AI-LAB REPORT

**IMPLEMENT TIC TAC TOE:**

#TIC TAC TOE GAME

import random

def sum(a, b, c):

    return a + b + c

def printBoard(xState, zState):

    zero = 'X' if xState[0] else ('O' if zState[0] else 0)

    one = 'X' if xState[1] else ('O' if zState[1] else 1)

    two = 'X' if xState[2] else ('O' if zState[2] else 2)

    three = 'X' if xState[3] else ('O' if zState[3] else 3)

    four = 'X' if xState[4] else ('O' if zState[4] else 4)

    five = 'X' if xState[5] else ('O' if zState[5] else 5)

    six = 'X' if xState[6] else ('O' if zState[6] else 6)

    seven = 'X' if xState[7] else ('O' if zState[7] else 7)

    eight = 'X' if xState[8] else ('O' if zState[8] else 8)

    print(f"{zero} | {one} | {two}")

    print("--|---|---")

    print(f"{three} | {four} | {five}")

    print("--|---|---")

    print(f"{six} | {seven} | {eight}")

def checkWin(xState, zState):

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

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

            [0, 4, 8], [2, 4, 6]]

    for win in wins:

        if sum(xState[win[0]], xState[win[1]], xState[win[2]]) == 3:

            print("X Won the match")

            return 1

        if sum(zState[win[0]], zState[win[1]], zState[win[2]]) == 3:

            print("O Won the match")

            return 0

    return -1

def getAvailableMoves(state):

    return [i for i in range(9) if state[i] == 0]

def computerMove(zState):

    available\_moves = getAvailableMoves(zState)

    return random.choice(available\_moves)

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

    xState = [0] \* 9

    zState = [0] \* 9

    turn = 1  # 1 for X (user) and 0 for O (computer)

    print("Welcome to Tic Tac Toe")

    while True:

        printBoard(xState, zState)

        if turn == 1:  # User's turn (X)

            value = int(input("X's Chance. Please enter a value (0-8): "))

            while value not in getAvailableMoves(xState):

                print("Invalid move. Try again.")

                value = int(input("Please enter a value (0-8): "))

            xState[value] = 1

        else:  # Computer's turn (O)

            print("O's Chance (Computer's Move)")

            value = computerMove(zState)

            zState[value] = 1

            print(f"Computer placed O in position {value}")

        cwin = checkWin(xState, zState)

        if cwin != -1:

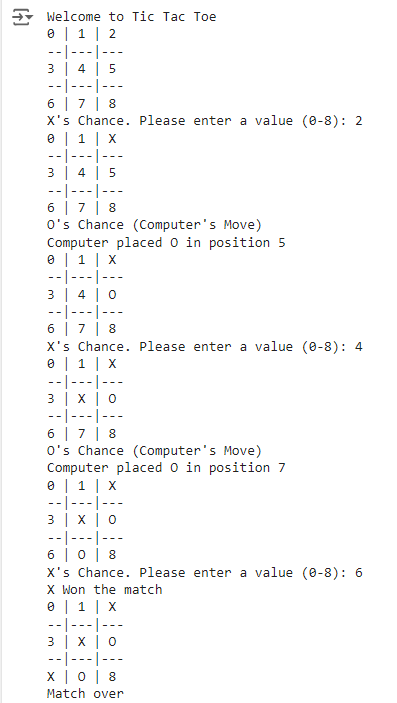
            printBoard(xState, zState)

            print("Match over")

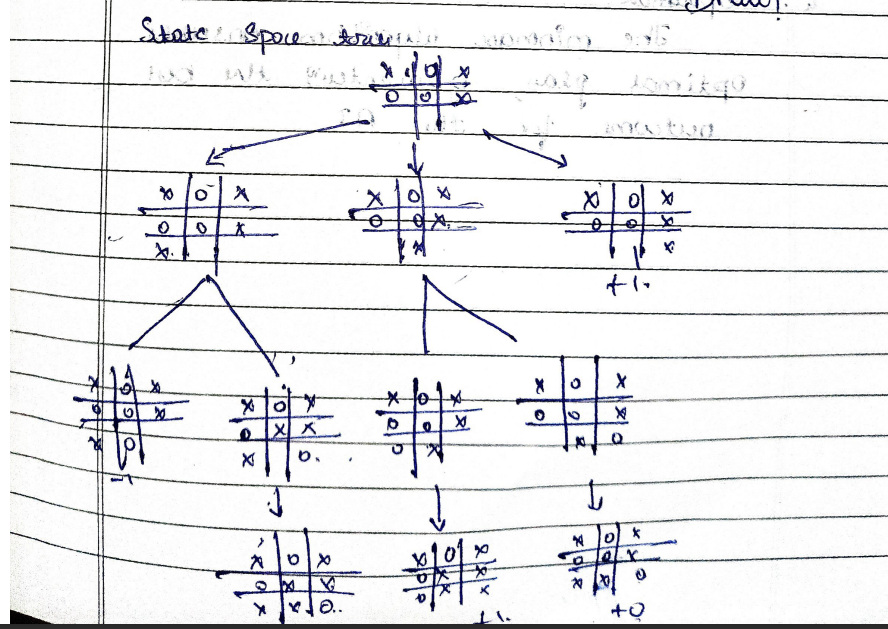
            break

        turn = 1 - turn  # Switch turn

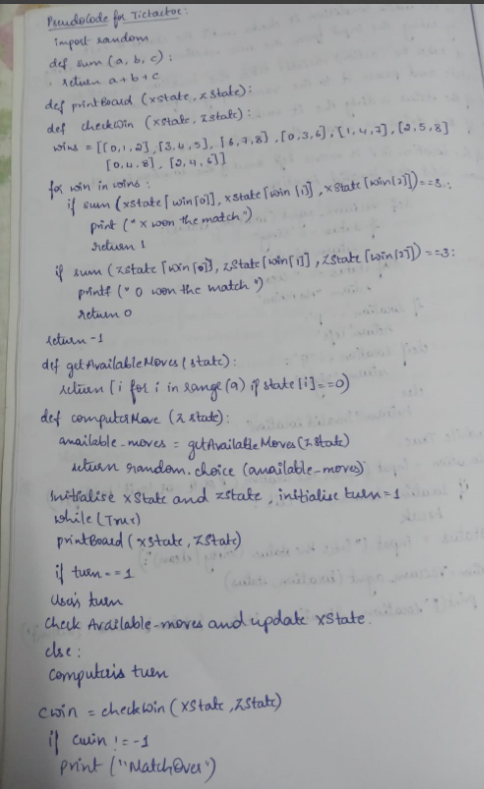
**OUTPUT:**



**STAE SPACE TREE:**



**ALGORITHM OR PSEUDO CODE :**

****

**IMPEMENT VACCUME AGENT PROGRAM:**

def vacuum\_agent(location, status):

    if status == "Dirty":

        return "Suck"

    elif status == "Clean":

        return "No Action"

    if location == "P":

        return "left"

    elif location == "Q":

        return "right"

    else:

        return "Invalid Location"

while True:

    location = input("Enter the location (P or Q, or 'exit' to stop): ")

    if location.lower() == 'exit':

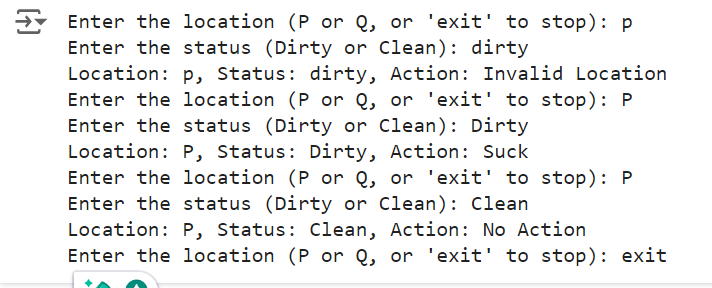
        break

    status = input("Enter the status (Dirty or Clean): ")

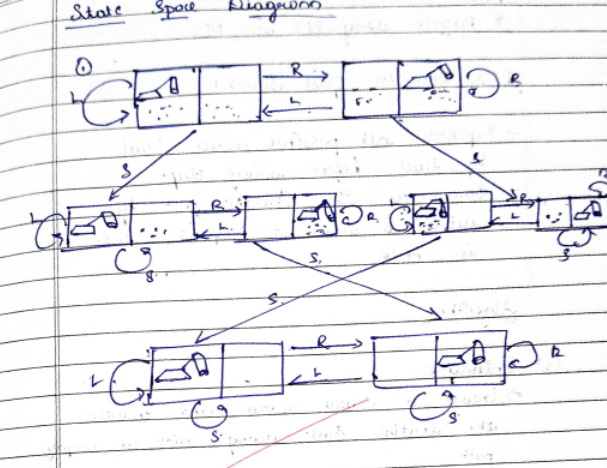
    action = vacuum\_agent(location, status)

    print(f"Location: {location}, Status: {status}, Action: {action}")

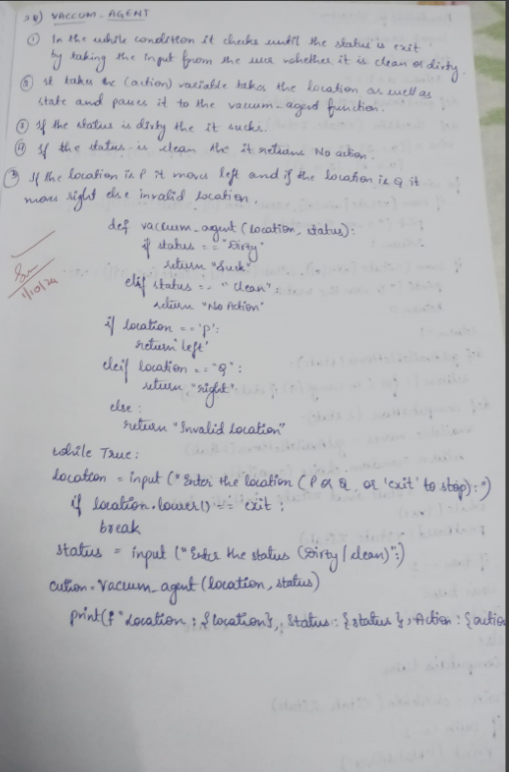
**OUTPUT:**

****

**STATE SPACE TREE:**

****

**ALGORITHM OR PSEUDO CODE:**

****

**SOLVE 8 PUZZLE PROBLEM USING DFS:**

class Node:

    def \_\_init\_\_(self, puzzle, x, y, parent=None):

        self.puzzle = [row[:] for row in puzzle]

        self.x = x

        self.y = y

        self.parent = parent

goal = [

    [1, 2, 3],

    [4, 5, 6],

    [7, 8, 0]

]

dx = [-1, 1, 0, 0]

dy = [0, 0, -1, 1]

def is\_goal(puzzle):

    return puzzle == goal

def print\_puzzle(puzzle):

    for row in puzzle:

        print(' '.join(str(x) if x != 0 else ' ' for x in row))

    print()

def is\_valid(x, y):

    return 0 <= x < 3 and 0 <= y < 3

def count\_inversions(puzzle):

    flat\_puzzle = [num for row in puzzle for num in row if num != 0]

    inversions = 0

    for i in range(len(flat\_puzzle)):

        for j in range(i + 1, len(flat\_puzzle)):

            if flat\_puzzle[i] > flat\_puzzle[j]:

                inversions += 1

    return inversions

def is\_solvable(puzzle):

    return count\_inversions(puzzle) % 2 == 0

def dfs(root):

    stack = [root]

    visited = set()

    while stack:

        node = stack.pop()

        if is\_goal(node.puzzle):

            print("Solution found:")

            path = []

            while node:

                path.append(node.puzzle)

                node = node.parent

            for state in reversed(path):

                print\_puzzle(state)

            return True

        puzzle\_tuple = tuple(map(tuple, node.puzzle))

        if puzzle\_tuple in visited:

            continue

        visited.add(puzzle\_tuple)

        for i in range(4):

            new\_x = node.x + dx[i]

            new\_y = node.y + dy[i]

            if is\_valid(new\_x, new\_y):

                new\_puzzle = [row[:] for row in node.puzzle]

                new\_puzzle[node.x][node.y], new\_puzzle[new\_x][new\_y] = new\_puzzle[new\_x][new\_y], new\_puzzle[node.x][node.y]

                new\_puzzle\_tuple = tuple(map(tuple, new\_puzzle))

                if new\_puzzle\_tuple not in visited:

                    new\_node = Node(new\_puzzle, new\_x, new\_y, node)

                    stack.append(new\_node)

    print("No solution found.")

    return False

def main():

    initial\_puzzle = [

        [1, 2, 3],

        [4, 5, 6],

        [7, 0, 8]

    ]

    print("Initial Puzzle:")

    print\_puzzle(initial\_puzzle)

    if not is\_solvable(initial\_puzzle):

        print("The provided puzzle is unsolvable.")

        return

    try:

        x, y = [(i, j) for i in range(3) for j in range(3) if initial\_puzzle[i][j] == 0][0]

    except IndexError:

        print("Invalid puzzle: No blank space (0) found.")

        return

    root = Node(initial\_puzzle, x, y)

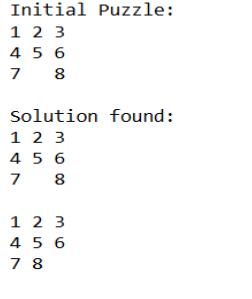
    if not dfs(root):

        print("Failed to find a solution.")

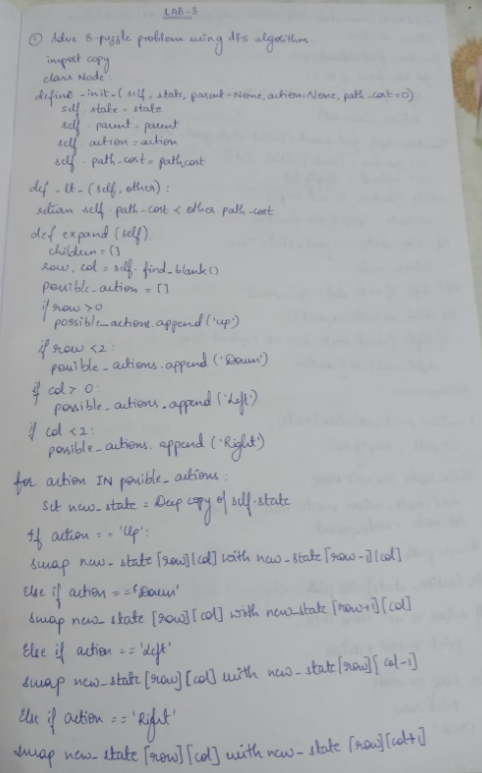
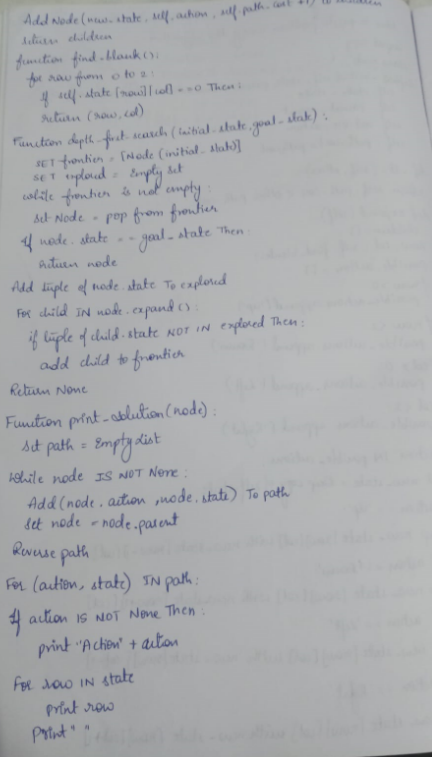
if \_\_name\_\_ == "\_\_main\_\_":

    main()

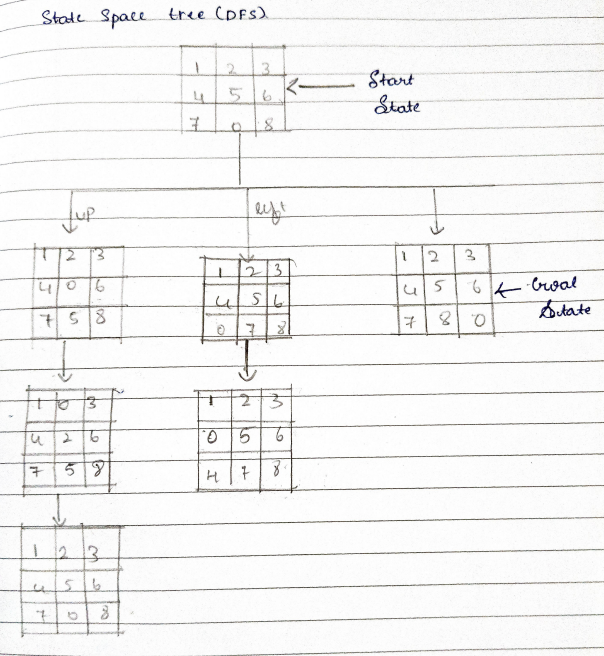
**OUTPUT:**



**ALGORITHM OR PSEUDO CODE:**

**** ****

**STATE SPACE TREE:**



**ITERATIVE DEEPENING DFS:**

def iterative\_depth\_search(graph, start, goal):

    """

    Performs iterative depth search to find a path from the start node to the goal node.

    Args:

        graph: A dictionary representing the graph.

        start: The starting node.

        goal: The goal node.

    Returns:

        A list representing the path from the start node to the goal node, or None if no path exists.

    """

    depth\_limit = 1

    while True:

        print(f"Exploring depth limit: {depth\_limit}")  # Show current depth limit

        stack = [(start, [start])]

        while stack:

            node, path = stack.pop()

            print(f"Current node: {node}, Current path: {path}")  # Show current node and path

            if node == goal:

                return path

            if len(path) < depth\_limit:  # Check if the current path length is less than the depth limit

                for neighbor in graph[node]:

                    if neighbor not in path:  # Avoid cycles

                        stack.append((neighbor, path + [neighbor]))

        depth\_limit += 1  # Increment depth limit for the next iteration

# Example usage:

graph = {

    'A': ['B', 'C'],

    'B': ['D', 'E'],

    'C': ['F', 'G'],

    'D': ['H'],

    'E': ['I'],

    'F': [],

    'G': ['J'],

    'J': []

}

start\_node = 'A'

goal\_node = 'G'

path = iterative\_depth\_search(graph, start\_node, goal\_node)

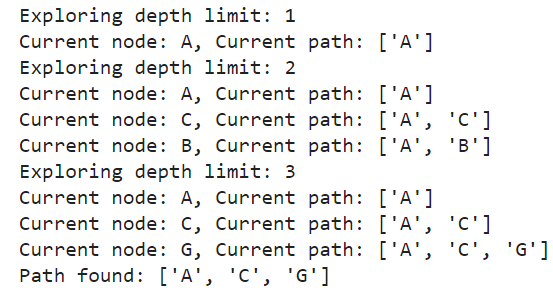
if path:

    print("Path found:", path)

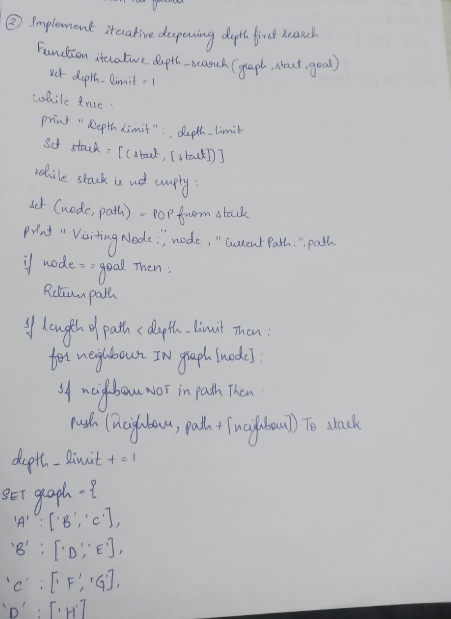
else:

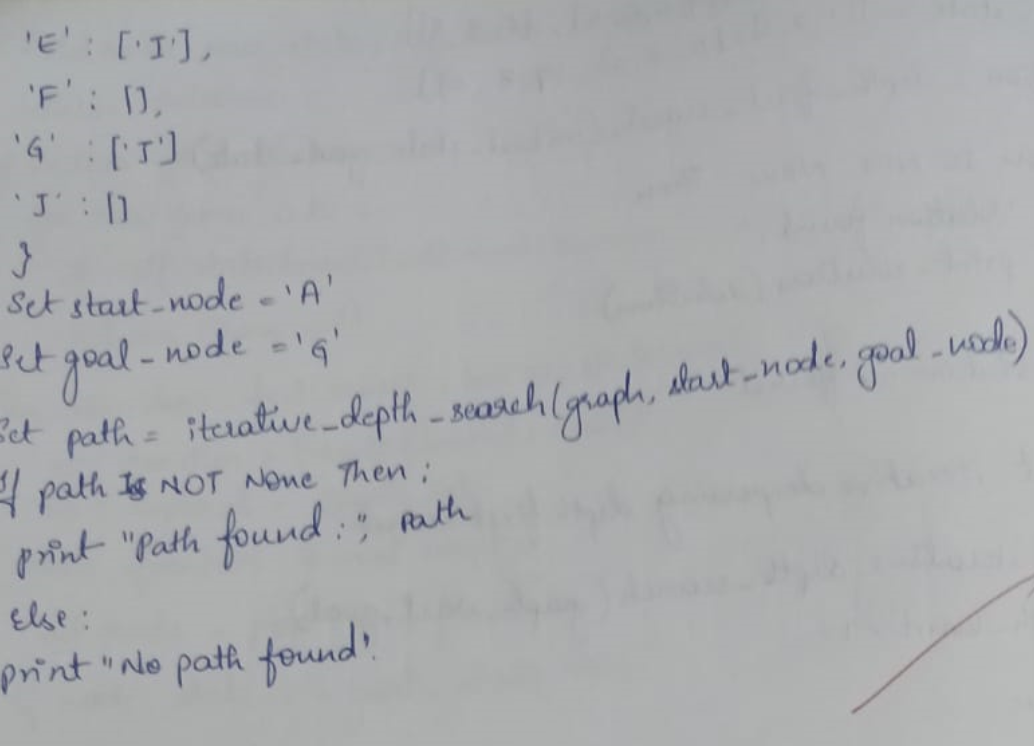
    print("No path found.")

**OUTPUT:**

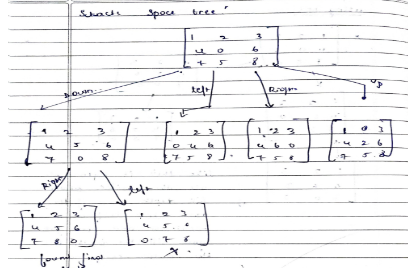
****

**ALGORITHM OR PSEUDO CODE:**

****

****

**STATE SPACE TREE:**

****

**IMPLEMENT A\* ALGORITHM(MISPLACED TILES):**

import heapq

class PuzzleState:

    def \_\_init\_\_(self, board, g, h, move=None, previous=None):

        self.board = board

        self.g = g  # cost to reach this node

        self.h = h  # heuristic cost (number of misplaced tiles)

        self.f = g + h  # total cost

        self.move = move  # store the move made to reach this state

        self.previous = previous  # track the previous state for path reconstruction

    def \_\_lt\_\_(self, other):

        return self.f < other.f  # for priority queue

    def get\_blank\_position(self):

        return divmod(self.board.index(0), 3)

    def generate\_successors(self):

        successors = []

        x, y = self.get\_blank\_position()

        directions = [(-1, 0, 'up'), (1, 0, 'down'), (0, -1, 'left'), (0, 1, 'right')]  # up, down, left, right

        for dx, dy, move in directions:

            new\_x, new\_y = x + dx, y + dy

            if 0 <= new\_x < 3 and 0 <= new\_y < 3:

                new\_board = self.board[:]

                # Swap the blank space with the adjacent tile

                new\_board[x \* 3 + y], new\_board[new\_x \* 3 + new\_y] = new\_board[new\_x \* 3 + new\_y], new\_board[x \* 3 + y]

                successors.append((new\_board, move))  # Append the new board and move

        return successors

    def count\_misplaced\_tiles(self):

        return sum(1 for i in range(9) if self.board[i] != (i + 1) % 9)  # Correctly count misplaced tiles

def a\_star(initial\_board, goal\_board):

    initial\_h = PuzzleState(initial\_board, 0, 0).count\_misplaced\_tiles()

    initial\_state = PuzzleState(initial\_board, 0, initial\_h)

    open\_set = []

    heapq.heappush(open\_set, initial\_state)

    closed\_set = set()

    while open\_set:

        current\_state = heapq.heappop(open\_set)

        # Check if we reached the goal state

        if current\_state.board == goal\_board:

            return current\_state  # Return the goal state to reconstruct the path

        closed\_set.add(tuple(current\_state.board))

        for successor\_board, move in current\_state.generate\_successors():

            if tuple(successor\_board) in closed\_set:

                continue

            g\_cost = current\_state.g + 1

            h\_cost = PuzzleState(successor\_board, 0, 0).count\_misplaced\_tiles()

            successor\_state = PuzzleState(successor\_board, g\_cost, h\_cost, move, current\_state)

            # Add to open set if not already present

            if not any(successor\_state.board == state.board for state in open\_set):

                heapq.heappush(open\_set, successor\_state)

    return None  # If no solution found

def print\_solution(solution\_state):

    path = []

    moves = []

    while solution\_state:

        path.append(solution\_state.board)

        moves.append(solution\_state.move)

        solution\_state = solution\_state.previous

    for step, move in zip(reversed(path), reversed(moves)):

        print\_board(step)

        if move:

            print("Move:", move)

    print("Solution found in", len(path) - 1, "moves.")

def print\_board(board):

    for i in range(3):

        print(board[i \* 3:(i + 1) \* 3])

    print()

# Function to input board states from the user

def input\_board(prompt):

    print(prompt)

    board = []

    for \_ in range(3):  # Loop for 3 rows

        row = list(map(int, input().strip().split()))

        if len(row) != 3:

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

        board.extend(row)

    if len(board) != 9:

        raise ValueError("The board must contain exactly 9 numbers (including the blank tile).")

    return board

# Example usage:

try:

    initial\_board = input\_board("Enter the start state matrix (3 rows, space-separated):")

    goal\_board = input\_board("Enter the goal state matrix (3 rows, space-separated):")

    solution = a\_star(initial\_board, goal\_board)

    if solution:

        print\_solution(solution)

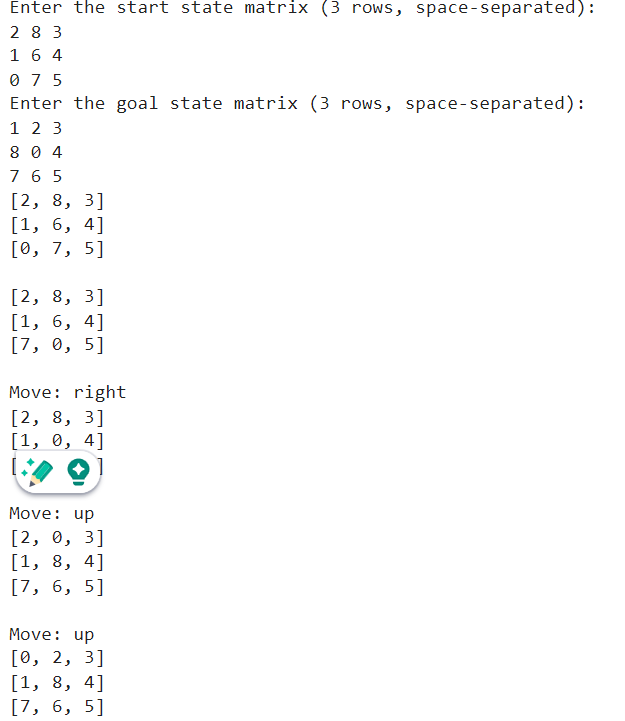
    else:

        print("No solution found.")

except Exception as e:

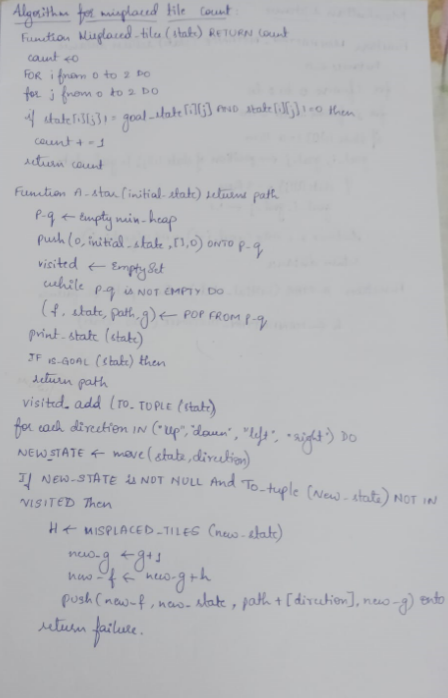
    print("Error:", str(e))

**OUTPUT:**

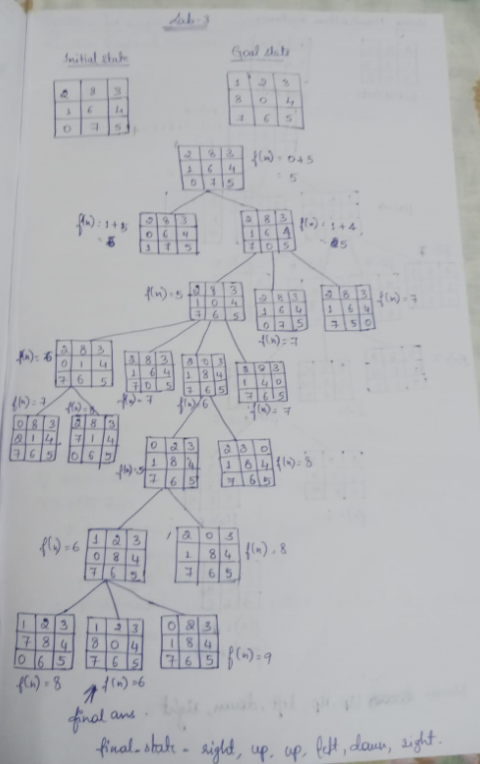
****

****

**ALGORITHM OR PSEUDO CODE:**

****

**STATE SPACE TREE:**

****

**IMPLEMENT A\* ALGORITHM(MANHATTAN DISTANCE):**

import heapq

# Class to represent the state of the 8-puzzle

class PuzzleState:

    def \_\_init\_\_(self, board, parent=None, move="", g=0, h=0):

        self.board = board  # current board configuration

        self.parent = parent  # parent state

        self.move = move  # move made to reach this state

        self.g = g  # cost from start to this state (depth of the node)

        self.h = h  # heuristic cost to goal (Manhattan distance)

        self.f = g + h  # total cost (f = g + h)

    # Defining comparison functions for priority queue

    def \_\_lt\_\_(self, other):

        return self.f < other.f

# Function to print the board in a readable format

def print\_board(board):

    for row in board:

        print(row)

    print()

# Function to calculate the Manhattan distance (heuristic)

def manhattan\_distance(state, goal\_state):

    distance = 0

    flat\_goal = sum(goal\_state, [])  # Flattening the 2D list into 1D

    for i in range(3):

        for j in range(3):

            if state[i][j] != 0:

x, y = divmod(flat\_goal.index(state[i][j]), 3)

                distance += abs(i - x) + abs(j - y)

    return distance

# Function to find the position of the blank (0) tile

def find\_blank(board):

    for i in range(3):

        for j in range(3):

            if board[i][j] == 0:

                return i, j

# Function to check if the current state is the goal state

def is\_goal(state, goal\_state):

    return state == goal\_state

# Function to get the possible moves from the current state

def get\_neighbors(state):

    neighbors = []

    row, col = find\_blank(state)

    # Possible moves: Up, Down, Left, Right

    moves = [

        ('Up', (row - 1, col)),

        ('Down', (row + 1, col)),

        ('Left', (row, col - 1)),

        ('Right', (row, col + 1))

    ]

    for move, (new\_row, new\_col) in moves:

        if 0 <= new\_row < 3 and 0 <= new\_col < 3:

            new\_state = [list(row) for row in state]  # deep copy of the board

            new\_state[row][col], new\_state[new\_row][new\_col] = new\_state[new\_row][new\_col], new\_state[row][col]

            neighbors.append((new\_state, move))

    return neighbors

# Function to print the state, g, and h values

def print\_state\_info(state, g, h):

    print("State:")

    print\_board(state)

    print(f"g = {g}, h = {h}, f = {g + h}")

    print("-" \* 20)

def a\_star(start\_state, goal\_state):

    open\_list = []

    closed\_set = set()

    heapq.heappush(open\_list, PuzzleState(start\_state, None, "", 0, manhattan\_distance(start\_state, goal\_state)))

    while open\_list:

        current\_state = heapq.heappop(open\_list)

        # Print the current state, g, and h values

        print\_state\_info(current\_state.board, current\_state.g, current\_state.h)

        # Check if the current state is the goal

        if is\_goal(current\_state.board, goal\_state):

            path = []

            while current\_state.parent:

               path.append(current\_state.move)

                current\_state = current\_state.parent

            return path[::-1]  # Return the path to the goal

        closed\_set.add(tuple(map(tuple, current\_state.board)))

        # Explore neighbors

        for neighbor, move in get\_neighbors(current\_state.board):

            if tuple(map(tuple, neighbor)) not in closed\_set:

                g = current\_state.g + 1  # Increment cost

                h = manhattan\_distance(neighbor, goal\_state)

                heapq.heappush(open\_list, PuzzleState(neighbor, current\_state, move, g, h))

    return None  # If no solution is found

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":  # Corrected the method to check for main execution

    # Initial state of the 8-puzzle (0 represents the blank tile)

    start\_state = [

        [1, 2, 3],

        [4, 0, 5],

        [7, 8, 6]

    ]

    # Goal state of the 8-puzzle

    goal\_state = [

        [1, 2, 3],

        [4, 5, 6],

        [7, 8, 0]

    ]

    print("Start state:")

    print\_board(start\_state)

    print("Goal state:")

    print\_board(goal\_state)

    # Run A\* algorithm

    solution = a\_star(start\_state, goal\_state)

    if solution:

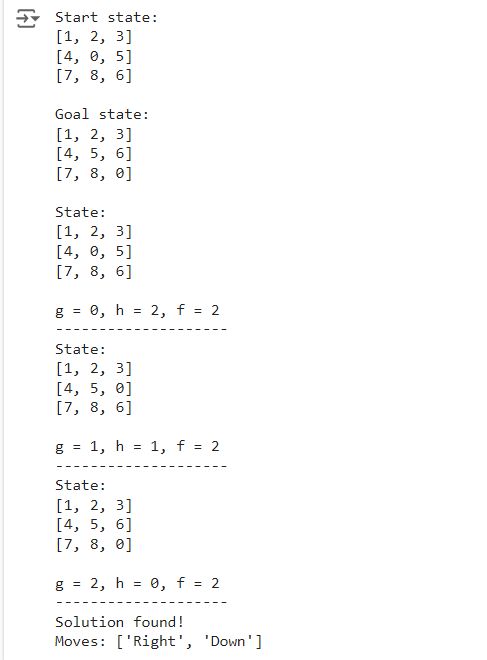
        print("Solution found!")

        print("Moves:", solution)

    else:

        print("No solution found.")

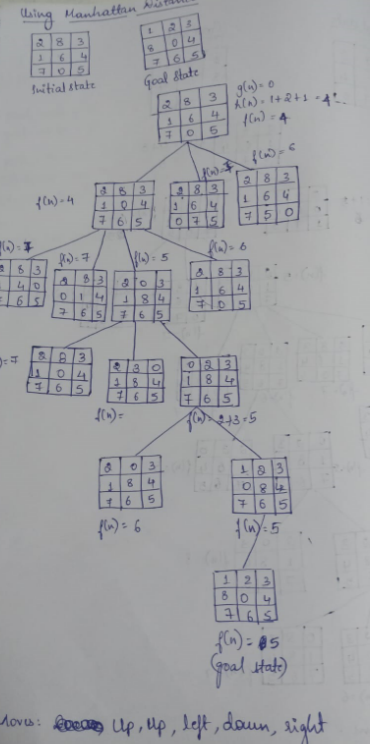
**OUTPUT:**

****

**ALGORITHM OR PSEUDO CODE:**

****

**STATE SPACE TREE:**

****

**IMPLEMENT SIMULATED ANNEALING ALGORITHM:**

import random

import math

import matplotlib.pyplot as plt

def calculate\_cost(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):

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 simulated\_annealing\_with\_tracking(initial\_state,

schedule, max\_iterations=1000):

current\_state = initial\_state

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

costs = []

temperatures = []

for t in range(max\_iterations):

T = schedule(t)

if T == 0:

break

if current\_cost == 0:

costs.append(current\_cost)

temperatures.append(T)

print(f"Solution found at iteration {t}:

{current\_state} with cost {current\_cost}")

break

neighbors = get\_neighbors(current\_state)

next\_state = random.choice(neighbors)

next\_cost = calculate\_cost(next\_state)

ΔE = next\_cost - current\_cost

acceptance\_probability = math.exp(-ΔE / T) if T > 0 else 0

accept = ΔE < 0 or random.random() <

acceptance\_probability

print(f"Iteration {t}:")

print(f" Current state: {current\_state}, Cost:

{current\_cost}, Temperature (T): {T}")

print(f" Next state: {next\_state}, Next cost:

{next\_cost}")

print(f" ΔE = {ΔE}")

print(f" Acceptance probability:

{acceptance\_probability}")

print(f" Acceptance condition met: {accept}")

costs.append(current\_cost)

temperatures.append(T)

if accept:

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

next\_cost

costs.append(current\_cost)

temperatures.append(T)

return costs, temperatures

def linear\_schedule(t, initial\_temp=1000,

final\_temp=1, max\_iter=1000):

return max(final\_temp, initial\_temp - (initial\_temp -

final\_temp) \* (t / max\_iter))

try:

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

if n <= 0:

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

initial\_state = list(map(int, input(f"Enter the initial

state as a list of {n} integers (rows for each column):

").split()))

if len(initial\_state) != n or any(not (0 <= row < n)

for row in initial\_state):

raise ValueError(f"Invalid initial state. Please

provide {n} integers between 0 and {n-1}.")

except ValueError as e:

print(e)

n = 4

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

range(n)]

print(f"Using random initial state: {initial\_state}")

costs, temperatures =

simulated\_annealing\_with\_tracking(initial\_state,

linear\_schedule)

plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)

plt.plot(costs, label="Objective Function (Cost)")

plt.xlabel("Iterations")

plt.ylabel("Objective Function (Cost)")

plt.title("Objective Function (Cost) over Iterations")

plt.legend()

plt.subplot(1, 2, 2)

plt.plot(temperatures, costs, label="Objective Function

(Cost)")

plt.xlabel("Temperature")

plt.ylabel("Objective Function (Cost)")

plt.title("Objective Function (Cost) over Temperature")

plt.legend()

plt.tight\_layout()

plt.show()

if costs[-1] == 0:

print(f"Solution found: {initial\_state}")

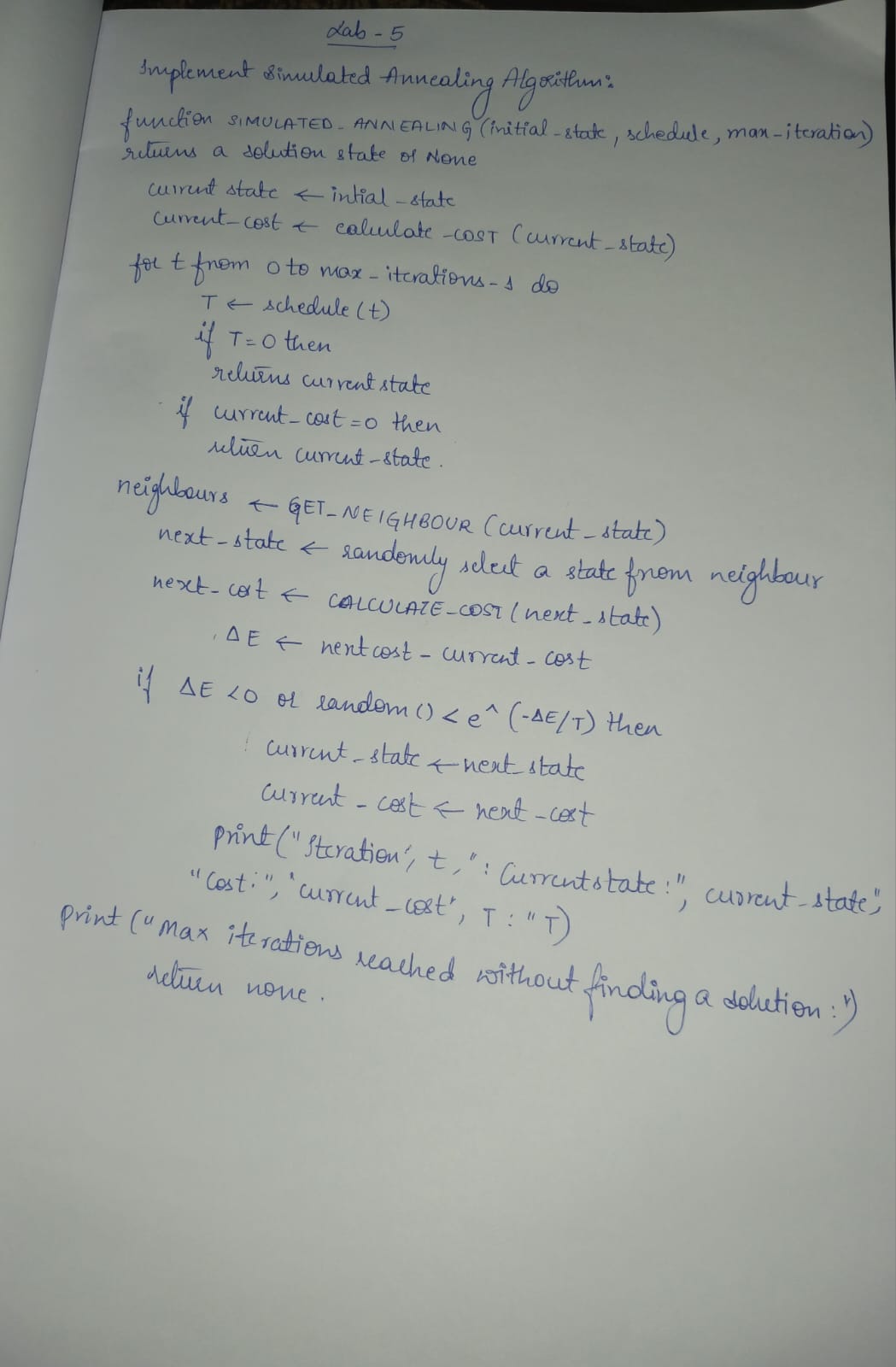
else:

print("Max iterations reached without finding a

solution.")

**OUTPUT:**

**ALGORITHM:**

****

**STATE SPACE TREE:**