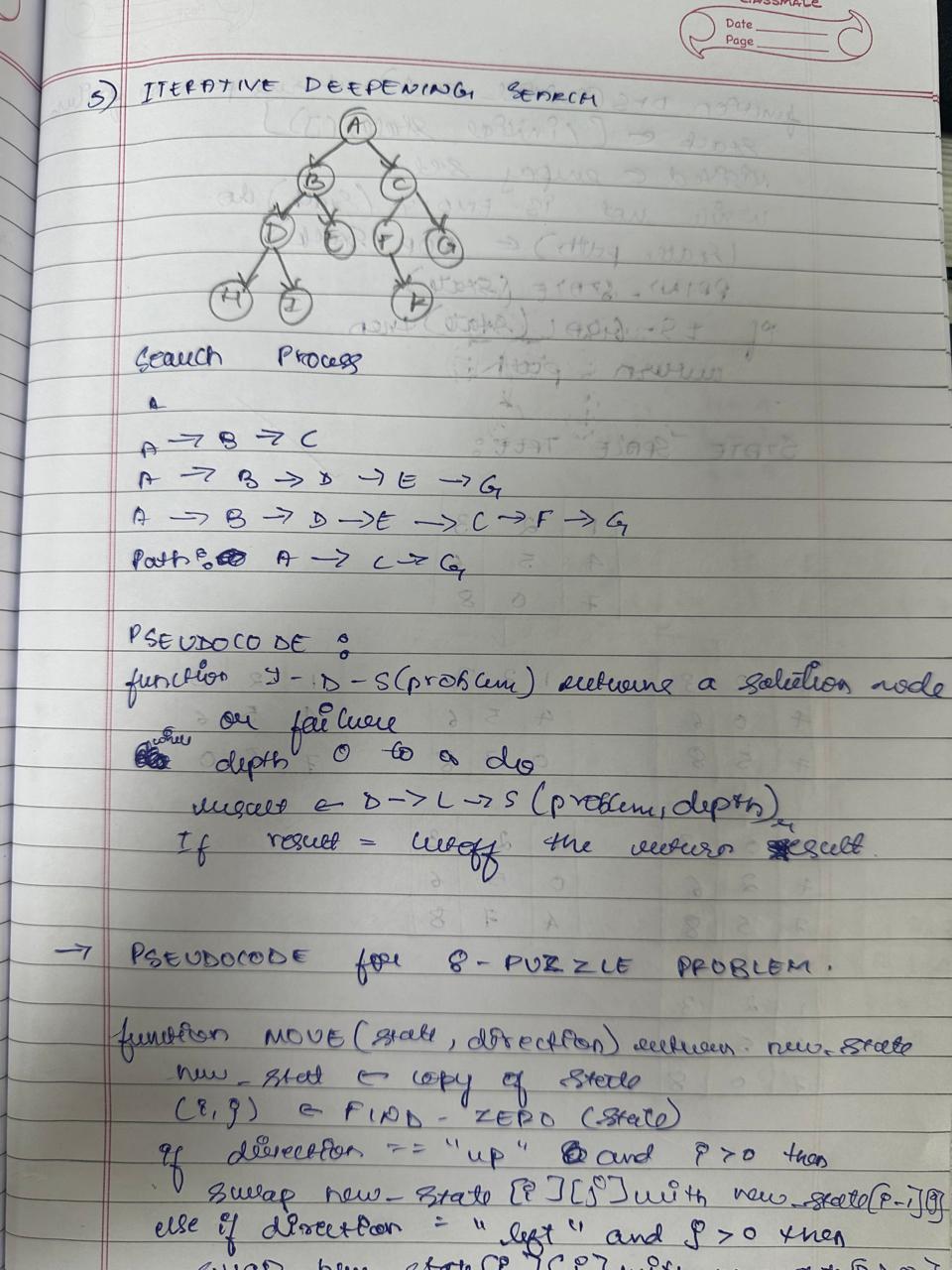
**SEARCH SPACE TREE:**



**LAB - 02**

**Program 1 : Iterative Deepening Search**

**OBSERVATION:**



**CODE:**

import copy

class Node:

def \_\_init\_\_(self, state, parent=None, action=None, depth=0):

self.state = state

self.parent = parent

self.action = action

self.depth = depth

def \_\_lt\_\_(self, other):

return self.depth < other.depth

def expand(self):

children = []

row, col = self.find\_blank()

possible\_actions = []

if row > 0: # Can move the blank tile up

possible\_actions.append('Up')

if row < 2: # Can move the blank tile down

possible\_actions.append('Down')

if col > 0: # Can move the blank tile left

possible\_actions.append('Left')

if col < 2: # Can move the blank tile right

possible\_actions.append('Right')

for action in possible\_actions:

new\_state = copy.deepcopy(self.state)

if action == 'Up':

new\_state[row][col], new\_state[row - 1][col] = new\_state[row - 1][col], new\_state[row][col]

elif action == 'Down':

new\_state[row][col], new\_state[row + 1][col] = new\_state[row + 1][col], new\_state[row][col]

elif action == 'Left':

new\_state[row][col], new\_state[row][col - 1] = new\_state[row][col - 1], new\_state[row][col]

elif action == 'Right':

new\_state[row][col], new\_state[row][col + 1] = new\_state[row][col + 1], new\_state[row][col]

children.append(Node(new\_state, self, action, self.depth + 1))

return children

def find\_blank(self):

for row in range(3):

for col in range(3):

if self.state[row][col] == 0:

return row, col

def depth\_limited\_search(node, goal\_state, limit):

if node.state == goal\_state:

return node

if node.depth >= limit:

return None

for child in node.expand():

result = depth\_limited\_search(child, goal\_state, limit)

if result is not None:

return result

return None

def iterative\_deepening\_search(initial\_state, goal\_state, max\_depth):

for depth in range(max\_depth):

result = depth\_limited\_search(Node(initial\_state), goal\_state, depth)

if result is not None:

return result

return None

def print\_solution(node):

path = []

while node is not None:

path.append((node.action, node.state))

node = node.parent

path.reverse()

for action, state in path:

if action:

print(f"Action: {action}")

for row in state:

print(row)

print()

initial\_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]

goal\_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

max\_depth = 20

solution = iterative\_deepening\_search(initial\_state, goal\_state, max\_depth)

if solution:

print("Solution found:")

print\_solution(solution)

else:

print("Solution not found.")

OUTPUT:

Solution found:

[1, 2, 3]

[0, 4, 6]

[7, 5, 8]

Action: Right

[1, 2, 3]

[4, 0, 6]

[7, 5, 8]

Action: Down

[1, 2, 3]

[4, 5, 6]

[7, 0, 8]

Action: Right

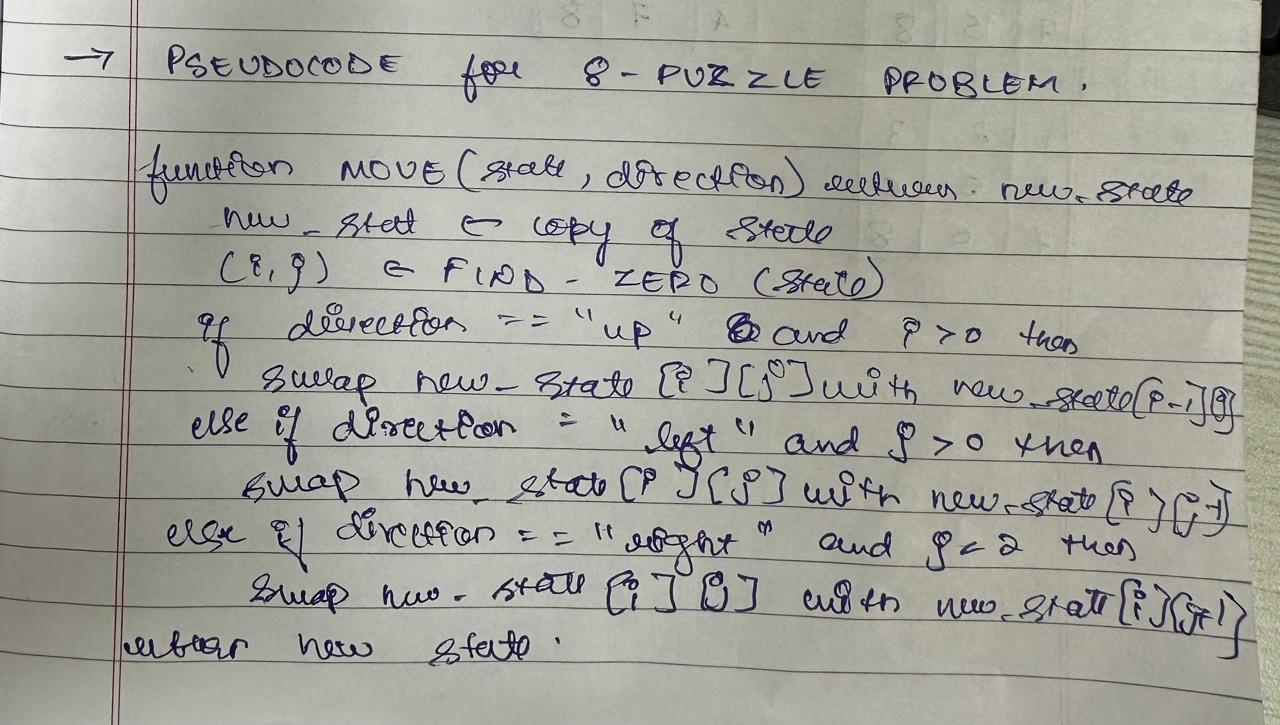
[1, 2, 3]

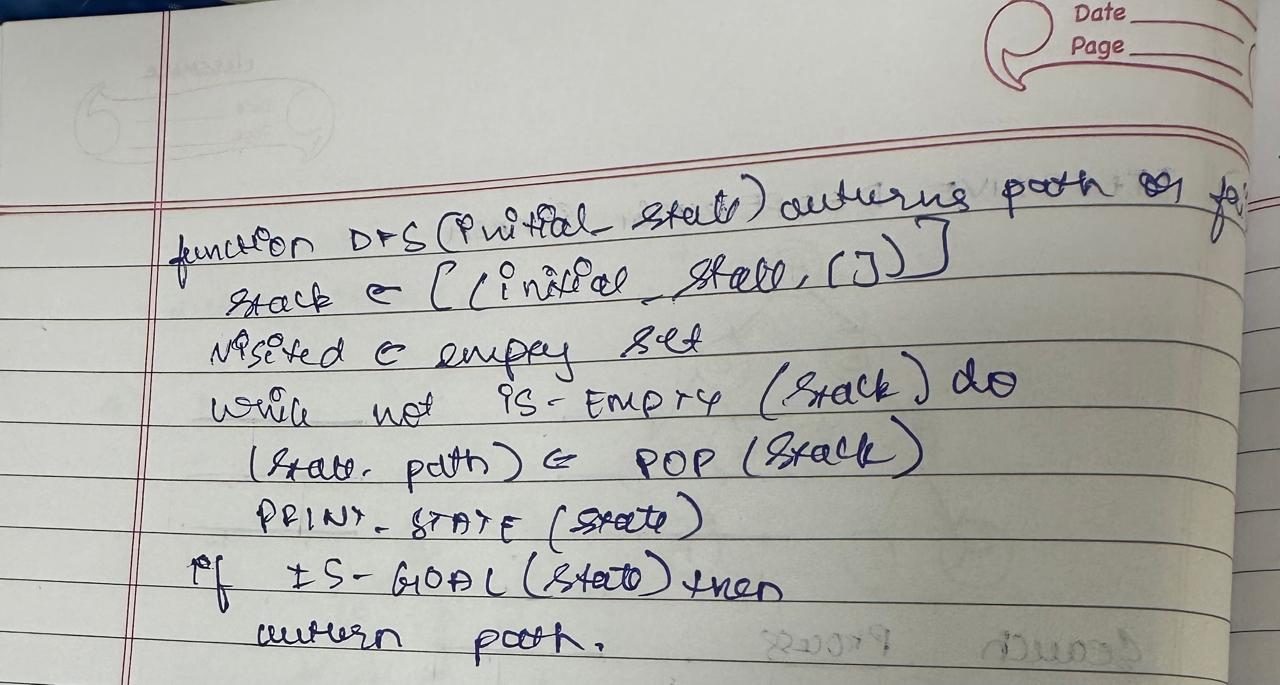
[4, 5, 6]

[7, 8, 0]

**Program 2 : 8 Puzzle Problem**

**OBSERVATION:**





**CODE:**

import copy

class Node:

def \_\_init\_\_(self, state, parent=None, action=None, path\_cost=0):

self.state = state

self.parent = parent

self.action = action

self.path\_cost = path\_cost

def \_\_lt\_\_(self, other):

return self.path\_cost < other.path\_cost

def expand(self):

children = []

row, col = self.find\_blank()

possible\_actions = []

if row > 0:

possible\_actions.append('Up')

if row < 2:

possible\_actions.append('Down')

if col > 0:

possible\_actions.append('Left')

if col < 2:

possible\_actions.append('Right')

for action in possible\_actions:

new\_state = copy.deepcopy(self.state)

if action == 'Up':

new\_state[row][col], new\_state[row - 1][col] = new\_state[row - 1][col], new\_state[row][col]

elif action == 'Down':

new\_state[row][col], new\_state[row + 1][col] = new\_state[row + 1][col], new\_state[row][col]

elif action == 'Left':

new\_state[row][col], new\_state[row][col - 1] = new\_state[row][col - 1], new\_state[row][col]

elif action == 'Right':

new\_state[row][col], new\_state[row][col + 1] = new\_state[row][col + 1], new\_state[row][col]

children.append(Node(new\_state, self, action, self.path\_cost + 1))

return children

def find\_blank(self):

for row in range(3):

for col in range(3):

if self.state[row][col] == 0:

return row, col

def depth\_first\_search(initial\_state, goal\_state):

frontier = [Node(initial\_state)]

explored = set()

while frontier:

node = frontier.pop()

if node.state == goal\_state:

return node

explored.add(tuple(map(tuple, node.state))) # Track visited states

for child in node.expand():

child\_state\_tuple = tuple(map(tuple, child.state))

if child\_state\_tuple not in explored and child not in frontier:

frontier.append(child)

return None

def print\_solution(node):

path = []

while node is not None:

path.append((node.action, node.state))

node = node.parent

path.reverse() # Reverse the path to start from the initial state

for action, state in path:

if action:

print(f"Action: {action}")

for row in state:

print(row)

print()

initial\_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]

goal\_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

solution = depth\_first\_search(initial\_state, goal\_state)

if solution:

print("Solution found:")

print\_solution(solution)

else:

print("Solution not found.")

**Output:**

Action: Left

[1, 2, 3]

[7, 4, 5]

[8, 0, 6]

Action: Left

[1, 2, 3]

[7, 4, 5]

[0, 8, 6]

Action: Up

[1, 2, 3]

[0, 4, 5]

[7, 8, 6]

Action: Right

[1, 2, 3]

[4, 0, 5]

[7, 8, 6]

Action: Right

[1, 2, 3]

[4, 5, 0]

[7, 8, 6]

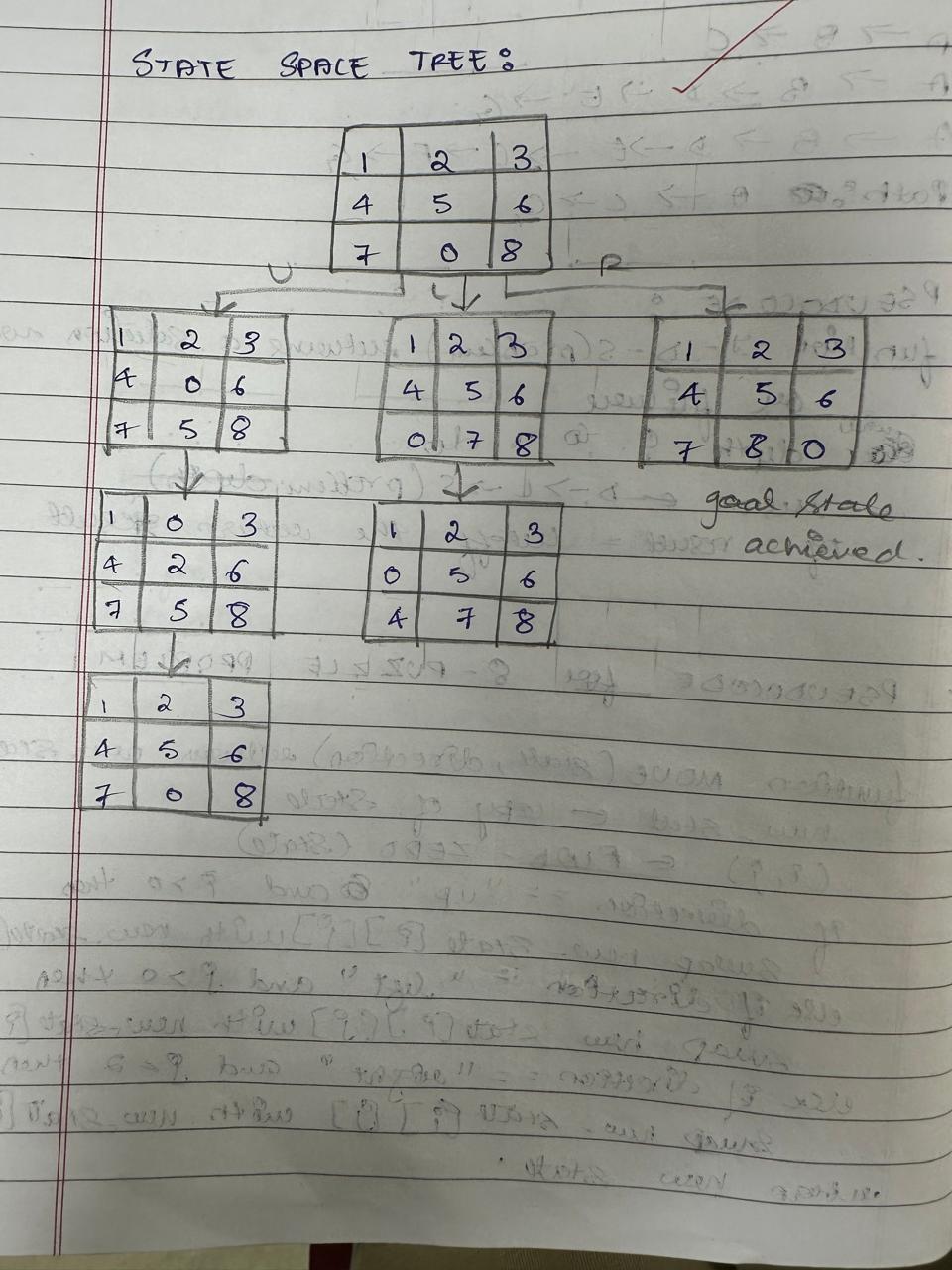
Action: Down

[1, 2, 3]

[4, 5, 6]

[7, 8, 0]

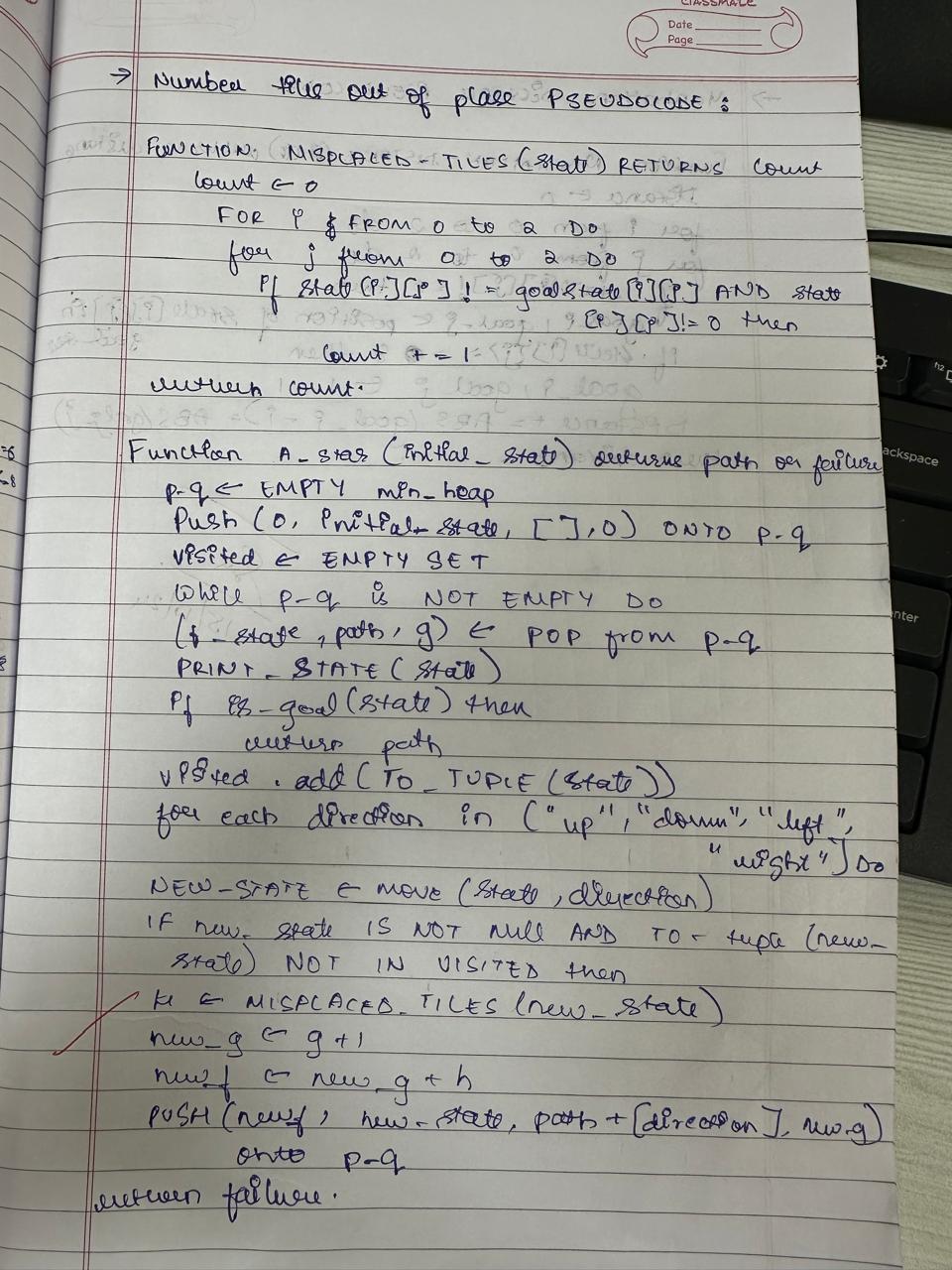
**STATE SPACE TREE:**



**LAB - 03**

**Program 1 : Heuristic - Number of Tiles Out of Place**

**OBSERVATION:**



**CODE:**

import heapq

goal\_state = [[1, 2, 3],

[8, 0, 4],

[7, 6, 5]]

def find\_zero(state):

for i in range(3):

for j in range(3):

if state[i][j] == 0:

return (i, j)

def move(state, direction):

new\_state = [row[:] for row in state]

i, j = find\_zero(state)

if direction == "up" and i > 0:

new\_state[i][j], new\_state[i-1][j] = new\_state[i-1][j], new\_state[i][j]

elif direction == "down" and i < 2:

new\_state[i][j], new\_state[i+1][j] = new\_state[i+1][j], new\_state[i][j]

elif direction == "left" and j > 0:

new\_state[i][j], new\_state[i][j-1] = new\_state[i][j-1], new\_state[i][j]

elif direction == "right" and j < 2:

new\_state[i][j], new\_state[i][j+1] = new\_state[i][j+1], new\_state[i][j]

else:

return None

return new\_state

def is\_goal(state):

return state == goal\_state

def misplaced\_tiles(state):

return sum(1 for i in range(3) for j in range(3)

if state[i][j] != goal\_state[i][j] and state[i][j] != 0)

def a\_star(initial\_state):

priority\_queue = []

heapq.heappush(priority\_queue, (0, initial\_state, [], 0))

visited = set()

while priority\_queue:

f, state, path, g = heapq.heappop(priority\_queue)

print("Exploring state in A\*:")

print\_state(state)

if is\_goal(state):

return path

visited.add(tuple(map(tuple, state)))

for direction in ["up", "down", "left", "right"]:

new\_state = move(state, direction)

if new\_state and tuple(map(tuple, new\_state)) not in visited:

h = misplaced\_tiles(new\_state)

new\_g = g + 1

new\_f = new\_g + h

heapq.heappush(priority\_queue, (new\_f, new\_state, path + [direction], new\_g))

return None

def print\_state(state):

for row in state:

print(row)

print()

def get\_initial\_state():

print("Enter the initial state of the 8-puzzle (0 for empty space):")

initial\_state = []

for i in range(3):

row = list(map(int, input(f"Enter row {i+1} (space-separated): ").strip().split()))

if len(row) != 3:

raise ValueError("Each row must contain exactly 3 numbers.")

initial\_state.append(row)

return initial\_state

if \_\_name\_\_ == "\_\_main\_\_":

initial\_state = get\_initial\_state()

print("Initial State:")

print\_state(initial\_state)

print("Solving using A\* search:")

a\_star\_solution = a\_star(initial\_state)

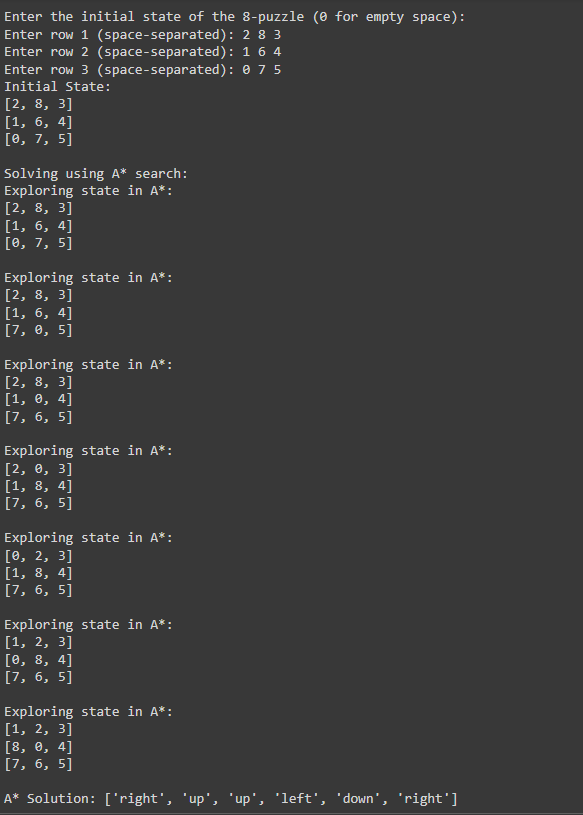
if a\_star\_solution:

print("A\* Solution:", a\_star\_solution)

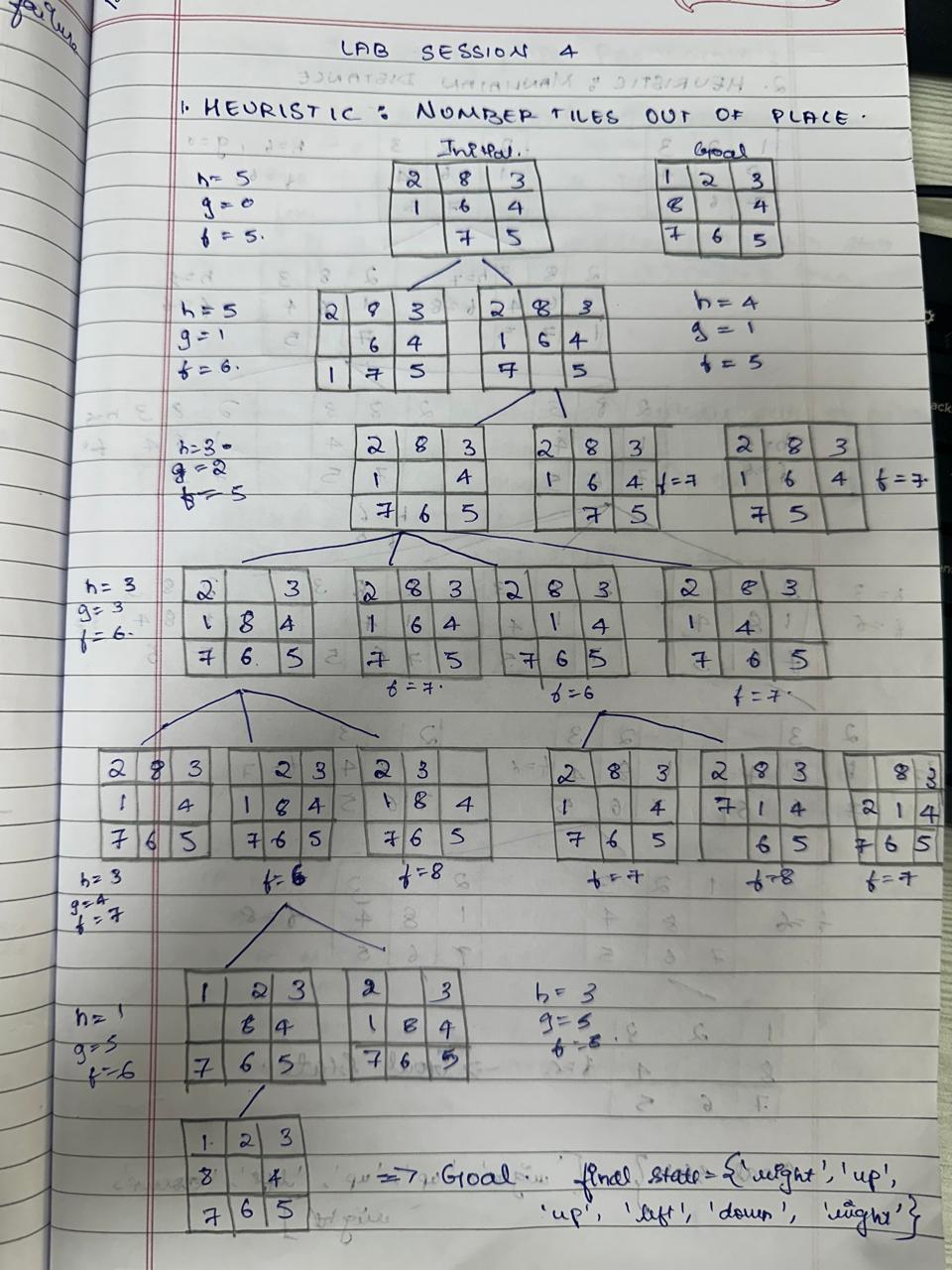
else:

print("No solution found with A\*.")

**OUTPUT:**

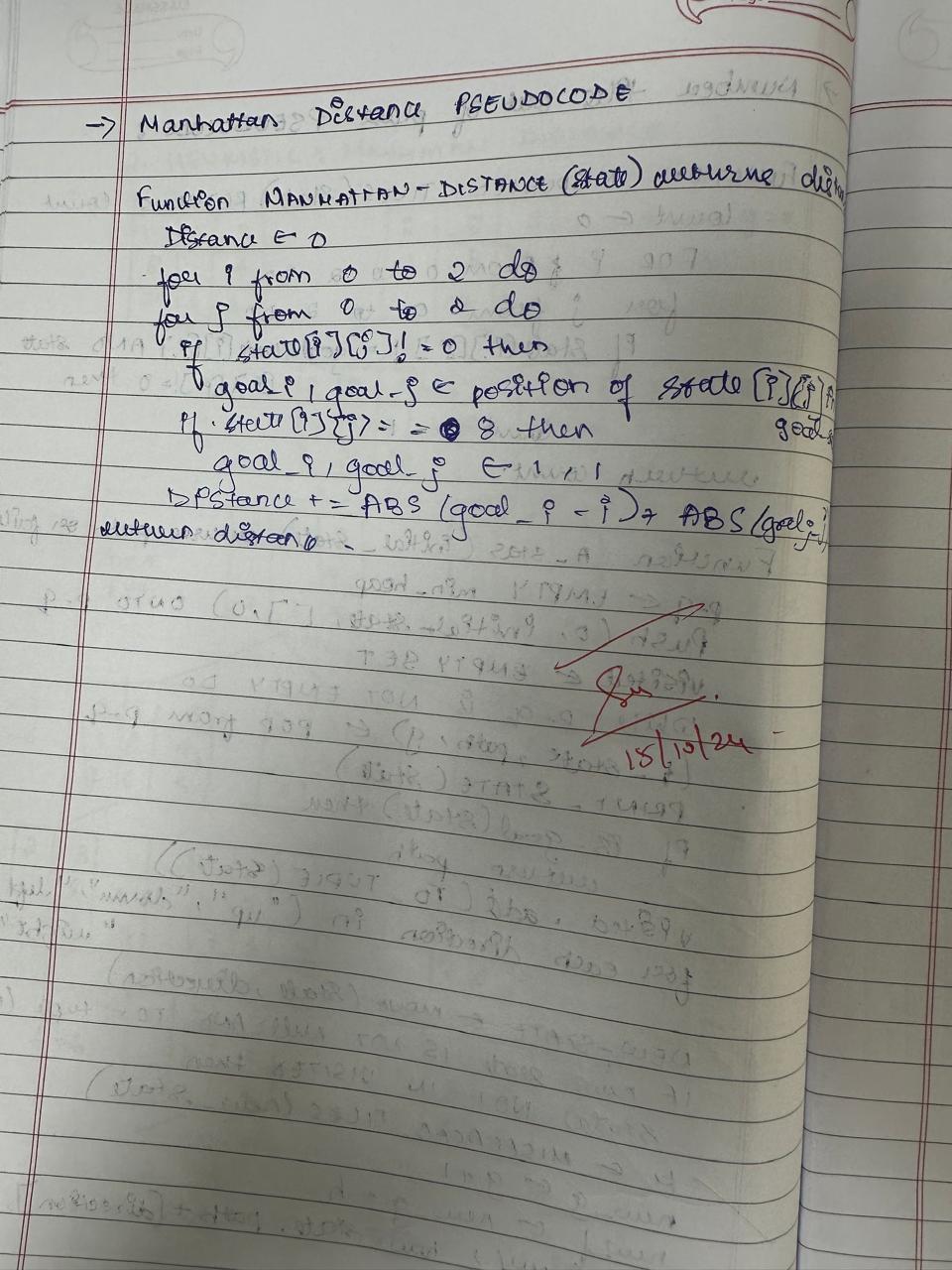


**STATE SPACE TREE:**



**Program 2 : Heuristic - Manhattan Distance**

**OBSERVATION:**



**CODE:**

import heapq

goal\_state = [[1, 2, 3],

[8, 0, 4],

[7, 6, 5]]

def find\_zero(state):

for i in range(3):

for j in range(3):

if state[i][j] == 0:

return (i, j)

def move(state, direction):

new\_state = [row[:] for row in state]

i, j = find\_zero(state)

if direction == "up" and i > 0:

new\_state[i][j], new\_state[i-1][j] = new\_state[i-1][j], new\_state[i][j]

elif direction == "down" and i < 2:

new\_state[i][j], new\_state[i+1][j] = new\_state[i+1][j], new\_state[i][j]

elif direction == "left" and j > 0:

new\_state[i][j], new\_state[i][j-1] = new\_state[i][j-1], new\_state[i][j]

elif direction == "right" and j < 2:

new\_state[i][j], new\_state[i][j+1] = new\_state[i][j+1], new\_state[i][j]

else:

return None

return new\_state

def is\_goal(state):

return state == goal\_state

def manhattan\_distance(state):

distance = 0

for i in range(3):

for j in range(3):

if state[i][j] != 0:

goal\_i, goal\_j = divmod(state[i][j] - 1, 3)

if state[i][j] == 8:

goal\_i, goal\_j = 1, 1

distance += abs(goal\_i - i) + abs(goal\_j - j)

return distance

def a\_star(initial\_state):

priority\_queue = []

heapq.heappush(priority\_queue, (0, initial\_state, [], 0))

visited = set()

while priority\_queue:

f, state, path, g = heapq.heappop(priority\_queue)

print("Exploring state in A\*:")

print\_state(state)

if is\_goal(state):

return path

visited.add(tuple(map(tuple, state)))

for direction in ["up", "down", "left", "right"]:

new\_state = move(state, direction)

if new\_state and tuple(map(tuple, new\_state)) not in visited:

h = manhattan\_distance(new\_state)

new\_g = g + 1

new\_f = new\_g + h

heapq.heappush(priority\_queue, (new\_f, new\_state, path + [direction], new\_g))

return None

def print\_state(state):

for row in state:

print(row)

print()

def get\_initial\_state():

print("Enter the initial state of the 8-puzzle (0 for empty space):")

initial\_state = []

for i in range(3):

row = list(map(int, input(f"Enter row {i+1} (space-separated): ").strip().split()))

if len(row) != 3:

raise ValueError("Each row must contain exactly 3 numbers.")

initial\_state.append(row)

return initial\_state

if \_\_name\_\_ == "\_\_main\_\_":

initial\_state = get\_initial\_state()

print("Initial State:")

print\_state(initial\_state)

print("Solving using A\* search:")

a\_star\_solution = a\_star(initial\_state)

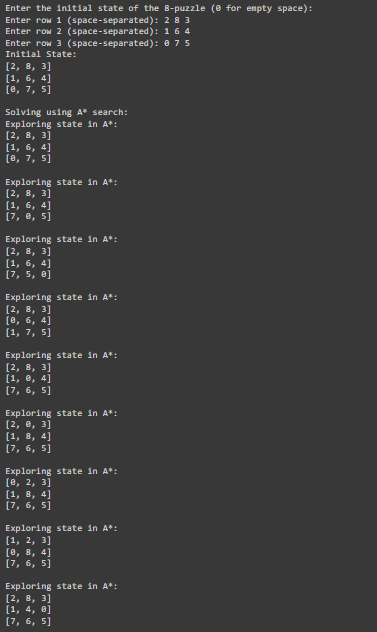
if a\_star\_solution:

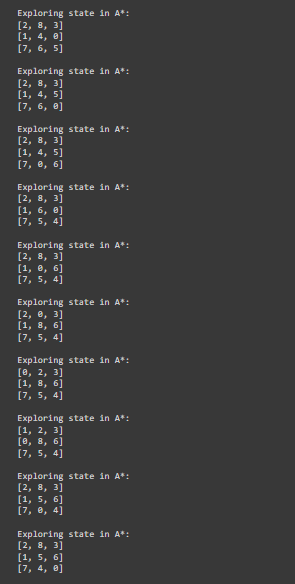
print("A\* Solution:", a\_star\_solution)

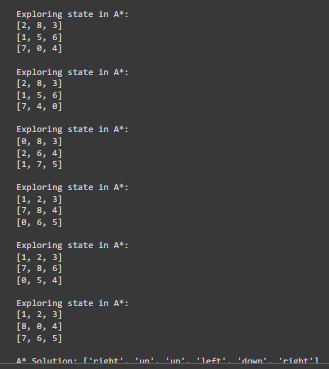
else:

print("No solution found with A\*.")

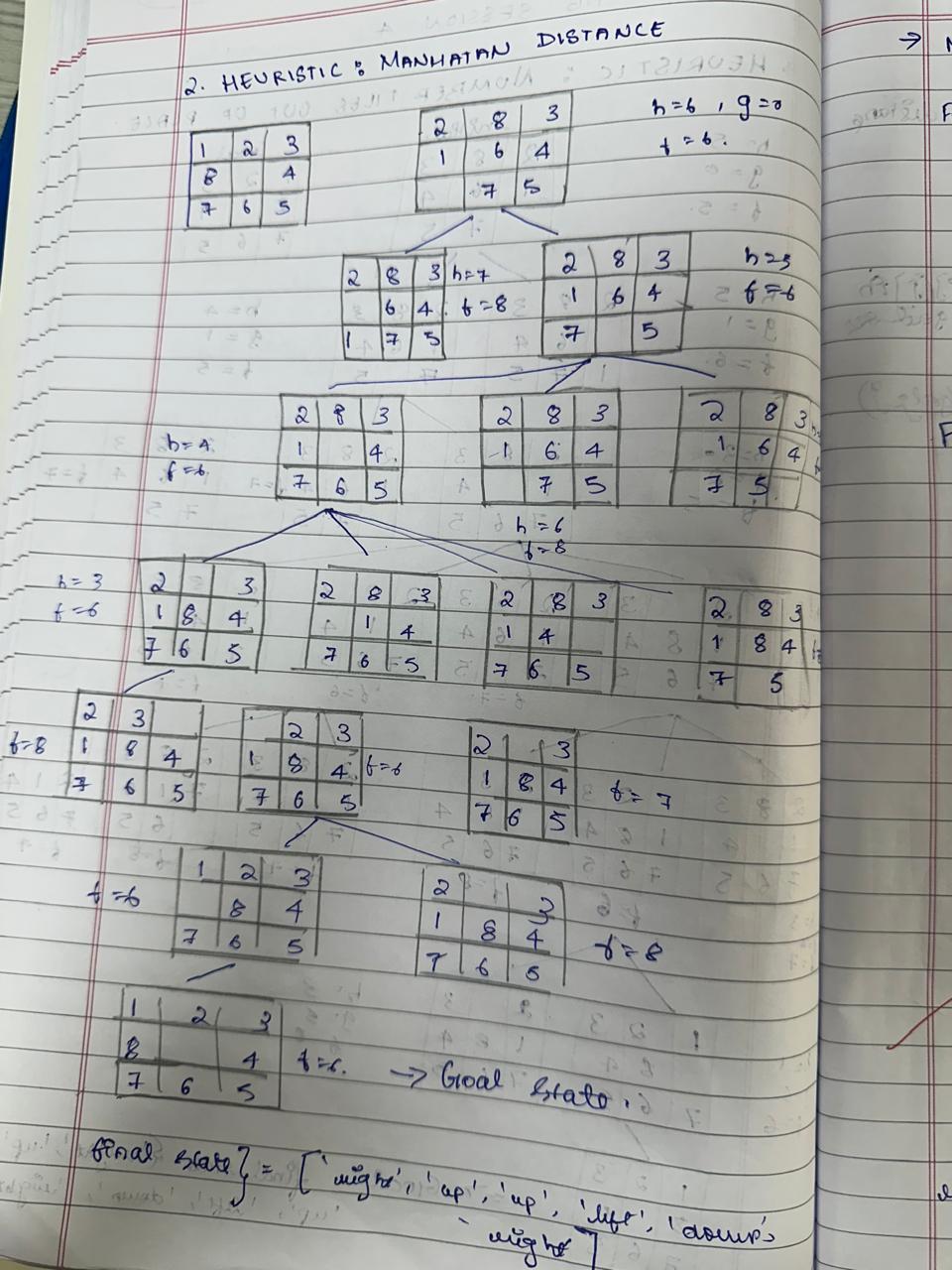
**OUTPUT:**







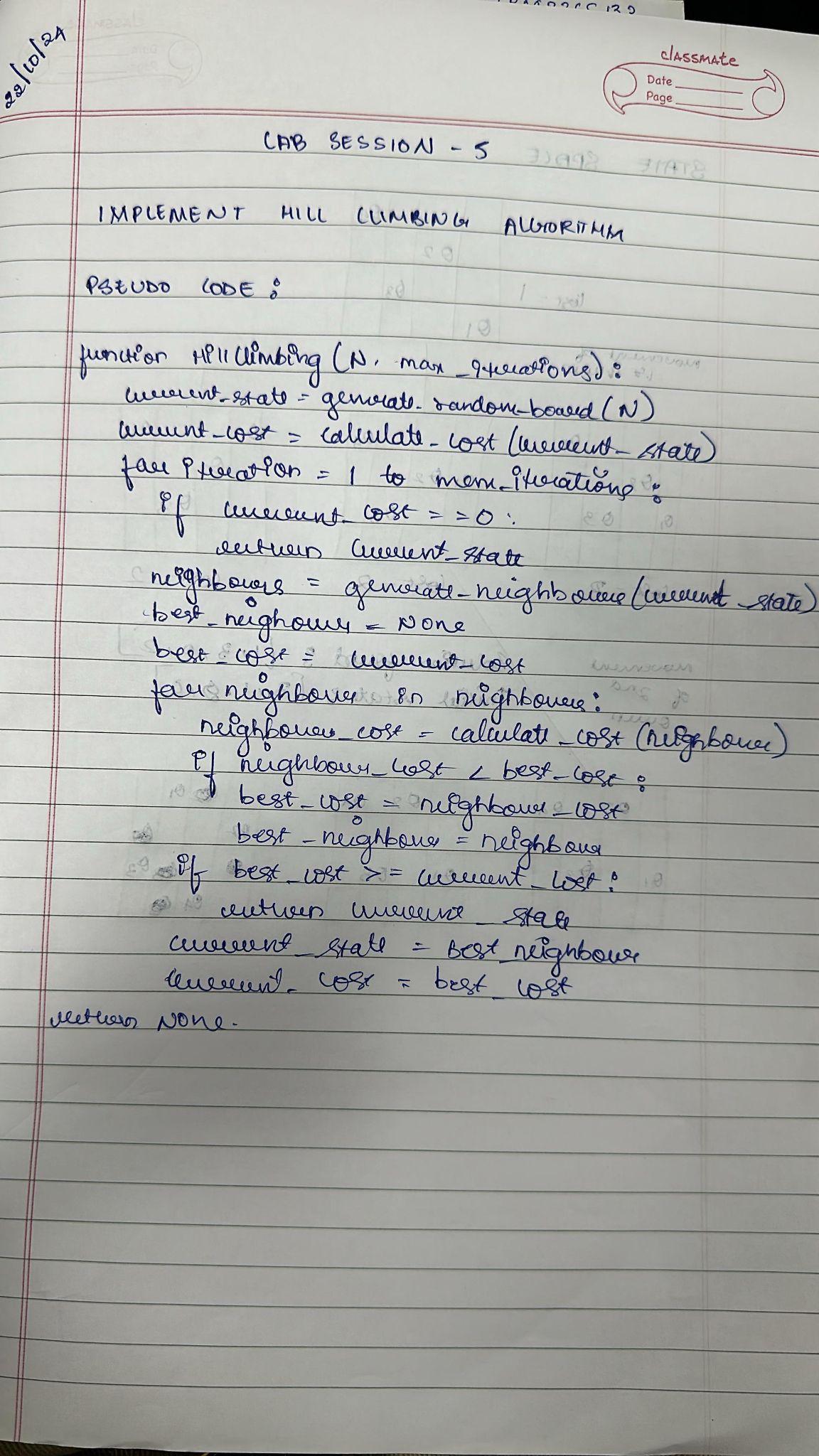
**STATE SPACE TREE:**



**LAB - 04**

**Program: Hill Climbing Algorithm for 8 Queens problem**

**OBSERVATION:**

****

**CODE:**

import random

def calculate\_cost(state):

"""Calculate the number of conflicts in the current state."""

cost = 0

n = len(state)

for i in range(n):

for j in range(i + 1, n):

if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):

cost += 1

return cost

def get\_neighbors(state):

"""Generate all possible neighbors by moving each queen in its column."""

neighbors = []

n = len(state)

for col in range(n):

for row in range(n):

if state[col] != row:

new\_state = list(state)

new\_state[col] = row

neighbors.append(new\_state)

return neighbors

def hill\_climbing(n, max\_iterations=1000):

"""Perform hill climbing search to solve the N-Queens problem."""

current\_state = [random.randint(0, n - 1) for \_ in range(n)]

current\_cost = calculate\_cost(current\_state)

for iteration in range(max\_iterations):

if current\_cost == 0:

return current\_state

neighbors = get\_neighbors(current\_state)

neighbor\_costs = [(neighbor, calculate\_cost(neighbor)) for neighbor in neighbors]

next\_state, next\_cost = min(neighbor\_costs, key=lambda x: x[1])

if next\_cost >= current\_cost:

print(f"Local maximum reached at iteration {iteration}. Restarting...")

return None

current\_state, current\_cost = next\_state, next\_cost

print(f"Iteration {iteration}: Current state: {current\_state}, Cost: {current\_cost}")

print(f"Max iterations reached without finding a solution.")

return None

try:

n = int(input("Enter the number of queens (N): "))

if n <= 0:

raise ValueError("N must be a positive integer.")

except ValueError as e:

print(e)

n = 4

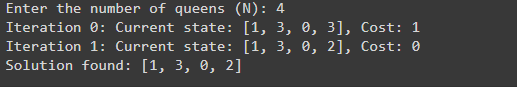
solution = None

while solution is None:

solution = hill\_climbing(n)

print(f"Solution found: {solution}")

**OUTPUT:**



**STATE SPACE TREE:**

